Chapel Language Specification
Version 0.82

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

October 11, 2011

Contents

1 Scope 13

2 Notation 15

3 Organization 17

4 Acknowledgments 19

5 Language Overview 21
5.1 Guiding Principles 21
5.1.1 General Parallel Programming 21
5.1.2 Locality-Aware Programming 22
5.1.3 Object-Oriented Programming 22
5.1.4 Generic Programming 23
5.2 Getting Started 23

6 Lexical Structure 25
6.1 Comments 25
6.2 White Space 25
6.3 Case Sensitivity 25
6.4 Tokens 25
6.4.1 Identifiers 26
6.4.2 Keywords 26
6.4.3 Literals 27
6.4.4 Operators and Punctuation 29
6.4.5 Grouping Tokens 29

7 Types 31
7.1 Primitive Types 31
7.1.1 The Void Type 32
7.1.2 The Bool Type 32
7.1.3 Signed and Unsigned Integral Types 32
7.1.4 Real Types 33
7.1.5 Imaginary Types 33
7.1.6 Complex Types 33
7.1.7 The String Type 33
7.2 Enumerated Types 34
7.3 Locale Types 34
7.4 Structured Types 35
7.4.1 Class Types 35
7.4.2 Record Types 35
7.4.3 Union Types 35
7.4.4 Tuple Types 35
7.5 Data Parallel Types 36
7.5.1 Range Types 36
7.5.2 Domain, Array, and Index Types 36

7.6 Synchronisation Types 36
7.7 Type Aliases 36

8 Variables 39
8.1 Variable Declarations 39
8.1.1 Default Initialization 40
8.1.2 Local Type Inference 41
8.1.3 Multiple Variable Declarations 41
8.2 Global Variables 42
8.3 Local Variables 42
8.4 Constants 43
8.4.1 Compile-Time Constants 43
8.4.2 Runtime Constants 44
8.5 Configuration Variables 44

9 Conversions 45
9.1 Implicit Conversions 45
9.1.1 Implicit Numeric, Bool and Enumeration Conversions 46
9.1.2 Implicit Compile-Time Constant Conversions 47
9.1.3 Implicit Statement Bool Conversions 47
9.2 Explicit Conversions 47
9.2.1 Explicit Numeric Conversions 47
9.2.2 Explicit Enumeration Conversions 48
9.2.3 Explicit Class Conversions 49
9.2.4 Explicit Record Conversions 49

10 Expressions 51
10.1 Literal Expressions 51
10.2 Variable Expressions 52
10.3 Enumeration Constant Expression 52
10.4 Parenthesized Expressions 52
10.5 Call Expressions 52
10.6 Indexing Expressions 53
10.7 Member Access Expressions 53
10.8 The Query Expression 53
10.9 Casts 54
10.10 LValue Expressions 54
10.11 Precedence and Associativity 54
10.12 Operator Expressions 57
10.13 Arithmetic Operators 57
10.13.1 Unary Plus Operators 57
10.13.2 Unary Minus Operators 58
10.13.3 Addition Operators 58
10.13.4 Subtraction Operators 59
10.13.5 Multiplication Operators 59
10.13.6 Division Operators 60
10.13.7 Modulus Operators 61
10.13.8 Exponentiation Operators 62
10.14 Bitwise Operators 62
10.14.1 Bitwise Complement Operators 62
10.14.2 Bitwise And Operators 62

- 10.14.3 Bitwise Or Operators . . . 63
- 10.14.4 Bitwise Xor Operators . . . 63
- 10.15 Shift Operators . . . 64
- 10.16 Logical Operators . . . 64
- 10.16.1 The Logical Negation Operator . . . 65
- 10.16.2 The Logical And Operator . . . 65
- 10.16.3 The Logical Or Operator . . . 65
- 10.17 Relational Operators . . . 65
- 10.17.1 Ordered Comparison Operators . . . 66
- 10.17.2 Equality Comparison Operators . . . 67
- 10.18 Miscellaneous Operators . . . 67
- 10.18.1 The String Concatenation Operator . . . 68
- 10.18.2 The By Operator . . . 68
- 10.18.3 The Range Count Operator . . . 68
- 10.19 Let Expressions . . . 68
- 10.20 Conditional Expressions . . . 69
- 10.21 For Expressions . . . 69
- 10.21.1 Filtering Predicates in For Expressions . . . 69

11 Statements **71**
- 11.1 Blocks . . . 72
- 11.2 Expression Statements . . . 72
- 11.3 Assignment Statements . . . 73
- 11.4 The Swap Statement . . . 73
- 11.5 The Conditional Statement . . . 74
- 11.6 The Select Statement . . . 74
- 11.7 The While Do and Do While Loops . . . 75
- 11.8 The For Loop . . . 76
- 11.8.1 Zipper Iteration . . . 77
- 11.8.2 Parameter For Loops . . . 77
- 11.9 The Break, Continue and Label Statements . . . 77
- 11.10 The Use Statement . . . 78
- 11.11 The Type Select Statement . . . 79
- 11.12 The Empty Statement . . . 80

12 Modules **83**
- 12.1 Module Definitions . . . 83
- 12.2 Files and Implicit Modules . . . 83
- 12.3 Nested Modules . . . 84
- 12.4 Using Modules . . . 85
- 12.4.1 Explicit Naming . . . 85
- 12.4.2 Module Initialization . . . 86
- 12.5 Program Execution . . . 86
- 12.5.1 The *main* Function . . . 87
- 12.5.2 Module Initialization Order . . . 88

13 Procedures **89**
- 13.1 Function Calls . . . 89
- 13.2 Procedure Definitions . . . 90
- 13.3 Functions without Parentheses . . . 92
- 13.4 Formal Arguments . . . 92

13.4.1 Named Arguments . . . 92
13.4.2 Default Values . . . 93
13.5 Intents . . . 93
13.5.1 The Blank Intent . . . 93
13.5.2 The In Intent . . . 93
13.5.3 The Out Intent . . . 94
13.5.4 The Inout Intent . . . 94
13.6 Variable Number of Arguments . . . 94
13.7 Variable Functions . . . 95
13.8 Parameter Functions . . . 96
13.9 Type Functions . . . 96
13.10 The Return Statement . . . 97
13.11 Return Types . . . 97
13.11.1 Explicit Return Types . . . 97
13.11.2 Implicit Return Types . . . 98
13.12 Nested Functions . . . 98
13.13 Function and Operator Overloading . . . 98
13.14 Function Resolution . . . 98
13.14.1 Determining Visible Functions . . . 99
13.14.2 Determining Candidate Functions . . . 99
Valid Mapping . . . 99
Legal Argument Mapping . . . 99
13.14.3 Determining More Specific Functions . . . 100

14 Tuples **103**
14.1 Tuple Types . . . 103
14.2 Tuple Values . . . 104
14.3 Tuple Assignment . . . 105
14.4 Tuple Destructuring . . . 105
14.4.1 Splitting a Tuple with Assignment . . . 105
14.4.2 Splitting a Tuple in a Declaration . . . 106
14.4.3 Splitting a Tuple into Mutiple Indices of a Loop . . . 107
14.4.4 Splitting a Tuple into Multiple Formal Arguments in a Function Call . . . 108
14.4.5 Splitting a Tuple via Tuple Expansion . . . 109
14.5 Tuple Indexing . . . 109
14.6 Tuple Operators . . . 110
14.6.1 Unary Operators . . . 110
14.6.2 Binary Operators . . . 110
14.6.3 Relational Operators . . . 111
14.7 Predefined Functions and Methods on Tuples . . . 111

15 Classes **113**
15.1 Class Declarations . . . 113
15.1.1 Class Types . . . 114
15.1.2 Class Values . . . 114
15.1.3 Class Fields . . . 115
15.1.4 Class Methods . . . 115
15.1.5 Nested Classes . . . 115
15.2 Inheritance . . . 116
15.2.1 The object Class . . . 116
15.2.2 Accessing Base Class Fields . . . 116

15.2.3 Derived Class Constructors . . . 116
15.2.4 Shadowing Base Class Fields . . . 117
15.2.5 Overriding Base Class Methods . . . 117
15.2.6 Inheriting from Multiple Classes . . . 117
15.2.7 The *nil* Value . . . 117
15.2.8 Default Initialization . . . 118
15.3 Class Constructors . . . 118
15.3.1 User-Defined Constructors . . . 118
15.3.2 The Compiler-Generated Constructor . . . 119
15.4 Field Accesses . . . 120
15.4.1 Variable Getter Methods . . . 120
15.5 Class Method Calls . . . 121
15.5.1 The Method Receiver and the *this* Argument . . . 121
15.6 The *this* Method . . . 122
15.7 The *these* Method . . . 122
15.8 Common Operations . . . 122
15.8.1 Class Assignment . . . 122
15.8.2 Implicit Class Conversions . . . 123
15.9 Dynamic Memory Management . . . 123

16 Records **125**
16.1 Record Declarations . . . 125
16.1.1 Record Types . . . 126
16.1.2 Record Fields . . . 127
16.1.3 Record Methods . . . 127
16.1.4 Nested Record Types . . . 127
16.2 Record Inheritance . . . 127
16.2.1 Shadowing Base Record Fields . . . 128
16.2.2 Overriding Base Record Methods . . . 128
16.3 Record Variable Declarations . . . 128
16.3.1 Storage Allocation . . . 129
16.3.2 Record Initialization . . . 129
16.4 Record Arguments . . . 130
16.5 Record Field Access . . . 130
16.5.1 Field Getter Methods . . . 130
16.6 Record Method Calls . . . 131
16.6.1 The Method Receiver and the *this* Argument . . . 131
16.7 The *this* Method . . . 131
16.8 The *these* Method . . . 131
16.9 Common Operations . . . 131
16.9.1 Record Assignment . . . 131
16.9.2 Assigning Objects to Records . . . 132
16.9.3 Default Comparison Operators . . . 133
16.9.4 Implicit Record Conversions . . . 133
16.10 Differences between Classes and Records . . . 133
16.10.1 Declarations . . . 133
16.10.2 Storage Allocation . . . 133
16.10.3 Assignment . . . 134
16.10.4 Arguments . . . 134
16.10.5 Inheritance . . . 134
16.10.6 Shadowing and Overriding . . . 134

16.10.7 No *nil* Value 135
16.10.8 The *delete* operator 135
16.10.9 Default Comparison Operators 135

17 Unions 137
17.1 Union Types 137
17.2 Union Declarations 137
17.2.1 Union Fields 137
17.3 Union Assignment 138
17.4 The Type Select Statement and Unions 138

18 Ranges 139
18.1 Range Concepts 139
18.2 Range Types 141
18.3 Range Values 142
18.3.1 Range Literals 142
18.3.2 Default Values 143
18.4 Common Operations 143
18.4.1 Range Assignment 143
18.4.2 Range Comparisons 144
18.4.3 Iterating over Ranges 144
Iterating over Unbounded Ranges in Zippered Iterations 144
18.5 Range Operators 144
18.5.1 By Operator 145
18.5.2 Align Operator 146
18.5.3 Count Operator 146
18.5.4 Arithmetic Operators 147
18.5.5 Range Slicing 148
18.6 Predefined Functions on Ranges 149
18.6.1 Range Type Parameters 149
18.6.2 Range Properties 149
18.6.3 Other Queries 151
18.6.4 Range Transformations 151

19 Domains 153
19.1 Domain Overview 153
19.2 Base Domains 154
19.2.1 Rectangular Domains 154
Rectangular Domain Types 154
Rectangular Domain Values 155
19.2.2 Associative Domains 156
Associative Domain Types 156
Associative Domain Values 157
19.3 Simple Subdomains 158
19.3.1 Simple Subdomain Types 158
19.3.2 Simple Subdomain Values 159
19.4 Sparse Subdomains 159
19.4.1 Sparse Subdomain Types 159
19.4.2 Sparse Subdomain Values 159
19.5 Domain Index Types 160
19.6 Iteration Over Domains 160

19.7 Domains as Arguments 160
19.7.1 Formal Arguments of Domain Type 161
19.7.2 Domain Promotion of Scalar Functions 161
19.8 Domain Operations 161
19.8.1 Domain Assignment 161
19.8.2 Domain Striding 162
19.8.3 Domain Slicing 162
Domain-based Slicing 162
Range-based Slicing 163
Rank-Change Slicing 163
19.8.4 Count Operator 163
19.8.5 Adding and Removing Domain Indices 163
19.9 Predefined Methods on Domains 164
19.9.1 Methods on All Domain Types 164
19.9.2 Methods on Regular Domains 165
19.9.3 Methods on Irregular Domains 166

20 Arrays 167
20.1 Array Types 167
20.2 Array Values 167
20.2.1 Runtime Representation of Array Values 168
20.3 Array Indexing 168
20.3.1 Rectangular Array Indexing 168
20.4 Iteration over Arrays 169
20.5 Array Assignment 169
20.6 Array Slicing 170
20.6.1 Rectangular Array Slicing 170
20.6.2 Rectangular Array Slicing with a Rank Change 171
20.7 Count Operator 171
20.8 Array Arguments to Functions 171
20.8.1 Formal Arguments of Rectangular Array Type 171
20.8.2 Array Promotion of Scalar Functions 172
20.9 Array Aliases 172
20.10 Sparse Arrays 173
20.11 Association of Arrays to Domains 173
20.12 Predefined Functions and Methods on Arrays 174

21 Iterators 175
21.1 Iterator Definitions 175
21.2 The Yield Statement 175
21.3 Iterator Calls 176
21.3.1 Iterators in For and Forall Loops 176
21.3.2 Iterators as Arrays 176
21.3.3 Iterators and Generics 176
21.3.4 Recursive Iterators 177
21.4 Parallel Iterators 177

22 Generics **179**
22.1 Generic Functions 179
22.1.1 Formal Type Arguments 179
22.1.2 Formal Parameter Arguments 180
22.1.3 Formal Arguments without Types 180
22.1.4 Formal Arguments with Queried Types 180
22.1.5 Formal Arguments of Generic Type 181
22.1.6 Formal Arguments of Generic Array Types 183
22.2 Function Visibility in Generic Functions 183
22.3 Generic Types 184
22.3.1 Type Aliases in Generic Types 184
22.3.2 Parameters in Generic Types 185
22.3.3 Fields without Types 185
22.3.4 The Type Constructor 186
22.3.5 Generic Methods 186
22.3.6 The Compiler-Generated Constructor 187
22.3.7 User-Defined Constructors 187
22.4 Where Expressions 188
22.5 User-Defined Compiler Diagnostics 188
22.6 Example: A Generic Stack 189

23 Input and Output **191**
23.1 The *file* type 191
23.2 Standard files *stdout*, *stdin*, and *stderr* 192
23.3 The *write*, *writeln*, *read*, and *readln* functions 192
23.4 User-Defined *writeThis* methods 192
23.5 The *write* and *writeln* method on files 193
23.5.1 The *write* and *writeln* method on strings 193
23.5.2 Generalized *write* and *writeln* 193
23.6 The *read* and *readln* methods on files 193
23.7 Default *read* and *write* methods 194

24 Task Parallelism and Synchronization **195**
24.1 Tasks and Task Parallelism 195
24.2 The Begin Statement 195
24.3 Synchronization Variables 196
24.3.1 Predefined Single and Sync Methods 198
24.4 The Cobegin Statement 199
24.5 The Coforall Loop 200
24.6 The Sync Statement 200
24.7 The Serial Statement 201
24.8 Atomic Statements 202

25 Data Parallelism **205**
25.1 The Forall Statement 205
25.1.1 Syntax 205
25.1.2 Execution and Serializability 206
25.1.3 Zipper Iteration 206
25.2 The Forall Expression 206
25.2.1 Syntax 206
25.2.2 Execution and Serializability 207

25.2.3 Zipper Iteration 207
25.2.4 Filtering Predicates in Forall Expressions 207
25.3 Promotion 207
25.3.1 Zipper Promotion 208
25.3.2 Whole Array Assignment 208
25.3.3 Evaluation Order 209
25.4 Reductions and Scans 209
25.4.1 Reduction Expressions 209
25.4.2 Scan Expressions 210
25.5 Configuration Constants for Default Data Parallelism 211

26 Locales 213
26.1 Locales 213
26.1.1 Locale Types 213
26.1.2 Locale Methods 213
26.1.3 The Predefined Locales Array 214
26.1.4 The *here* Locale 214
26.1.5 Querying the Locale of an Expression 215
26.2 The On Statement 216
26.2.1 Remote Variable Declarations 216

27 Domain Maps 217
27.1 Domain Maps for Domain Types 217
27.2 Domain Maps for Domain Values 219
27.3 Domain Maps for Arrays 219
27.4 Domain Maps Are Not Retained upon Domain Assignment 220

28 User-Defined Reductions and Scans 221

29 User-Defined Domain Maps 223

30 Memory Consistency Model 225

31 Interoperability 227
31.1 Interoperability Overview 227
31.1.1 Calling External Functions 227
31.1.2 Calling Chapel Functions 228
31.2 Shared Language Elements 229
31.2.1 Shared Types 229
Referring to Standard C Types 229
Referring to External C Types 230
Referring to External C Structs 230
Referring to External Structs Through Pointers 231
Opaque Types 231
31.2.2 Shared Data 232
31.2.3 Shared Procedures 232
Calling External C Functions 232
31.2.4 Calling Chapel Procedures Externally 233
31.2.5 Argument Passing 234

32 Standard Modules **235**
32.1 Automatic Modules 235
32.1.1 Math 235
32.1.2 Base 240
32.1.3 Types 241
32.2 Optional Modules 241
32.2.1 AdvancedIters 241
32.2.2 BitOps 242
32.2.3 Norm 242
32.2.4 Random 243
32.2.5 Search 244
32.2.6 Sort 244
32.2.7 Time 245
32.2.8 UtilMath 246

33 Standard Distributions **247**
33.1 The Block Distribution 247
33.2 The Cyclic Distribution 248
33.3 The Block-Cyclic Distribution 250
33.4 The Replicated Distribution 251

34 Standard Layouts **253**

A Collected Lexical and Syntax Productions **255**
A.1 Alphabetical Lexical Productions 255
A.2 Alphabetical Syntax Productions 256
A.3 Depth-First Lexical Productions 270
A.4 Depth-First Syntax Productions 272

Index **287**

1 Scope

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

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

2 Notation

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

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

Example.

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

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

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

Example. The production

bool−literal:
  true
  false

defines *bool−literal* to be either the symbol **true** or **false**.

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

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

Example. The production

unary−operator: *one of*
  +\ \ −\ \ ˜\ \ !

is equivalent to

unary−operator:
  +
  −
  ˜
  !

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

> *Example*. The production
>
> *formal*:
> *formal−tag identifier formal−type*$_{opt}$ *default−expression*$_{opt}$
>
> is equivalent to
>
> *formal*:
> *formal−tag identifier formal−type default−expression*
> *formal−tag identifier formal−type*
> *formal−tag identifier default−expression*
> *formal−tag identifier*

3 Organization

This specification is organized as follows:

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

- Chapter 24, Task Parallelism and Synchronization, describes task-parallel expressions and statements in Chapel as well as synchronization constructs and the atomic statement.
- Chapter 25, Data Parallelism, describes data-parallel expressions and statements in Chapel including reductions and scans, whole array assignment, and promotion.
- Chapter 26, Locales, describes constructs for managing locality and executing Chapel programs on distributed-memory systems.
- Chapter 27, Domain Maps, describes Chapel's *domain map* construct for defining the layout of domains and arrays within a single locale and/or the distribution of domains and arrays across multiple locales.
- Chapter 28, User-Defined Reductions and Scans, describes how Chapel programmers can define their own reduction and scan operators.
- Chapter 29, User-Defined Domain Maps, describes how Chapel programmers can define their own domain maps to implement domains and arrays.
- Chapter 30, Memory Consistency Model, describes Chapel's rules for ordering the reads and writes performed by a program's tasks.
- Chapter 31 describes Chapel's interoperability features for combining Chapel programs with code written in different languages.
- Chapter 32, Standard Modules, describes the standard modules that are provided with the Chapel language.
- Chapter 33, Standard Distributions, describes the standard distributions (multi-locale domain maps) that are provided with the Chapel language.
- Chapter 34, Standard Layouts, describes the standard layouts (single locale domain maps) that are provided with the Chapel language.
- Appendix A, Collected Lexical and Syntax Productions, contains the syntax productions listed throughout this specification in both alphabetical and depth-first order.

4 Acknowledgments

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

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

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

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

5 Language Overview

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

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

5.1 Guiding Principles

The following four principles guided the design of Chapel:

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

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

5.1.1 General Parallel Programming

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

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

Such a global view of data contrasts with most parallel languages which tend to require users to partition distributed data aggregates into per-processor chunks either manually or using language abstractions. As a simple example, consider creating a 0-based vector with n elements distributed between p locales. A language like Chapel that supports global-view data structures allows the user to declare the array to contain n elements and to refer to the array using the indices $0 \ldots n-1$. In contrast, most traditional approaches require the user to declare the array as p chunks of n/p elements each and to specify and manage 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. 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 code to more closely match a particular machine's characteristics.

5.1.2 Locality-Aware Programming

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

5.1.3 Object-Oriented Programming

A third principle in Chapel is support for object-oriented programming. Object-oriented programming has been instrumental in raising productivity in the mainstream programming community due to its encapsulation of related data and functions within 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 coding 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 to use Chapel effectively. Many of Chapel's standard library capabilities are implemented using objects, so such programmers may need to utilize a method-invocation style of syntax to use these capabilities. However, using such libraries does not necessitate broader adoption of object-oriented methodologies.

5.1.4 Generic Programming

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

5.2 Getting Started

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

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

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

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

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

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

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

The following output will be printed to the console:

```
hello, world
```

6 Lexical Structure

This section describes the lexical components of Chapel programs. The purpose of lexical analysis is to separate the raw input stream into a sequence of tokens suitable for input to the parser.

6.1 Comments

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

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

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

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

6.2 White Space

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

6.3 Case Sensitivity

Chapel is a case sensitive language.

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

6.4 Tokens

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

6.4.1 Identifiers

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

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

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

legal−identifier−char:
  letter
  digit
  _
  $

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

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

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

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

6.4.2 Keywords

The following identifiers are reserved as keywords:

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

The following identifiers are keywords reserved for future use:

```
lambda
```

6.4.3 Literals

Bool literals are designated by the following syntax:

bool−literal: *one of*
true false

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

integer−literal:
digits
0x *hexadecimal−digits*
0X *hexadecimal−digits*
0b *binary−digits*
0B *binary−digits*

digits:
digit
digit digits

hexadecimal−digits:
hexadecimal−digit
hexadecimal−digit hexadecimal−digits

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

binary−digits:
binary−digit
binary−digit binary−digits

binary−digit: *one of*
0 1

All integer literals have the type `uint`.

> *Rationale*. Why are there no suffixes on integral literals? Suffixes, like those in C, are not necessary. Explicit conversions can then be used to change the type of the literal to another integer size.

Real literals are designated by the following syntax:

real−literal:
$digits_{opt}$. *digits* $exponent\text{−}part_{opt}$
digits $._{opt}$ *exponent−part*

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

> *Rationale*. Why can’t a real literal end with ’.’? There is a lexical ambiquity between real literals ending in ’.’ and the range operator ’..’ that makes it difficult to parse. For example, we want to parse `1..10` as a range from 1 to 10 without concern that `1.` is a real literal.

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

Imaginary literals are designated by the following syntax:

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

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

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

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

String literals are designated by the following syntax:

> *string−literal*:
> ” *double−quote−delimited−characters*$_{opt}$ ”
> ’ *single−quote−delimited−characters*$_{opt}$ ’
>
> *double−quote−delimited−characters*:
> *string−character double−quote−delimited−characters*$_{opt}$
> ’ *double−quote−delimited−characters*$_{opt}$
>
> *single−quote−delimited−characters*:
> *string−character single−quote−delimited−characters*$_{opt}$
> ” *single−quote−delimited−characters*$_{opt}$
>
> *string−character*:
> **any character except the double quote, single quote, or new line**
> *simple−escape−character*
> *hexadecimal−escape−character*
>
> *simple−escape−character*: *one of*
> **\\’ \\” \\? \\a \\b \\f \\n \\r \\t \\v**
>
> *hexadecimal−escape−character*:
> **\\x** *hexadecimal−digits*

6.4.4 Operators and Punctuation

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

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

6.4.5 Grouping Tokens

The following braces are part of the Chapel language:

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

7 Types

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

The syntax of a type is as follows:

type−specifier:
  primitive−type
  enum−type
  locale−type
  structured−type
  dataparallel−type
  synchronization−type

Programmers can define their own enumerated types, classes, records, unions, and type aliases using type declaration statements:

type−declaration−statement:
  enum−declaration−statement
  class−declaration−statement
  record−declaration−statement
  union−declaration−statement
  type−alias−declaration−statement

These statements are defined in Sections §7.2, §15.1, §16.1, §17.2, and §7.7, respectively.

7.1 Primitive Types

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

The primitive types are summarized by the following syntax:

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

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

integer−parameter−expression:
  expression

If present, the parenthesized *integer−parameter−expression* must evaluate to a compile-time constant of integer type. See §8.4.1

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

7.1.1 The Void Type

The `void` type is used to represent the lack of a value, for example when a function has no arguments and/or no return type.

There may be storage associated with a value of type `void`, in which case its lifetime obeys the same rules as a value of type `int`.

7.1.2 The Bool Type

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

Some statements require expressions of `bool` type and Chapel supports a special conversion of values to `bool` type when used in this context (§9.1.3).

7.1.3 Signed and Unsigned Integral Types

The integral types can be parameterized by the number of bits used to represent them. Valid bit-sizes are 8, 16, 32, and 64. The default signed integral type, `int`, and the default unsigned integral type, `uint`, are treated as distinct types (for the purposes of type resolution) but behave like int(64) and uint(64) respectively.

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)`	-2147483648	2147483647
`uint(32)`	0	4294967295
`int(64),int`	-9223372036854775808	9223372036854775807
`uint(64),uint`	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 an implicit conversion to the corresponding 32-bit types according to the rules defined in §9.1.

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

7.1.4 Real Types

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

7.1.5 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)`.

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

7.1.6 Complex Types

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

The real and imaginary components can be accessed via the methods `re` and `im`. The type of these components is real. See §32.1.1 for math routines for complex types.

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

7.1.7 The String Type

Strings are a primitive type designated by the symbol `string` comprised of ASCII characters. Their length is unbounded. See §32.1.2 for routines for manipulating strings.

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

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

7.2 Enumerated Types

Enumerated types are declared with the following syntax:

enum-declaration-statement:
  enum *identifier* { *enum-constant-list* }

enum-constant-list:
  enum-constant
  enum-constant , *enum-constant-list*$_{opt}$

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

init-part:
  = *expression*

The enumerated type can then be referenced by its name, as summarized by the following syntax:

enum-type:
  identifier

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

Example (enum.chpl). The code

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

defines an enumerated type with four constants. The function

```
proc quote(s: statesman) {
  select s {
    when statesman.Aristotle do
       writeln("All paid jobs absorb and degrade the mind.");
    when statesman.Roosevelt do
       writeln("Every reform movement has a lunatic fringe.");
    when statesman.Churchill do
       writeln("A joke is a very serious thing.");
    when statesman.Kissinger do
       { write("No one will ever win the battle of the sexes; ");
         writeln("there's too much fraternizing with the enemy."); }
  }
}
```

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

7.3 Locale Types

Locale types are summarized by the following syntax:

locale−type:
locale

The `locale` type is defined in §26.1.1.

7.4 Structured Types

The structured types are summarized by the following syntax:

structured−type:
class−type
record−type
union−type
tuple−type

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

7.4.1 Class Types

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

7.4.2 Record Types

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

7.4.3 Union Types

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

7.4.4 Tuple Types

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

7.5 Data Parallel Types

The data parallel types are summarized by the following syntax:

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

Ranges and their index types are discussed in §18. Domains and their index types are discussed in §19. Arrays are discussed in §20.

7.5.1 Range Types

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

7.5.2 Domain, Array, and Index Types

A domain defines a set of indices. An array defines a set of elements that correspond to the indices in its domain. A domain's indicies can be of any type. Domains, arrays, and their index types are defined in §19 and §20.

7.6 Synchronisation Types

The synchronization types are summarized by the following syntax:

synchronization-type:
  sync-type
  single-type

Sync and single types are discussed in §24.3.

7.7 Type Aliases

Type aliases are declared with the following syntax:

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

type-alias-declaration-list:
  type-alias-declaration
  type-alias-declaration , *type-alias-declaration-list*

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

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

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

If the keyword `extern` precedes the `type` keyword, the type alias is external. The declared type name is used by Chapel for type resolution, but no type alias is generated by the backend. See the chapter on interoperability (§31) for more information on external types.

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

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

8 Variables

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

8.1 Variable Declarations

Variables are declared with the following syntax:

variable−declaration−statement:
  config−or−extern$_{opt}$ *variable−kind variable−declaration−list* ;

config−or−extern: *one of*
  config extern

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
  array−alias−declaration

type−part:
  : *type−specifier*

initialization−part:
  = *expression*

identifier−list:
  identifier
  identifier , *identifier−list*
  tuple−grouped−identifier−list
  tuple−grouped−identifier−list , *identifier−list*

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

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

The optional keyword `config` specifies that the variables are configuration variables, described in Section §8.5. The optional keyword `extern` indicates that the variable is externally defined. Its name and type are used within the Chapel program for resolution, but no space is allocated for it and no initialization code emitted.

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

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

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

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

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

The *array-alias-declaration* is defined in §20.9.

8.1.1 Default Initialization

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

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

Open issue. In the case that the first enumerator in an enumeration type is offset from zero, as in

Example. enum foo { red = 0xff0000, green = 0xff00, blue = 0xff } ;

the compiler has to look up the first named type to see what to use as the default.

An alternative would be to specify that the default value is the enumerator whose underlying value is zero. But that approach also has issues, since the default value does not conform to any named enumerator.

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

8.1.2 Local Type Inference

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

```
var v = e;
```

is equivalent to

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

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

8.1.3 Multiple Variable Declarations

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

Example (multiple.chpl). In the declaration

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  where the function `readXX` is defined as follows:

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

  Note that the use of the helper function `readXX()` in this code fragment is solely for the purposes of illustration. It is not actually a part of Chapel's semantics or implementation.

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

8.2 Global Variables

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

8.3 Local Variables

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

A local variable only exists during the execution of code that lies within that block statement. This time is called the lifetime of the variable. When execution has finished within that block statement, the local variable and the storage it represents is removed. Variables of class type are the sole exception. Constructors of class types create storage that is not associated with any scope. Such storage can be reclaimed as described in §15.9.

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

8.4.2 Runtime Constants

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

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

8.5 Configuration Variables

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

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

Configuration parameters are set at compilation time via compilation flags or other implementation-defined means. The value passed via these means can be an arbitrary Chapel expression as long as the expression can be evaluated at compile-time. If present, the value thus supplied overrides the default value appearing in the Chapel code.

> *Example (config-param.chpl).* For example,
>
> ```
> config param rank = 2;
> ```
>
> sets a integer parameter `rank` to `2`. At compile-time, `rank` can be set via a configuration file or compile-line override to `3` or `2*n`[1] or indeed to any other expression that can be evaluated at compile-time. The value supplied at compile time overrides the value `2` appearing in the code. In this example, the `rank` configuration variable can be used to write rank-independent code.

---

[1] It is assumed here that `n` is also a param variable.

9 Conversions

A *conversion* converts an expression of one type to another type, possibly changing its value. We refer to these two types the *source* and *target* types. Conversions can be either implicit (§9.1) or explicit (§9.2).

9.1 Implicit Conversions

An *implicit conversion* is a conversion that occurs implicitly, that is, not due to an explicit specification in the program. Implicit conversions occur at the locations in the program listed below. Each location determines the target type. The source and target types of an implicit conversion must be allowed. They determine whether and how the expression's value changes.

An implicit conversion occurs at each of the following program locations:

- In an assignment, the expression on the right-hand side of the assignment is converted to the type of the variable or another lvalue on the left-hand side of the assignment.
- The actual argument of a function call or an operator is converted to the type of the corresponding formal argument, if the formal's intent is `in` or the blank intent with the semantics of `in`.
- The formal argument of a function call is converted to the type of the corresponding actual argument, if the formal's intent is `out`.
- The return or yield expression within a function is converted to the return type of that function.
- The condition of a conditional expression, conditional statement, while-do or do-while loop statement is converted to the boolean type (§9.1.3). A special rule defines the allowed source types and how the expression's value changes in this case.

Implicit conversions are allowed between the following source and target types, as defined in the referenced subsections:

- numeric, boolean, and enumerated types (§9.1.1),
- class types (§15.8.2),
- record types (§16.9.4),
- integral types in the special case when the expression's value is a compile-time constant (§9.1.2), and
- from an integral or class type to `bool` in certain cases (§9.1.3).

In addition, an implicit conversion from a type to the same type is allowed for any type. Such conversion does not change the value of the expression.

Implicit conversion is not transitive. That is, if an implicit conversion is allowed from type `T1` to `T2` and from `T2` to `T3`, that by itself does not allow an implicit conversion from `T1` to `T3`.

9.1.1 Implicit Numeric, Bool and Enumeration Conversions

Implicit conversions among numeric types are allowed when all values representable in the source type can also be represented in the target type, retaining their full precision. In addition, implicit conversions from types `int(64)` and `uint(64)` to types `real(64)` and `complex(128)` are allowed, even though they may result in a loss of precision.

> *Rationale*. We allow these additional conversions because they are an important convenience for application programmers. Therefore we are willing to lose precision in these cases. The largest real and complex types are chosen to retain precision as often as as possible.

Any boolean type can be implicitly converted to any other boolean type, retaining the boolean value. Any boolean type can be implicitly converted to any integral type by representing `false` as 0 and `true` as 1, except (if applicable) a boolean cannot be converted to `int(1)`.

> *Rationale*. We disallow implicit conversion of a boolean to a real, imaginary, or complex type because of the following. We expect that the cases where such a conversion is needed will more likely be unitended by the programmer. Marking those cases as errors will draw the programmer's attention. If such a conversion is actually desired, a cast §9.2 can be inserted.

An expression of an enumerated type can be implicitly converted to an integral type, provided that all of the constants defined by the enumerated type are representable by the integral type.

Legal implicit conversions with numeric, boolean and enumerated types may thus be tabulated as follows:

	Target Type					
Source Type	bool(t)	uint(t)	int(t)	real(t)	imag(t)	complex(t)
bool(s)	all s, t	all s, t	all s; $2 \le t$			
enum		(see rules)	(see rules)			
uint(s)		$s \le t$	$s < t$	$s \le mant(t)$		$s \le mant(t/2)$
uint(64)				real(64)		complex(128)
int(s)			$s \le t$	$s \le mant(t) + 1$		$s \le mant(t/2) + 1$
int(64)				real(64)		complex(128)
real(s)				$s \le t$		$s \le t/2$
imag(s)					$s \le t$	$s \le t/2$
complex(s)						$s \le t$

Here, $mant(i)$ is the number of bits in the (unsigned) mantissa of the i-bit floating-point type.[1] Conversions for the default integral and real types (`uint`, `complex`, etc.) are the same as for their explicitly-sized counterparts.

---

[1] For the IEEE 754 format, $mant(32) = 24$ and $mant(64) = 53$.

9.1.2 Implicit Compile-Time Constant Conversions

The following two implicit conversions of parameters are allowed:

- 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.

9.1.3 Implicit Statement Bool Conversions

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

- An expression of integral type is taken to be false if it is zero and is true otherwise.
- An expression of a class type is taken to be false if it is nil and is true otherwise.

9.2 Explicit Conversions

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

The explicit conversions are a superset of the implicit conversions.

9.2.1 Explicit Numeric Conversions

Explicit conversions are allowed from any numeric type, bool, or string to any other numeric type, bool, or string.

When a `bool` is converted to a `bool`, `int` or `uint` of equal or larger size, its value is zero-extended to fit the new representation. When a `bool` is converted to a smaller `bool`, `int` or `uint`, its most significant bits are truncated (as appropriate) to fit the new representation.

When an `int` is converted to a larger `int` or `uint`, its value is sign-extended to fit the new representation. When a `uint` is converted to a larger `int` or `uint`, its value is zero-extended. When an `int` or `uint` is converted to an `int` or `uint` of the same size, its binary representation is unchanged. When an `int` or `uint` is converted to a smaller `int` or `uint`, its value is truncated to fit the new representation.

> *Future*. There are several kinds of integer conversion which can result in a loss of precision. Currently, the conversions are performed as specified, and no error is reported. In the future, we intend to improve type checking, so the user can be informed of potential precision loss at compile time, and actual precision loss at run time. Such cases include: When an `int` is converted to a `uint` and the original value is negative; When a `uint` is converted to an `int` and the sign bit of the result is true; When an `int` is converted to a smaller `int` or `uint` and any of the truncated bits differs from the original sign bit; When a `uint` is converted to a smaller `int` or `uint` and any of the truncated bits is true;

> *Rationale*. For integer conversions, the default behavior of a program should be to produce a run-time error if there is a loss of precision. Thus, cast expressions not only give rise to a value conversion at run time, but amount to an assertion that the required precision is preserved. Explicit conversion procedures would be available in the run-time library so that one can perform explicit conversions that result in a loss of precision but do not generate a run-time diagnostic.

When converting from a `real` type to a larger `real` type, the represented value is preserved. When converting from a `real` type to a smaller `real` type, the closest representation in the target type is chosen.[2]

When converting to a `real` type from an integer type, integer types smaller than `int` are first converted to `int`. Then, the closest representation of the converted value in the target type is chosen. The exact behavior of this conversion is implementation-defined.

When converting from `real(`k`)` to `complex(`$2k$`)`, the original value is copied into the real part of the result, and the imaginary part of the result is set to zero. When converting from a `real(`k`)` to a `complex(`ℓ`)` such that $\ell > 2k$, the conversion is performed as if the original value is first converted to `real(`$\ell/2$`)` and then to ℓ.

The rules for converting from `imag` to `complex` are the same as for converting from real, except that the imaginary part of the result is set using the input value, and the real part of the result is set to zero.

9.2.2 Explicit Enumeration Conversions

Explicit conversions are allowed from any enumerated type to any integer or real type, `bool`, or `string`, and vice versa.

When the target type is an integer type, the value is first converted to its underlying integer type and then to the target type, following the rules above for converting between integers.

When the target type is a real or complex type, the value is first converted to its underlying integer type and then to the target type.

The conversion of an enumerated type to `imag` is not permitted.

When the target type is `bool`, the value is first converted to its underlying integer type. If the result is zero, the value of the `bool` is `false`; otherwise, it is `true`.

When the target type is `string`, the value becomes the name of the enumerator.

---

[2] When converting to a smaller real type, a loss of precision is *expected*. Therefore, there is no reason to produce a run-time diagnostic.

When the source type is `bool`, enumerators corresponding to the values 0 and 1 in the underlying integer type are selected, corresponding to input values of `false` and `true`, respectively.

When the source type is a real or integer type, the value is converted to the target type's underlying integer type.

The conversion from `complex` and `imag` types to an enumerated type is not permitted.

When the source type is string, the enumerator whose name matches value of the input string is selected. If no such enumerator exists, a runtime error occurs.

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

Chapel provides the following expressions:

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

Individual expressions are defined in the remainder of this chapter and additionally as follows:

- forall, reduce, and scan §25
- module access §12.4.1
- tuple and tuple expand §14
- locale access §26.1.5
- mapped domain §27
- constructor calls §15.3

10.1 Literal Expressions

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

literal−expression:
  bool−literal
  integer−literal
  real−literal
  imaginary−literal
  string−literal
  range−literal
  domain−literal

10.2 Variable Expressions

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

variable−expression:
  identifier

10.3 Enumeration Constant Expression

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

enum−constant−expression:
  enum−type . *identifier*

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

10.4 Parenthesized Expressions

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

parenthesized−expression:
  (*expression*)

Such an expression evaluates to the expression. The parentheses are ignored and have only a syntactical effect.

10.5 Call Expressions

Functions and function calls are defined in §13.

10.6 Indexing Expressions

Indexing, for example into arrays, tuples, and domains, has the same syntax as a call expression.

Indexing is performed by an implicit invocation of the `this` method on the value being indexed, passing the indices as the actual arguments.

10.7 Member Access Expressions

Member access expressions provide access to a field or invoke a method of an instance of a class, record, or union. They are defined in §15.4 and §15.5, respectively.

member−access−expression:
  field−access−expression
  method−call−expression

10.8 The Query Expression

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

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

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

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

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

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

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

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

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

10.9 Casts

A cast is specified with the following syntax:

cast−expression:
  expression : *type−specifier*

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

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

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

10.10 LValue Expressions

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

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

LValue expressions are given by the following syntax:

lvalue−expression:
  variable−expression
  member−access−expression
  call−expression

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

10.11 Precedence and Associativity

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

Operator	Associativity	Use

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

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

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

`!` `~`	right	logical negation bitwise negation

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

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

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

`==` `!=`	left	equal-to comparison not-equal-to comparison
`&`	left	bitwise/logical and
`^`	left	bitwise/logical xor
`|`	left	bitwise/logical or
`&&`	left	short-circuiting logical and
`||`	left	short-circuiting logical or
`..`	left	range construction
`in`	left	forall expression
`by` `#`	left	range/domain stride application range count application

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

`,`	left	comma separated expressions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

10.12 Operator Expressions

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

unary−expression:
  unary−operator expression

unary−operator: *one of*
  + −˜ !

The syntax of a binary expression is given by:

binary−expression:
  expression binary−operator expression

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

The operators are defined in subsequent sections.

10.13 Arithmetic Operators

This section describes the predefined arithmetic operators. These operators can be redefined over different types using operator overloading (§13.13).

For each operator, implicit conversions are applied to the operands of an operator such that they are compatible with one of the function forms listed, those listed earlier in the list being given preference. If no compatible implicit conversions exist, then a compile-time error occurs. In these cases, an explict cast is required.

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

10.13.1 Unary Plus Operators

The unary plus operators are predefined as follows:

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

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

10.13.2 Unary Minus Operators

The unary minus operators are predefined as follows:

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

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

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

10.13.3 Addition Operators

The addition operators are predefined as follows:

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

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

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

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

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

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

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

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

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

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

10.13.4 Subtraction Operators

The subtraction operators are predefined as follows:

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

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

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

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

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

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

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

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

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

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

10.13.5 Multiplication Operators

The multiplication operators are predefined as follows:

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

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

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

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

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

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

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

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

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

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

10.13.6 Division Operators

The division operators are predefined as follows:

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

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

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

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

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

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

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

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

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

When the operands are integers, the result (quotient) is also an integer. If `b` does not divide `a` exactly, then there are two candidate quotients $q1$ and $q2$ such that $b * q1$ and $b * q2$ are the two multiples of `b` closest to `a`. The integer result q is the candidate quotient which lies closest to zero.

10.13.7 Modulus Operators

The modulus operators are predefined as follows:

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

The sign of the result is the same as the sign of the dividend `a`, and the magnitude of the result is always smaller than that of the divisor `b`. For integer operands, the `%` and `/` operators are related by the following identity:

```
var q = a / b;
var r = a % b;
writeln(q * b + r == a);    // true
```

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

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

10.13.8 Exponentiation Operators

The exponentiation operators are predefined as follows:

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

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

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

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

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

10.14 Bitwise Operators

This section describes the predefined bitwise operators. These operators can be redefined over different types using operator overloading (§13.13).

10.14.1 Bitwise Complement Operators

The bitwise complement operators are predefined as follows:

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

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

10.14.2 Bitwise And Operators

The bitwise and operators are predefined as follows:

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

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

Chapel allows mixing signed and unsigned integers of the same size when passing them as arguments to bitwise and. In the mixed case the result is of the same size as the arguments and is unsigned. No run-time error is issued, even if the apparent sign changes as the required conversions are performed.

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

10.14.3 Bitwise Or Operators

The bitwise or operators are predefined as follows:

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

For each of these definitions, the result is computed by applying the logical or operation to the bits of the operands. Chapel allows mixing signed and unsigned integers of the same size when passing them as arguments to bitwise or. No run-time error is issued, even if the apparent sign changes as the required conversions are performed.

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

10.14.4 Bitwise Xor Operators

The bitwise xor operators are predefined as follows:

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

For each of these definitions, the result is computed by applying the XOR operation to the bits of the operands. Chapel allows mixing signed and unsigned integers of the same size when passing them as arguments to bitwise xor. No run-time error is issued, even if the apparent sign changes as the required conversions are performed.

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

10.15 Shift Operators

This section describes the predefined shift operators. These operators can be redefined over different types using operator overloading (§13.13).

The shift operators are predefined as follows:

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

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

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

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

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

10.16 Logical Operators

This section describes the predefined logical operators. These operators can be redefined over different types using operator overloading (§13.13).

10.16.1 The Logical Negation Operator

The logical negation operator is predefined as follows:

```
proc !(a: bool): bool
```

The result is the logical negation of the operand.

10.16.2 The Logical And Operator

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

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

```
a && b
```

is evaluated as the expression

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

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

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

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

10.16.3 The Logical Or Operator

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

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

```
a || b
```

is evaluated as the expression

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

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

10.17 Relational Operators

This section describes the predefined relational operators. These operators can be redefined over different types using operator overloading (§13.13).

10.17.1 Ordered Comparison Operators

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

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

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

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

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

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

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

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

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

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

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

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

The ordered comparison operators are predefined over strings as follows:

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

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

10.17.2 Equality Comparison Operators

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

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

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

The equality comparison operators are predefined over classes as follows:

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

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

Default equality comparison operators are generated for records if the user does not define them. These operators are described in §16.9.3.

The equality comparison operators are predefined over strings as follows:

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

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

10.18 Miscellaneous Operators

This section describes several miscellaneous operators. These operators can be redefined over different types using operator overloading (§13.13).

10.18.1 The String Concatenation Operator

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

> *Example (string-concat.chpl).* The code
>
> ```
> "result: "+i
> ```
>
> where `i` is an integer appends the string representation of `i` to the string literal `"result: "`. If `i` is `3`, then the resulting string would be `"result: 3"`.

10.18.2 The By Operator

The operator `by` is predefined on ranges and rectangular domains. It is described in §18.5.1 for ranges and §19.8.2 for domains.

10.18.3 The Range Count Operator

The operator `#` is predefined on ranges. It is described in §18.5.3.

10.19 Let Expressions

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

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

The scope of the variables is the let-expression.

> *Example (let.chpl).* Let expressions are useful for defining variables in the context of expression. In the code
>
> ```
> let x: real = a*b, y = x*x in 1/y
> ```
>
> the value determined by `a*b` is computed and converted to type real if it is not already a real. The square of the real is then stored in `y` and the result of the expression is the reciprocal of that value.

10.20 Conditional Expressions

A conditional expression is given by the following syntax:

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

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

The 'else' clause can be omitted only when the conditional expression is nested immediately inside a for or forall expression. Such an expression is used to filter predicates as described in §10.21.1 and §25.2.4, respectively.

Example (condexp.chpl). This example shows how if-then-else can be used in a context in which an expression is expected.

```
writehalf(8);
writehalf(21);
writehalf(1000);

proc writehalf(i: int) {
  var half = if (i % 2) then i/2 +1 else i/2;
  writeln("Half of ",i," is ",half);
}
```

Half of 8 is 4 Half of 21 is 11 Half of 1000 is 500

10.21 For Expressions

A for expression is given by the following syntax:

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

The for expression executes a for loop (§11.8), evaluates the body expression on each iteration of the loop, and returns the resulting values as a collection. The size and shape of that collection are determined by the iteratable-expression.

10.21.1 Filtering Predicates in For Expressions

A conditional expression that is immediately enclosed in a for expression does not require an else clause. Such a conditional expression filters the iterations of the for expression. The iterations for which the condition does not hold are not reflected in the result of the for expression.

Example (yieldPredicates.chpl). The code

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

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

11 Statements

Chapel is an imperative language with statements that may have side effects. Statements allow for the sequencing of program execution. Chapel provides the following statements:

statement:
  block−statement
  expression−statement
  assignment−statement
  swap−statement
  conditional−statement
  select−statement
  while−do−statement
  do−while−statement
  for−statement
  label−statement
  break−statement
  continue−statement
  param−for−statement
  use−statement
  type−select−statement
  empty−statement
  return−statement
  yield−statement
  module−declaration−statement
  procedure−declaration−statement
  external−procedure−declaration−statement
  exported−procedure−declaration−statement
  iterator−declaration−statement
  method−declaration−statement
  type−declaration−statement
  variable−declaration−statement
  remote−variable−declaration−statement
  on−statement
  cobegin−statement
  coforall−statement
  begin−statement
  sync−statement
  serial−statement
  atomic−statement
  forall−statement
  delete−statement

Individual statements are defined in the remainder of this chapter and additionally as follows:

- return §13.10
- yield §21.2
- module declaration §12

- procedure declaration §13.2
- iterator declaration §21.1
- method declaration §15.1.4
- type declaration §7
- variable declaration §8.1
- remote variable declaration §26.2.1
- `on` statement §26.2
- cobegin, coforall, begin, sync, serial and atomic statements §24
- forall §25
- delete §15.9

11.1 Blocks

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

block-statement:
  { *statements*$_{opt}$ }

statements:
  statement
  statement statements

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

The statements within a block are executed serially unless the block is in a cobegin statement (§24.4).

11.2 Expression Statements

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

expression-statement:
  expression ;

11.3 Assignment Statements

An assignment statement assigns the value of an expression to another expression, for example, a variable. Assignment statements are given by

assignment-statement:
lvalue-expression assignment-operator expression

assignment-operator: *one of*
= += −= ∗= /= %= ∗∗= &= |= ˆ= &&= ||= <<= >>=

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

When the assignment operator is =, there is an implicit conversion (§9.1) of the right-hand side expression to the type of the left-hand side expression. The left-hand side expression is updated with the result of this conversion.

Additionally, the validity and semantics of assigning between classes (§15.8.1), records (§16.9.1), unions (§17.3), tuples (§14.3), ranges (§18.4.1), domains (§19.8.1), and arrays (§20.5) is discussed in these later sections.

The assignment operators that contain a binary operator symbol as a prefix are *compound assignment* operators. A compound assignment is a short-hand for applying the binary operator to the left and right-hand side expressions and then assigning the result to the left-hand side expression. As with =, the left-hand side expression is evaluated only once, and there is an implicit conversion of the result of the binary operator to the type of the left-hand side expression. Thus, for example, `x += y` is equivalent to `x = x + y` where the expression `x` is evaluated once.

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

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

11.4 The Swap Statement

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

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

swap-operator:
<=>

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

Example. The following swap statement

```
var a, b: real;

a <=> b;
```

is semantically equivalent to:

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

11.5 The Conditional Statement

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

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

else−part:
else *statement*

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

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

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

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

11.6 The Select Statement

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

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

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

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

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

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

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

11.7 The While Do and Do While Loops

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

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

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

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

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

The while-do loop is executed as follows:

1. The expression is evaluated.
2. If the expression evaluates to `false`, the statement is not executed and control continues to the statement following the loop.
3. If the expression evaluates to `true`, the statement is executed and control continues to step 1, evaluating the expression again.

The do-while loop is executed as follows:

1. The statement is executed.

2. The expression is evaluated.
3. If the expression evaluates to `false`, control continues to the statement following the loop.
4. If the expression evaluates to `true`, control continues to step 1 and the the statement is executed again.

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

> *Example (while.chpl).* The following example illustrates the difference between the *do-while-statement* and the *while-do-statement*. The body of the do-while loop is always executed at least once, even if the loop conditional is already false when it is entered.
>
> ```
> var t = 11;
>
> writeln("Scope of do while loop:");
> do {
>   t += 1;
>   writeln(t);
> } while (t <= 10);
>
> t = 11;
> writeln("Scope of while loop:");
> while (t <= 10) {
>   t += 1;
>   writeln(t);
> }
> ```
>
> Scope of do while loop: 12 Scope of while loop:

11.8 The For Loop

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

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

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

iteratable-expression:
  expression

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

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

If the iteratable-expression is a tuple delimited by parentheses, the components of the tuple must support zipper iteration.

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 then statement is executed.

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

Example (zipper.chpl). 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 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−iteratable−expression:
  range−literal
  range−literal **by** *integer−literal*

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

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

11.9 The Break, Continue and Label Statements

The break- and continue-statements are used to alter the flow of control within a loop construct. A break-statement causes flow to exit the containing loop and resume with the statement immediately following it. A continue-statement causes control to jump to the end of the body of the containing loop and resume execution from there. By default, break- and continue-statements exit or skip the body of the immediately-containing loop construct.

The label-statement is used to name a specific loop so that `break` and `continue` can exit or resume a less-nested loop. Labels can only be attached to for-, while-do- and do-while-statements. When a break statement has a label, execution continues with the first statement following the loop statement with the matching label. When a continue statement has a label, execution continues at the end of the body of the loop with the matching label. If there is no containing loop construct with a matching label, a compile-time error occurs.

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

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

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

label−statement:
  label *identifier statement*

Break-statements cannot be used to exit parallel loops.

> *Rationale*. Breaks are not permitted in parallel loops because the execution order of the iterations of parallel loops is not defined.

> *Future*. We expect to support a *eureka* concept which would enable one or more tasks to stop the execution of all current and future iterations of the loop.

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

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

11.10 The Use Statement

The use statement makes the symbols defined by a module available within the scope containing the use statement without requiring them to be prefixed by the module's name. The syntax of the use statement is given by:

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

module−name−list:
  module−name
  module−name , *module−name−list*

module−name:
  identifier
  module−name . *module−name*

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

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

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

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

Example (use.chpl). The following example illustrates how the use statement makes symbols declared in `M1`'s global scope (like procedure `foo()`) visible within the scope of `M2`'s `main` function. Without the `use` statement, the procedure call to `foo` could not be resolved since `M2` would not have access to symbols in `M1`.

```
module M1 {
  proc foo() {
    writeln("In M1's foo.");
  }
}

module M2 {
  proc main() {
    use M1;

    writeln("In M2's main.");
    foo();
  }
}
```

In M2's main.
In M1's foo.

11.11 The Type Select Statement

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

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

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

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

otherwise *statement*

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

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

Let the expressions following the keyword `select` be called 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.

Example (typeselect.chpl). The following example shows how the type select statement can be used to perform a different action based on the static type of the operand.

```
var x = 32, y = 15.5;
var z: int(8);
var coord = (0.0,0.0);
var yes: bool;

writetype(x);
writetype(y);
writetype(z);
writetype(coord);
writetype(yes);
writetype("no");

proc writetype(x) {
  type select x {
    when int do writeln("Integer type");
    when uint do writeln("Unsigned integer type");
    when real do writeln("Real type");
    when complex do writeln("Complex type");
    when string do writeln("String type");
    when bool do writeln("Boolean type");
    otherwise writeln("Non-primitive type");
  }
}
```

Integer type Real type Integer type Non-primitive type Boolean type String type

11.12 The Empty Statement

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

empty−statement:
  ;

12 Modules

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

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

12.1 Module Definitions

A module is declared with the following syntax:

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

module−identifier:
  identifier

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

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

12.2 Files and Implicit Modules

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

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

```
module MX {
  var x: string = "Module MX";
  proc printX() {
    writeln(x);
  }
}

module MY {
  var y: string = "Module MY";
  proc printY() {
    writeln(y);
  }
}
```

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

For any file that contains top-level statements other than module declarations, the file itself is treated as the module declaration. In this case, the module is implicit and takes its name from the base filename. In particular, the module name is defined as the remaining string after removing the `.chpl` suffix and any path specification from the specified filename. If the resulting name is not a legal Chapel identifier, it cannot be referenced in a use statement.

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

```
var x: int = 0;
var y: int = 1;

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

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

12.3 Nested Modules

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

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

Example (nested-use.chpl). The code

```
use libsci.blas;
```

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

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

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

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

module MY {
  var y: int = 0;
}
```

```
use MX, MY;

proc printX() {
  writeln(x);
}

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

12.4 Using Modules

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

12.4.1 Explicit Naming

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

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

module-identifier-list:
  module-identifier
  module-identifier . *module-identifier-list*

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

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

> *Open issue*. It is currently unspecified whether the first-named module is always at the global scope, or whether a scope-search mechanism is used starting at the scope containing the usage.

> *Example (ambiguity.chpl)*. In the following example,

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

```
module M2 {
  use M3;
  use M1;

  var x: int = 2;

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

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

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

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

12.4.2 Module Initialization

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

Example (init.chpl). In the code,

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

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

Module initialization order is discussed in §12.5.2.

12.5 Program Execution

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

12.5.1 The *main* Function

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

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

> *Example (main-module.chpl).* If the following code is compiled without specifying a main module, it will yield an error.
>
> ```
> module M1 {
>   const x = 1;
>   proc main() {
>     writeln("M", x, "'s main");
>   }
> }
>
> module M2 {
>   use M1;
>
>   const x = 2;
>   proc main() {
>     M1.main();
>     writeln("M", x, "'s main");
>   }
> }
> ```
>
> If M1 is specified as the main module, the program will output:
>
> ```
> M1's main
> ```
>
> If M2 is specified as the main module the program will output:
>
> ```
> M1's main
> M2's main
> ```
>
> Notice that main is treated like just another function if it is not in the main module and can be called as such.

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

> *Example (no-main.chpl).* The code
>
> ```
> writeln("hello, world");
> ```
>
> is a legal and complete Chapel program. Its initialization function, which also serves as the program's main function, is the top-level writeln() statement. The module declaration is taken to be the file as described in §12.2.

12.5.2 Module Initialization Order

Module initialization is performed using the following algorithm.

Starting from the module that defines the main function, the modules named in its use statements are visited depth-first and initialized in post-order. If a use statment names a module that has already been visited, it is not visited a second time. Thus, infinite recursion is avoided.

Modules used by a given module are visited in the order in which they appear in the program text. For nested modules, uses in the parent module are initialized before uses in the nested module.

Example (init-order.chpl). The code

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

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

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

prints the following

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

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

13 Procedures

A *function* is a code abstraction that can be invoked by a call expression. Throughout this specification the term “function” is used in this programming-languages sense, rather than in the mathematical sense. A function has zero or more *formal arguments*, or simply *formals*. Upon a function call each formal is associated with the corresponding *actual argument*, or simply *actual*. Actual arguments are provided as part of the call expression, or at the the *call site*. Direct and indirect recursion is supported.

A function can be a *procedure*, which completes and returns to the call site exactly once, returning no result, a single result, or multiple results aggregated in a tuple. A function can also be an iterator, which can generate, or *yield*, multiple results (in sequence and/or in parallel). A function (either a procedure or an iterator) can be a method if it is bound to a type (often a class). An *operator* in this chapter is a procedure with a special name, which can be invoked using infix notation, i.e., via a unary or binary expression. This chapter defines procedures, but most of its contents apply to iterators and methods as well.

Functions are presented as follows:

- procedures (this chapter)
- operators §13.2, §10.12
- iterators §21
- methods (when bound to a class) §15.1.4
- function calls §13.1
- various aspects of defining a procedure §13.2–§13.12
- determining the function to invoke for a given call site: function and operator overloading §13.13, function resolution §13.14

13.1 Function Calls

The syntax to call a non-method function is given by:

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

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

named-expression:
  expression
  identifier = expression
```

parenthesesless-function-identifier:
identifier

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

Functions can be called using either parentheses or brackets. The only difference in the call has to do with promotion and is discussed in §25.3.

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

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

Calls to methods are defined in Section §15.5.

13.2 Procedure Definitions

Procedures are defined with the following syntax:

procedure-declaration-statement:
linkage-specifier$_{opt}$ **proc** *function-name argument-list*$_{opt}$ *return-intent*$_{opt}$ *return-type*$_{opt}$ *where-clause*$_{opt}$
function-body

linkage-specifier:
inline
export *identifier*$_{opt}$
extern

function-name:
identifier
operator-name

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

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

formals:
formal
formal , *formals*

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

formal-type:

: *type-specifier*
: ? *identifier*$_{opt}$

default-expression:
= *expression*

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

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

return-intent: *one of*
var const param type

return-type:
: *type-specifier*

where-clause:
where *expression*

function-body:
block-statement
return-statement

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

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

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

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

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

The *return-intent* can be used to indicate a variable function §13.7, or a parameter function §13.8, or a type function §13.9. By default, or if the *return-intent* is `const`, the function returns a value that cannot be used as an lvalue.

> *Open issue*. Parameter and type procedures are supported. Parameter and type iterators are currently not supported.

The *return-type* is optional and is discussed in §13.11. A type function may not specify a return type.

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

Function and operator overloading is supported in Chapel and is discussed in §13.13. Operator overloading is supported on the operators listed above (see *operator-name*).

Linkage specifiers are optional and mutually exclusive. The linkage specifier `inline` indicates that the function body should be inlined at every call site for improved performance. The linkage specifier `export` indicates that this function may be called from outside the current Chapel program. The linkage specifier `extern` indicates that this function is defined outside the Chapel program. See the chapter on interoperability (§31) for details on exported and imported functions.

13.3 Functions without Parentheses

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

Example (function-no-parens.chpl). Given the definitions

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

the procedure `foo` can be called by writing `foo` and the procedure `bar` can be called by writing `bar()`. It is an error to use parentheses when calling `foo` or omit them when calling `bar`.

13.4 Formal Arguments

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

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

13.4.1 Named Arguments

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

Example (named-args.chpl). In the code

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

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

2 3 2 3

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

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

13.4.2 Default Values

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

Example (default-values.chpl). In the code

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

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

```
5 7 7 7 5 5
```

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

13.5 Intents

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

13.5.1 The Blank Intent

If the intent is omitted, it is called a blank intent. In such a case, the value is copied from the actual to the formal. Thus class instances are passed by reference and records are passed by value. Furthermore, a formal that has blank intent cannot be assigned within the function.

The exception to this rule is made for formals of array, domain, `sync`, and `single` types. Arguments of these types are passed by reference and can be modified 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 by value. The formal 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. Changes to the elements within the array formal are thus not reflected back at the call site.

13.5.3 The Out Intent

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

The formal argument cannot not be generic and is treated as a variable declaration. It is initialized to the default value for its type, unless it has a default expression.

13.5.4 The Inout Intent

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

13.6 Variable Number of Arguments

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

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

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

Example (varargs.chpl). The code

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

defines a generic procedure called `mywriteln` that takes a variable number of arguments of any type and then writes them out on separate lines. The parameter for-loop (§11.8.2) is unrolled by the compiler so that `i` is a parameter, rather than a variable. This needs to be a parameter for-loop because the expression `x(i)` will have a different type on each iteration. The type of `x` can be specified in the formal argument list to ensure that the actuals all have the same type.

Example (varargs-with-type.chpl). Either or both the number of variable arguments and their types can be specified. For example, a basic procedure to sum the values of three integers can be wrtten as

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

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

Example (varargs-returns-tuples.chpl). The code

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

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

13.7 Variable Functions

A variable function is a function whose call is an lvalue (specifically, a call expression for a variable procedure and an iterator variable for an variable iterator). This implies that a variable function produces a reference; however, this reference cannot be captured.

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

When a variable procedure 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 procedure is assigned a value.

Variable functions provide an implicit param formal argument `setter` of type `bool`. If the variable function is called in a context requiring an lvalue, the actual argument for `setter` is implicitly `true`; otherwise it is `false`. This argument is useful for adjusting the behavior depending to the calling context.

Example (variable-functions.chpl). The following code defines a procedure that can be interpreted as a simple two-element array where the elements are actually global variables:

```
var x, y = 0;

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

Calls to this procedure can be assigned to in order to write to the “elements” of the array as in

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

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

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

This code outputs the number `3`.

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

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

13.8 Parameter Functions

A *parameter function*, or a *param 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 substituted for the call expression.

Example (param-functions.chpl). In the code

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

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

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

Parameter functions may not contain control flow that is not resolved at compile-time. This includes loops other than the parameter for loop §11.8.2 and conditionals with a conditional expressions that is not a parameter.

13.9 Type Functions

A *type function* is a function that returns a type, not a value. It is specified by following the function's argument list by the keyword `type`, without the subsequent return type. It is often, but not necessarily, generic.

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

As with parameter functions, type functions may not contain control flow that is not resolved at compile-time. This includes loops other than the parameter for loop §11.8.2 and conditionals with a conditional expression that is not a parameter.

Example (type-functions.chpl). In the code

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

`myType` is a type procedure that takes a single argument `x` and returns `int` if the number of bits used to represent `x` is less than or equal to 32, otherwise it returns `int(64)`. `numBits` is a param procedure defined in the Types module (§32.1.3).

13.10 The Return Statement

The return statement can only appear in a function. It exits that function, returning control to the point at which that function was called. A return statement in a procedure of a non-void return type must include an expression, which indicates the value to be returned, i.e., the value that becomes the value of the invoking call expression. Other returns statements may not include an expression.

The syntax of the return statement is given by

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

Example (return.chpl). The following code defines a procedure that returns the sum of three integers:

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

13.11 Return Types

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

13.11.1 Explicit Return Types

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

13.11.2 Implicit Return Types

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

13.12 Nested Functions

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

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

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

13.13 Function and Operator 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 function resolution algorithm in §13.14.

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

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

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

13.14 Function Resolution

Function resolution is the algorithm that determines which function to invoke for a given call expression. Function resolution is defined as follows.

- Identify the set of visible functions for the function call. A *visible function* is any function that satisfies the criteria in §13.14.1. If no visible function can be found, the compiler will issue an error stating that the call cannot be resolved.

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

13.14.1 Determining Visible Functions

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

13.14.2 Determining Candidate Functions

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

Valid Mapping

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

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

Legal Argument Mapping

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

- T_A and T_F are the same type.

- There is an implicit conversion from T_A to T_F.
- T_A is derived from T_F.
- T_A is scalar promotable to T_F.

13.14.3 Determining More Specific Functions

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

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

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

- If T_{F1} and T_{F2} are the same type, $F1$ is an instantiated parameter, and $F2$ is not an instantiated parameter, M_1 is more specific.
- If T_{F1} and T_{F2} are the same type, $F2$ is an instantiated parameter, and $F1$ is not an instantiated parameter, M_2 is more specific.
- If M_1 does not require scalar promotion and M_2 requires scalar promotion, M_1 is more specific.
- If M_1 requires scalar promotion and M_2 does not require scalar promotion, M_2 is more specific.
- If T_{F1} and T_{F2} are the same type, $F1$ is generic, and $F2$ is not generic, M_1 is more specific.
- If T_{F1} and T_{F2} are the same type, $F2$ is generic, and $F1$ is not generic, M_2 is more specific.
- If $F1$ is not generic over all types and $F2$ is generic over all types, M_1 is more specific.

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

14 Tuples

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

14.1 Tuple Types

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

The syntax of a tuple type is given by:

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

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

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

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

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

> *Example (homogenous.chpl).* The statement
>
> ```
> var x1: (string, real),
>     x2: (int, int, int),
>     x3: 3*int;
> ```
>
> defines three variables. Variable `x1` is a 2-tuple with component types `string` and `real`. Variables `x2` and `x3` are homogeneous 3-tuples with component type `int`. The types of `x2` and `x3` are identical even though they are specified in different ways.

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

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

14.2 Tuple Values

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

The syntax of a tuple expression is given by:

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

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

> *Example (values.chpl).* The statement
>
> ```
> var x1: (string, real) = ("hello", 3.14),
>     x2: (int, int, int) = (1, 2, 3),
>     x3: 3*int = (4, 5, 6);
> ```
>
> defines three variables. Variable `x1` is a 2-tuple with component types `string` and `real`. It is initialized such that the first component is `"hello"` and the second component is `3.14`. Variables `x2` and `x3` are homogeneous 3-tuples with component type `int`. Their initialization expressions specify 3-tuples of integers.

The function

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

is defined in the standard context to create arbitrary tuples.

> *Example (tuple-function.chpl).* The statement
>
> ```
> var x1 =      ("hello", 3.14),
>     x2 = tuple("hello", 3.14),
>     x3 = tuple(1);
> ```
>
> creates two identical tuples: `x1` and `x2`; and `x3`: an integer 1-tuple initialized to `(1)`.

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

Tuple expressions are evaluated similarly to function calls where the arguments are all generic with no explicit intent. So a tuple expression containing an array does not copy the array.

When a tuple is passed as an argument to a function, it is passed as if it is a record type containing fields of the same type and in the same order as in the tuple.

14.3 Tuple Assignment

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

14.4 Tuple Destructuring

Tuples can be split into their components in the following ways:

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

14.4.1 Splitting a Tuple with Assignment

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

Example (splitting.chpl). The code

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

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

Example (aliasing.chpl). The code

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

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

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

Example (omit-component.chpl). The code

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

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

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

14.4.2 Splitting a Tuple in a Declaration

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

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

Example (decl.chpl). The code

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

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

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

Example (one-tuple.chpl). The code

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

initialize the new variable `a` to 1.

When splitting a tuple into multiple variable declarations, variable names in the declarator on the left-hand side may be omitted. In this case, no variables are defined for the omitted components.

Example (omit-component-decl.chpl). The code

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

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

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

14.4.3 Splitting a Tuple into Mutiple Indices of a Loop

When multiple indices in a loop are grouped using tuple notation, the tuple returned by the iterator (§21) is split across the index tuple's components. The number of indices in the index tuple must equal the size of the tuple returned by the iterator.

Example (indices.chpl). The code

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

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

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

When a tuple is split across an index tuple, indices in the index tuple (left-hand side) may be omitted. In this case, no indices are defined for the omitted components.

However even when indices are omitted, the iterator is evaluated as if an index were defined. Execution proceeds as if the omitted indices are present but invisible. This means that the loop body controlled by the iterator may be executed multiple times with the same set of (visible) indices.

14.4.4 Splitting a Tuple into Multiple Formal Arguments in a Function Call

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

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

> *Example (formals.chpl).* The function
>
> ```
> proc f(x: int, (y, z): (int, int)) {
>   // body
> }
> ```
>
> is defined to take an integer value and a 2-tuple of integer values. The 2-tuple is split when the function is called into two formals. A call may look like the following:
>
> ```
> f(1, (2, 3));
> ```

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

> *Example (implicit-where.chpl).* The function `f` defined as
>
> ```
> proc f((x, (y, z))) {
>   writeln((x, y, z));
> }
> ```
>
> is equivalent to the function `g` defined as
>
> ```
> proc g(t) where isTuple(t) && t.size == 2 && isTuple(t(2)) && t(2).size == 2 {
>   writeln((t(1), t(2)(1), t(2)(2)));
> }
> ```
>
> except without the definition of the argument name `t`.

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

> *Example (grouping-Formals.chpl).* The empty function
>
> ```
> proc f((x)) { }
> ```
>
> accepts a 1-tuple actual with any component type.

When splitting a tuple into multiple formal arguments, the arguments that are grouped using the tuple notation may be omitted. In this case, no names are associated with the omitted components. The call is evaluated as if an argument were defined.

14.4.5 Splitting a Tuple via Tuple Expansion

Tuples can be expanded in place using the following syntax:

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

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

Example (expansion.chpl). Given two 2-tuples

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

the following statement

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

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

Example (expansion-2.chpl). The following code defines two functions, a function `first` that returns the first component of a tuple and a function `rest` that returns a tuple containing all of the components of a tuple except for the first:

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

14.5 Tuple Indexing

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

Example (access.chpl). The loop

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

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

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

Example (access-homogeneous.chpl). The loop

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

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

Rationale. Non-homogeneous tuples can only be accessed by compile-time constants since the type of an expression must be statically known.

14.6 Tuple Operators

14.6.1 Unary Operators

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

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

The type of every element of the operand tuple must have a well-defined operator matching the unary operator being applied. That is, if the element type is a user-defined type, it must supply an overloaded definition for the unary operator being used. Otherwise, a compile-time error will be issued.

14.6.2 Binary Operators

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

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

When a tuple binary operator is used, the same operator must be well-defined for successive pairs of operands in the two tuples. Otherwise, the operation is illegal and a compile-time error will result.

Example (binary-ops.chpl). The code

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

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

14.6.3 Relational Operators

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

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

However, just as for other binary tuple operators, the corresponding operation must be well-defined on each successive pair of operand types in the two operand tuples. Otherwise, a compile-time error will result.

> *Example (relational-ops.chpl).* The code
>
> ```
> var x = (1, 1, 0) > (1, 0, 1);
> ```
>
> creates a variable initialized to `true`. After comparing the first components and determining they are equal, the second components are compared to determine that the first tuple is greater than the second tuple.

14.7 Predefined Functions and Methods on Tuples

proc *Tuple*`.size` **param**

Returns the size of the tuple.

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

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

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

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

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

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

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

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

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

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

15 Classes

Classes are data structures with associated state and functions. Storage for a class instance, or object, is allocated independently of the scope of the variable that refers to it. An object is created by calling a class constructor (§15.3), which allocates storage, initializes it, and returns a reference to the newly-created object. Storage can be reclaimed by deleting the object (§15.9).

A class declaration (§15.1) generates a class type (§15.1.1). A variable of a class type can refer to an instance of that class or any of its derived classes.

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

15.1 Class Declarations

A class is defined with the following syntax:

class-declaration-statement:
  external-class-declaration-statement
  simple-class-declaration-statement

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

class-inherit-list:
  : *class-type-list*

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

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

class-statement:
  variable-declaration-statement
  method-declaration-statement
  type-declaration-statement
  empty-statement

A *class-declaration-statement* defines a new type symbol specified by the identifier. Classes inherit data and functionality from other classes if the *inherit-type-list* is specified. Inheritance is described in §15.2.

> *Future*. Record inheritance is not currently implemented. When enabled, it will have the same effect as creating a field of that record type within the class, except that all of the record's field and method names will appear within the scope of the inheriting class.

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 procedure or iterator (called a method), or a type alias. In addition, empty statements are allowed in class declarations, and they have no effect.

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

If the `extern` keyword appears before the `class` keyword, then an external class type is declared. An external class type declaration must not contain a *class-inherit-list*. An external class is used within Chapel for type and field resolution, but no corresponding backend definition is generated. It is presumed that the definition of an external class is supplied by a library or the execution environment. See the chapter on interoperability (§31) for more information on external classes.

15.1.1 Class Types

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

```
class-type:
  identifier
  identifier ( named-expression-list )
```

A class type may appear in inheritance lists of other class declarations.

15.1.2 Class Values

A class value is either a reference to an instance of a class or `nil` (§15.2.7). Class instances can be created using the `new` operator (§15.3) and deleted using the `delete` operator (§15.9).

For a given class type, a legal value of that type is a reference to an instance of either that class or a class inheriting, directly or indirectly, from that class. `nil` is a legal value of any class type.

The default value of a class type is `nil`.

Example (declaration.chpl).

```
class C { }
var c : C;       // c has the class type C, initialized with the value nil.
c = new C();     // Now c refers to an object of type C.
var c2 = new C();       // The type of c2 is also C.
class D : C {}   // Class D is derived from C.
c = new D();     // Now c refers to an object of type D.
```

When the variable `c` is declared, it initially has the value of `nil`. The next statement assigned to it an instance of the class `C`. The declaration of variable `c2` shows that these steps can be combined. The type of `c2` is also `C`, determined implicitly from the the initialization expression. Finally, an object of type `D` is created and assigned to `c`. The object previously referenced by `c` is no longer referenced anywhere. It could be reclaimed by the garbage collector.

15.1.3 Class Fields

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

> *Example (defineActor.chpl).* 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`.

Field access is described in §15.4.

15.1.4 Class Methods

A class method is a procedure or iterator that is bound to a class.

Methods are declared with the following syntax:

method-declaration-statement:
  linkage-specifier$_{opt}$ *proc-or-iter param-clause*$_{opt}$ *type-binding function-name argument-list*$_{opt}$
    return-intent$_{opt}$ *return-type*$_{opt}$ *where-clause*$_{opt}$ *function-body*

proc-or-iter:
  proc
  iter

param-clause:
  param

type-binding:
  identifier .

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

Method calls are described in §15.5.

15.1.5 Nested Classes

A class or record defined within another class is a nested class (or record).

Nested classes or records can refer to fields and methods in the outer class (or record) implicitly, or explicitly by means of an `outer` reference.

A nested class (or record) can be referenced only within its immediately enclosing class (or record).

15.2 Inheritance

A *derived* class can inherit from one or more other classes by listing those classes in the derived class declaration. 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.

These restrictions on inheritance induce a class hierarchy which has the form of a tree. A variable referring to an instance of class `C` can be cast to any type that is an ancestor of `C`. Note that casts to more- and less-derived classes are both permitted.

> *Future*. A derived class may also incorporate any number of records by listing them in the derived class declaration. As with record inheritance, this has the effect of injecting the record's fields and methods into the new class type. Record inheritance does not induce a well-defined class hierarchy. See §16.2 for details.

15.2.1 The object Class

All classes are derived from the `object` class, either directly or indirectly. If no class name appears in the inheritance list, the class derives implicitly from `object`. Otherwise, a class derives from `object` indirectly through the class or classes it inherits. A variable of type `object` can hold a reference to an object of any class type.

15.2.2 Accessing Base Class Fields

A derived class contains data associated with the fields in its base classes. The fields can be accessed in the same way that they are accessed in their base class unless the getter or setter method is overridden in the derived class, as discussed in §15.2.5.

15.2.3 Derived Class Constructors

The default initializer of a derived class automatically calls the default initializer of each of its base classes The same is not true for constructors: To initialize inherited fields to anything other than its default-initialized value, a constructor defined in a derived class must either call base class constructors or manipulate those base-class fields directly.

> *Open issue*. The syntax for calling a base-class constructor from a derived-class constructor has not yet been defined.
>
> There is an expectation that a more standard way of chaining constructor calls will be supported.

15.2.4 Shadowing Base Class Fields

A field in the derived class can be declared with the same name as a field in the base class. Such a field shadows the field in the base class in that it is always referenced when it is accessed in the context of the derived class.

> *Open issue*. There is an expectation that there will be a way to reference the field in the base class but this is not defined at this time.

15.2.5 Overriding Base Class Methods

If a method in a derived class is declared with a signature identical to that of 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 an object that has the derived class type.

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

Methods without parentheses are not candidates for dynamic dispatch.

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

15.2.6 Inheriting from Multiple Classes

A class can be derived from multiple base classes provided that only one of the base classes contains fields either directly or from base classes that it is derived from. The methods defined by the other base classes can be overridden. This provides functionality similar to the C# concept of interfaces.

> *Open issue*. It is an open question whether the language will support `interface` declarations and multiple inheritance. This is currently under study at the University of Colorado (Boulder).

15.2.7 The *nil* Value

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

15.2.8 Default Initialization

When an instance of a class (an object) is created, it is brought to a known and legal state first, before it can be accessed or operated upon. This is done through default initialization.

An object is default-initialized by initializing all of its fields in the order of the field declarations within the class. Fields inherited from a superclass are initialized before fields declared in current class.

If a field in the class is declared with an initialization expression, that expression is used to initialize the field. Otherwise, the field is initialized to the default value of its type (§8.1.1).

15.3 Class Constructors

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

When the constructor is called, memory is allocated to store a class instance, the instance undergoes default initialization, and then the constructor method is invoked on this newly-created instance.

If the program declares a class constructor method, it is a user-defined constructor. If the program declares no constructors for a class, a compiler-generated constructor for that class is created automatically.

15.3.1 User-Defined Constructors

A user-defined constructor is a constructor method explicitly declared in the program. A constructor declaration has the same syntax as a method declaration, except that the name of the function matches the name of the class, and there is no return type specifier.

A constructor for a given class is called by placing the `new` operator in front of the class name. Any constructor arguments follow the class name in a parenthesized list.

constructor-call-expression:
  new *class-name* (*argument-list*)

class-name:
  identifier

When a constructor is called, the usual function resolution mechanism (§13.14) is applied to determine which user-defined constructor to invoke.

> *Example (simpleConstructors.chpl).* The following example shows a class with two constructors:

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

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

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

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

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

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

15.3.2 The Compiler-Generated Constructor

A compiler-generated constructor for a class is created automatically if there are no constructors for that class in the program. The compiler-generated constructor has one argument for every field in the class, each of which has a default value equal to the field's initializer (if present) or default value of the field's type (if not). The list of fields (and hence arguments) includes fields inherited from superclasses, type aliases and parameter fields, if any. The order of the arguments in the argument list matches the order of the field declarations within the class, with the arguments for a superclass's fields occurring before the arguments for the fields declared in current class.

Generic fields are treated separately which are discussed in Section §22.3.6.

When invoked, the compiler-generated constructor initializes each field in the class to the value of the corresponding actual argument.

Example (defaultConstructor.chpl). Given the class

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

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

`C` instances can be created by calling the compiler-generated constructor as follows:

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

15.4 Field Accesses

The field in a class is accessed via a field access expression.

field-access-expression:
  receiver-clause$_{opt}$ *identifier*

receiver-clause:
  expression .

The receiver-clause specifies the *receiver*, which is the class instance whose field is being accessed. The receiver clause can be omitted when the field access is within a method. In this case the receiver is the method's receiver §15.5.1. The receiver clause can also be omitted when the field access is within a class declaration. In this case the receiver is the instance being implicitly defined or referenced.

The identifier in the field access expression indicates which field is accessed.

A field can be modified via an assignment statement where the left-hand side of the assignment is a field access expression. Accessing a parameter field returns a parameter.

> *Example (useActor1.chpl).* Given a variable `anActor` of type `Actor` as 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 field `age` in the object `anActor` the value `27`.

15.4.1 Variable Getter Methods

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

> *Example (getterSetter.chpl).* In the code
>
> ```
> class C {
>   var setCount: int;
>   var x: int;
>   proc x var {
>     if setter then
>       setCount += 1;
>     return x;
>   }
> }
> ```
>
> an explicit variable getter method is defined for field `x`. It returns the field `x` and increments another field that records the number of times x was assigned a value.

15.5 Class Method Calls

A method is invoked with a method call, which is similar to a non-method call expression.

method-call-expression:
  receiver-clause$_{opt}$ *expression* (*named-expression-list*)
  receiver-clause$_{opt}$ *expression* [*named-expression-list*]
  receiver-clause$_{opt}$ *parenthesesless-function-identifier*

The receiver-clause (or its absence) specifies the method's receiver §15.5.1 in the same way it does for field accesses §15.4.

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

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

This method can be called on an instance of the `Actor` class, `anActor`, with the call expression `anActor.print()`.

The actual arguments supplied in the method call are bound to the formal arguments in the method declaration following the rules specified for procedures (§13). The exception is the receiver §15.5.1.

15.5.1 The Method Receiver and the *this* Argument

A method's *receiver* is its implicit argument, the class instance on which the method is invoked. Its type is the class in which the method is defined. The receiver actual argument is specified by the receiver-clause of a method-call-expression as specified in §15.4. The receiver formal argument can be referred to within the method using the identifier `this`.

Example (implicitThis.chpl). Let class `C`, method `foo`, and function `bar` be defined as

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

Then given an instance of `C` called `c1`, the method call `c1.foo()` results in a call to `bar` where the argument is `c1`.

15.6 The *this* Method

A procedure method declared with the name `this` allows a class to be "indexed" similarly to how an array is indexed. Indexing into a class instance has the semantics of calling a method 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 (thisMethod.chpl).* 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;
  proc this(i: int) var {
    select i {
      when 1 do return x1;
      when 2 do return x2;
      when 3 do return x3;
    }
    halt("ThreeArray index out of bounds: ", i);
  }
}
```

15.7 The *these* Method

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

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

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

15.8 Common Operations

15.8.1 Class Assignment

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

15.8.2 Implicit Class Conversions

An implicit conversion from class type `D` to another class type `C` is allowed when `D` is a subclass of `C`. Such conversion does not change the value.

15.9 Dynamic Memory Management

Memory associated with class instances can be reclaimed with the `delete` statement:

delete-statement:
  delete *expression* ;

where the expression is a reference to the instance that will be reclaimed.

> *Example (delete.chpl).* The following example allocates a new object `c` of class type `C` and then deletes it.
>
> ```
> var c = new C();
> delete c;
> ```

> *Open issue.* Chapel was originally specified without a `delete` keyword. The intention was that Chapel would be implemented with a distributed-memory garbage collector. This is a research challenge. In order to focus elsewhere, the design has been scaled back. There is an expectation that Chapel will eventually support an optional distributed-memory garbage collector as well as a region-based memory management scheme similar to that used in the Titanium language. Support of `delete` will likely continue even as these optional features become supported.

16 Records

A record is a data structure that is similar to a class but instead has value semantics, similar to primitive types. Value semantics mean that assignment, argument passing and function return values are by default all done by copying. Value semantics also imply that a variable of record type is associated with only one piece of storage and has only one type throughout its lifetime. Storage is allocated for a variable of record type when the variable declaration is executed, and the record variable is also initialized at that time.

A record declaration creates a record type §16.1. A variable of record type contains all and only the fields defined by that type (§16.1.1). Value semantics imply that the type of a record variable is known at compile time (i.e. it is statically typed).

Records are assigned by value; see §16.9.1 for more details. Records are also passed by value unless argument intents (§13.5) are used.

Records can be created through an explicit call to a record constructor, which allocates storage, initializes it and returns it. A record is also created upon a variable declaration of a record type.

A record type is generic if it contains generic fields. Generic record types are discussed in detail in §22.3.

16.1 Record Declarations

A record is defined with the following syntax:

record-declaration-statement:
  simple-record-declaration-statement
  external-record-declaration-statement

simple-record-declaration-statement:
  record *identifier record-inherit-list*$_{opt}$ { *record-statement-list* }

record-inherit-list:
  : *record-type-list*

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

record-statement-list:
  record-statement
  record-statement record-statement-list

record-statement:
  type-declaration-statement
  method-declaration-statement
  variable-declaration-statement
  empty-statement

A *record-declaration-statement* defines a new type symbol specified by the identifier. A record inherits data and methods from other records if the *record-inherit-list* is specified.

> *Future*. Allowing a record to inherit from more than one record is future work.

> *Rationale*. We do not allow records to inherit from classes because of the following.
>
> Inheritance implies that the derived type can be cast to one of its base types. If the base type is a record type, casting to the base type has the effect of removing all of the data fields and all of functions that are not associated with the base type. Thereafter, the record variable has the base record type, in both compile-time and run-time interpretations.
>
> If the base type were a class type, the result of the cast would have the static type of the base class while its run-time type was a record type. Since a record's type is supposed to be determined at compile time, this is a bit incongruous with the definition of a record. Moreover, space would have to be allocated in this special case, to store the record's run-time type.

As in a class declarations, the body of a record declaration can contain variable, iterator and method declarations as well as nested type declarations.

If a record declaration contains a type alias or parameter field, or it contains a variable or constant without a specified type and without an initialization expression, then it declares a generic record type. Generic record types are described in §22.3.

If the `extern` keyword appears before the `record` keyword, then an external record type is declared. An external record type declaration must not contain a *record-inherit-list*. An external record is used within Chapel for type and field resolution, but no corresponding backend definition is generated. It is presumed that the definition of an external record is supplied by a library or the execution environment. See the chapter on interoperability (§31) for more information on external records.

16.1.1 Record Types

A record type specifier simply names a record type, using the following syntax:

> *record-type*:
>   *identifier*
>   *identifier* (*named-expression-list*)

A record type specifier may appear anywhere a type specifier is permitted.

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

16.1.2 Record Fields

Variable declarations within a record type declaration define fields within that record type. The presence of at least one parameter field causes the record type to become generic. Variable fields define the storage associated with a record.

> *Example (defineActorRecord.chpl).* The code
>
> ```
> record ActorRecord {
>   var name: string;
>   var age: uint;
> }
> ```
>
> defines a new record type called `ActorRecord` that has two fields: the string field `name` and the unsigned integer field `age`. The data contained by a record of this type is exactly the same as that contained by an instance of the `Actor` class defined in the preceding chapter §15.1.3.

16.1.3 Record Methods

A record method is a function or iterator that is bound to a record. Unlike functions that take a record as an argument, record methods access the record by reference, so that persistent field updates are possible.

The syntax for record method declarations is identical to that for class method declarations (§15.1.4).

16.1.4 Nested Record Types

Record type declarations may be nested within other class, record and union declarations. Methods defined in a nested record type may access fields declared in the containing aggregate type either implicitly, or explicitly by means of an `outer` reference.

16.2 Record Inheritance

A *derived* record type is a type that inherits from other record types. For each named base record type, inheritance effectively inserts all of its fields and methods into the new record type.

Since record types are resolved statically, there is no type hierarchy implied by record inheritance. It is merely a shorthand for including a list of fields in the record (or class) type being defined. Record inheritance can be useful for grouping data common to several or class or record types.

> *Future.* From the definition of record inheritance, it is apparent that a record of a derived type can be cast legally to any of its base record types. But given their semantics, records can also be legally cast to types with which they have no inheritance relationship. Thus, records do not induce a well-defined type hierarchy.

Example (recordInheritance.chpl).

```
record Center { var x, y: real; }
record Circle : Center {
  var radius: real;
}
record Ellipse : Center {
  var major, minor: real;
}
```

The record `Center` is defined and used as a shorthand in defining the `Circle` and `Ellipse` records. The `Circle` record contains three `real` fields named `x`, `y` and `radius`. The `Ellipse` record contains four `real` fields named `x`, `y`, `major` and `minor`.

The syntax and semantics for accessing methods (including getter methods and hence fields) in a base record type is the same as for accessing fields in a base class (§15.2.2).

16.2.1 Shadowing Base Record Fields

A field in the derived record can be declared with the same name as a field in a base record. Such a field shadows the field in the base record, meaning that the field by the same name in the base record is not directly accessible.

Open issue. A syntax for accessing shadowed fields has not yet been specified.

16.2.2 Overriding Base Record Methods

Future. If a method in a derived record is declared with a signature identical to that of a method in a base record, then it is said to override the base record's method. Since records do not support dynamic dispatch, method overriding is the same as method shadowing: When referenced via the derived record type, the derived type's version of the method is called; when referenced via the base record type, the base record type's version of the method is called.

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

16.3 Record Variable Declarations

A record variable declaration is a variable declaration using a record type. When a variable of record type is declared, storage is allocated sufficient to store all of the fields defined in that record type.

In the context of a class or record or union declaration, the fields are allocated within the object as if they had been declared individually. In this sense, records provide a way to group related fields within a containing class or record type.

In the context of a function body, a record variable declaration causes storage to be allocated sufficient to store all of the fields in that record type. The record variable is initialized through a call to its default initializer. The default initializer for a record is defined in the same way as the default initializer for a class (§15.2.8).

16.3.1 Storage Allocation

Storage for a record variable directly contains the data associated with the fields in the record, in the same manner as variables of primitive types directly contain the primitive values. Record storage is reclaimed when the record variable goes out of scope. No additional storage for a record is allocated or reclaimed. Field data of one variable's record is not shared with data of another variable's record.

16.3.2 Record Initialization

A variable of a record type declared without an initialization expression is initialized through a call to the record's default initializer, passing no arguments. The default initializer for a record is defined in the same way as the default initializer for a class (§15.2.8).

If the new record type is derived from other record types, the default initializer for each base record will be called in lexical order before default initializer for the record itself.

To construct a record as an expression, i.e. without binding it to a variable, the `new` operator is required. In this case, storage is allocated and reclaimed as for a record variable declaration (§16.3.1), except that the temporary record goes out of scope at the end of the enclosing expression.

To initialize a record variable with a non-default value, it can be assigned the value of a constructor call expression. The constructors for a record are defined in the same way as those for a class (§15.3).

> *Rationale*. The `new` keyword disambiguates types from values. This is needed because of the close relationship between constructors and type specifiers for classes and records.

Example (recordCreation.chpl).

```
record TimeStamp {
  var time: string = "1/1/1011";
}

var timestampDefault: TimeStamp;                  // use the default for ’time’
var timestampCustom = new TimeStamp("2/2/2022");  // ... or a different one
writeln(timestampDefault);
writeln(timestampCustom);

var idCounter = 0;
record UniqueID {
  var id: int;
  proc UniqueID() { idCounter += 1; id = idCounter; }
}

writeln(new UniqueID());  // create and use a record value without a variable
writeln(new UniqueID());
```

```
(time = 1/1/1011) (time = 2/2/2022) (id = 1) (id = 2)
```

The variable `timestampDefault` is initialized with `TimeStamp`'s default initializer. The expression `new TimeStamp` creates a record that is assigned to `timestampCustom`. It effectively initializes `timestampCustom` via a call to the constructor with desired arguments. The records created with `new UniqueID()` are discarded after they are used.

As with classes, the user can provide his own constructors (§15.3.1). If any user-defined constructors are supplied, the default initializer cannot be called directly.

16.4 Record Arguments

A function declared with an argument of record type expects that argument to be passed by value unless intents (§13.5) are used. The actual record argument is copied into the corresponding formal argument, consistent with the semantics described for record assignment (§16.9.1).

Example (paramPassing.chpl).

```
record MyColor {
  var color: int;
}
proc printMyColor(mc: MyColor) {
  writeln("my color is ", mc.color);
  mc.color = 6;   // does not affect the caller's record
}
var mc1: MyColor;        // 'color' defaults to 0
var mc2: MyColor = mc1;  // mc1's value is copied into mc2
mc1.color = 3;           // mc1's value is modified
printMyColor(mc2);       // mc2 is not affected by assignment to mc1
printMyColor(mc2);       // ... or by assignment in printMyColor()

proc modifyMyColor(inout mc: MyColor, newcolor: int) {
  mc.color = newcolor;
}
modifyMyColor(mc2, 7);   // mc2 is affected because of the 'inout' intent
printMyColor(mc2);
```

my color is 0 my color is 0 my color is 7

The assignment to `mc1.color` affects only the record stored in `mc1`. The record in `mc2` is not affected by the assignment to `mc1` or by the assignment in `printMyColor`. `mc2` is affected by the assignment in `modifyMyColor` because the intent `inout` is used.

16.5 Record Field Access

A record field is accessed the same way as a class field (§15.4). When a field access is used as an rvalue, the value of that field is returned. When it is used as an lvalue, the value of the record field is updated.

Member access expressions that access parameter fields produce a parameter.

16.5.1 Field Getter Methods

As in classes, field accesses are performed via getter methods (§15.4.1). By default, these methods simply return a reference to the specified field (so they can be written as well as read). The user may redefine these as needed.

16.6 Record Method Calls

A record method may be invoked the same way as a class method (§15.5). Unlike class methods, record methods are resolved at compile time.

16.6.1 The Method Receiver and the *this* Argument

The *receiver* of a record method is similar to and is determined in the same way as the receiver of a class method (§15.5.1). The type of the receiver is the record in which the method is defined. The receiver formal argument can be referred to within the method using the identifier `this`.

The difference from a class method is that the receiver actual argument, which must be a record value, is passed to the record method by reference, rather than by copying. Therefore updates to the receiver made in the method, if any, are visible outside the method.

16.7 The *this* Method

As with classes, records can be supplied with a `this` method. This method defines the behavior of the indexing operator `[]`.

16.8 The *these* Method

A `these` method can be defined for records as well as classes (§15.7). It provides an iterator which iterates over the contents of the record in a user-defined manner.

16.9 Common Operations

16.9.1 Record Assignment

In record assignment, each field of the record on the left-hand side of the assignment is overwritten with the value in the like-named field of the record on the right-hand side as if by assignment.

Assignment from record to a variable of class type is not permitted.

> *Rationale*. If such an assignment were permitted, it would be reasonable to restrict the RHS of the assignment to types derived from the type of the LHS. Since inheriting a record type from a class type is disallowed, this kind of assignment should be disallowed as well.
>
> That's the abbreviated version. The extended rationale is that implementing this kind of assignment would require carrying the type of the record as a run-time type within the class variable.

That further implies that invocation of a method associated with that class would be dispatched dynamically. Either the class variable needs to have an auxilliary field that points to a virtual function table associated with the record type, or the record itself must now accommodate a virtual function table. In any case, this runs somewhat counter to the notion that record types are purely static.

The following example demonstrates record assignment.

Example (assignment.chpl).

```
record R {
  var i: int;
  var x: real;
  proc print() { writeln("i = ", this.i, ", x = ", this.x); }
}
var A: R;
A.i = 3;
A.print();       // "i = 3, x = 0.0"

var C: R;
A = C;
A.print();       // "i = 0, x = 0.0"

C.x = 3.14;
A.print();       // "i = 0, x = 0.0"
```

Prior to the first call to `R.print`, the record `A` is created and initialized to all zeroes. Then, its `i` field is set to `3`. For the second call to `R.print`, the record `C` is created assigned to `A`. Since `C` is default-initialized to all zeroes, those zero values overwrite both values in `A`.

The next clause demonstrates that `A` and `C` are distinct entities, rather than two references to the same object. Assigning `3.14` to `C.x` does not affect the `x` field in `A`.

16.9.2 Assigning Objects to Records

Future. We intend to support assignment of objects to records, with the following specification.

An object (class instance) can be assigned to a record. The same rules apply as those that govern the assignment of one record to another. Namely, the type on the right-hand side must initially contain all of the fields — in the same order and of compatible types — as the record on the left-hand side.

The assignment is performed element-by-element between the right-hand expression and the left-hand expression until all elements in the left-hand expression have been overwritten. The types are compatible if a element-wise assignment is legal for each pair of elements.

A corresponding implicit conversion will be defined as well.

Open issue. Whether reference assignment is to be supported is an open question. If so, it would work like reference assignment in C++ – basically creating an alias for the RHS. References can be used to reduce the length of dereference expression, and also improve performance – especially if that expression is used repeatedly.

16.9.3 Default Comparison Operators

Default functions to overload == and `!=` are defined for records if none are explicitly defined. 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.

16.9.4 Implicit Record Conversions

An expression of record type `D` can be implicitly converted to another record type `C` if

- for each field in `C` there is a like-named field in `D`, and
- an implicit conversion is allowed from the type of the field in `D` to the type of the field in `C`.

Such a conversion removes any fields that are in `D` but not `C`.

The value produced by such a conversion is a record of type `C`. The value of each field of this record is obtained by an implicit conversion of the corresponding field in `D` to that field's type in `C`.

16.10 Differences between Classes and Records

The key differences between records and classes are listed below.

16.10.1 Declarations

Syntactically, class and record type declarations are identical, except that they begin with the `class` and `record` keywords, respectively. Also, a record type can only inherit from other record types. Class inheritance is not permitted.

16.10.2 Storage Allocation

For a variable of record type, storage necessary to contain the data fields has a lifetime equivalent to the scope in which it is declared. No two record variables share the same data. It is not necessary to call `new` to create a record.

By contrast, a class variable contains only a reference to a class instance. A class instance is created through a call to its `new` operator. Storage for a class instance, including storage for the data associated with the fields in the class, is allocated and reclaimed separately from variables referencing that instance. The same class instance can be referenced by multiple class variables.

16.10.3 Assignment

Assignment to a class variable is performed by reference, whereas assignment to a record is performed by value. When a variable of class type is assigned to another variable of class type, they both become names for the same object. In contrast, when a record variable is assigned to another record variable, then contents of the source record are copied into the target record field-by-field.

When a variable of class type is assigned to a record, matching fields (matched by name) are copied from the class instance into the corresponding record fields. Subsequent changes to the fields in the target record have no effect upon the class instance.

Assignment of a record to a class variable is not permitted.

16.10.4 Arguments

The semantics of argument passing is determined by the type of the formal argument (as declared inside the function header). An actual argument is of a type compatible with the formal argument only if it is legal to assign the actual to the formal.

Specifically, if the formal argument is of class type, the actual argument must be of that class type or of a type derived from that class type. If the formal argument is of a record type, then it is only necessary for the fields in the actual argument to "cover" the fields in the formal argument type.

The receiver argument is passed by value for class methods but is passed by reference for record methods. In both cases modifications to the receiver fields are visible outside the method.

16.10.5 Inheritance

The difference between record inheritance and class inheritance is that for records there is no dynamic dispatch. The record type of a variable is the exact type of that variable, i.e. a variable of a base record type cannot store a derived record type.

Casting a derived record type to a base record type truncates all fields except those belonging to the base record type. In the same way, only those methods accessible to the base record type may be invoked using the result of such a cast.

16.10.6 Shadowing and Overriding

Class variables have run-time types and (therefore) support dynamic dispatch. Records are statically typed, so they do not have run-time types and they do not support dynamic dispatch.

As a result, in record type hierarchies, shadowing and overriding are the same. Which field is accessed and/or which method is invoked is determined statically by the declared type of the record being referenced.

16.10.7 No *nil* Value

Records do not provide a counterpart of the `nil` value. A variable of record type is associated with storage throughout its lifetime, so `nil` has no meaning with respect to records.

16.10.8 The *delete* operator

Calling `delete` on a record is illegal.

16.10.9 Default Comparison Operators

For records, the compiler will supply default comparison operators if they are not supplied by the user. The compiler does not supply default comparison operators for classes.

17 Unions

Unions have the semantics of records, however, only one field in the union can contain data at any particular point in the program's execution. Unions are safe so that an access to a field that does not contain data is a runtime error. When a union is constructed, it is in an unset state so that no field contains data.

17.1 Union Types

The syntax of a union type is summarized as follows:

union-type:
  identifier

The union type is specified by the name of the union type. This simplification from class and record types is possible because generic unions are not supported.

17.2 Union Declarations

A union is defined with the following syntax:

union-declaration-statement:
  extern$_{opt}$ **union** *identifier* { *union-statement-list* }

union-statement-list:
  union-statement
  union-statement union-statement-list

union-statement:
  type-declaration-statement
  procedure-declaration-statement
  iterator-declaration-statement
  variable-declaration-statement
  empty-statement

If the `extern` keyword appears before the `union` keyword, then an external union type is declared. An external union is used within Chapel for type and field resolution, but no corresponding backend definition is generated. It is presumed that the definition of an external union type is supplied by a library or the execution environment.

17.2.1 Union Fields

Union fields are accessed in the same way that record fields are accessed. It is a runtime error to access a field that is not currently set.

Union fields should not be specified with initialization expressions.

17.3 Union Assignment

Union assignment is by value. The field set by the union on the right-hand side of the assignment is assigned to the union on the left-hand side of the assignment and this same field is marked as set.

17.4 The Type Select Statement and Unions

The type-select statement can be applied to unions to access the fields in a safe way by determining the type of the union.

18 Ranges

A *range* is a first-class, constant-space representation of a regular sequence of integer indices. Ranges support iteration over the sequences they represent and are the basis for defining domains (§19).

Ranges are presented as follows:

- definition of the key range concepts §18.1
- range types §18.2
- range values §18.3
- range assignment §18.4.1
- operators on ranges §18.5
- predefined functions on ranges §18.6

18.1 Range Concepts

A range has four primary properties. Together they define the sequence of indices that the range represents, or the *represented sequence*, as follows.

- The *low bound* is either an integer or -∞.
- The *high bound* is either an integer or +∞. The low and high bounds determine the span of the represented sequence. Chapel does not represent ∞ explicitly. Instead, infinite bound(s) are represented implicitly in the range's type (§18.2). When the low and/or high bound is ∞, the represented sequence is unbounded in the corresponding direction(s).
- The *stride* is a non-zero integer. It defines the distance between any two adjacent members of the represented sequence. The sign of the stride indicates the direction of the sequence:
  - $stride > 0$ indicates an increasing sequence,
  - $stride < 0$ indicates a decreasing sequence.
- The *alignment* is either an integer or is *ambiguous*. It defines how the represented sequence's members are aligned relative to 0. For a range with a stride other than 1 or -1, ambiguous alignment means that the represented sequence is undefined. In such a case, certain operations discussed later result in an error.

  Open issue. We consider disallowing ambiguous alignment for ranges whose both bounds are integers (not ∞), in order to enable more efficient implementation.

More formally, the represented sequence for the range $(low, high, stride, alignmt)$ contains all indices ix such that:

$low \le ix \le high$ and $ix \equiv alignmt \pmod{\lvert stride \rvert}$	if $alignmt$ is not ambiguous
$low \le ix \le high$	if $stride = 1$ or $stride = -1$
the represented sequence is undefined	otherwise

The sequence, if defined, is increasing if $stride > 0$ and decreasing if $stride < 0$.

If the represented sequence is defined but there are no indices satisfying the applicable equation(s) above, the range and its represented sequence are *empty*.

We will say that an integer ix is *aligned* w.r.t. the range $(low, high, stride, alignmt)$ if:

- $alignmt$ is not ambiguous and $ix \equiv alignmt \pmod{\lvert stride \rvert}$, or
- $stride$ is 1 or -1.

Furthermore, ∞ is never aligned.

Ranges have the following additional properties.

- A range is *ambiguously aligned* if
  - its alignment is ambiguous, and
  - its stride is neither 1 nor -1.
- The *first index* is the first member of the represented sequence.

  A range *has no* first index when the first member is undefined, that is, in the following cases:
  - the range is ambiguously aligned,
  - the represented sequence is empty,
  - the represented sequence is increasing and the low bound is -∞,
  - the represented sequence is decreasing and the high bound is +∞.
- The *last index* is the last member of the represented sequence.

  A range *has no* last index when the last member is undefined, that is, in the following cases:
  - it is ambiguously aligned,
  - the represented sequence is empty,
  - the represented sequence is increasing and the high bound is +∞,
  - the represented sequence is decreasing and the low bound is -∞.
- The *aligned low bound* is the smallest integer that is greater than or equal to the low bound and is aligned w.r.t. the range, if such an integer exists.

  The aligned low bound equals the smallest member of the represented sequence, when both exist.
- The *aligned high bound* is the largest integer that is less than or equal to the high bound and is aligned w.r.t. the range, if such an integer exists.

  The aligned high bound equals the largest member of the represented sequence, when both exist.
- The range is *iterable*, that is, it is legal to iterate over it, if is has the first index.

18.2 Range Types

The type of a range is characterized by three parameters:

- `idxType` is the type of the indices of the range's represented sequence. However, when the range's low and/or high bound is ∞, the represented sequence also contains indices that are not representable by `idxType`.

  `idxType` must be an integral type and is `int` by default. The range's low bound and high bound (when they are not ∞) and alignment are of the type `idxType`. The range's stride is of the signed integer type that has the same bit size as `idxType`.
- `boundedType` indicates which of the range's bounds are not ∞. `boundedType` is an enumeration constant of the type `BoundedRangeType`. It is discussed further below.
- `stridable` is a boolean that determines whether the range's stride can take on values other than 1. `stridable` is `false` by default. A range is called *stridable* if its type's `stridable` is `true`.

`boundedType` is one of the constants of the following type:

```
enum BoundedRangeType { bounded, boundedLow, boundedHigh, boundedNone };
```

The value of `boundedType` determines which bounds of the range are integers (making the range "bounded", as opposed to infinite, in the corresponding direction(s)) as follows:

- `bounded`: both bounds are integers.
- `boundedLow`: the low bound is an integer (the high bound is $+\infty$).
- `boundedHigh`: the high bound is an integer (the low bound is $-\infty$).
- `boundedNone`: neither bound is an integer (both bounds are ∞).

`boundedType` is `BoundedRangeType.bounded` by default.

The parameters `idxType`, `boundedType` and `stridable` affect all values of the corresponding range type. For example, the range's low bound is $-\infty$ if and only if the `boundedType` of that range's type is either `boundedHigh` or `boundedNone`.

> *Rationale*. Providing `boundedType` and `stridable` in a range's type allows the compiler to identify the more common cases where the range is `bounded` and/or its stride is 1. The compiler can also detect user and library code that is specialized to these cases. As a result, the compiler has the opportunity to optimize these cases and the specialized code more aggressively.

A range type has the following syntax:

range-type:
  range (*named-expression-list*)

That is, a range type is obtained as if by invoking the range type constructor (§22.3.4) that has the following header:

```
proc range(type idxType = int,
           param boundedType = BoundedRangeType.bounded,
           param stridable = false) type
```

As a special case, the keyword `range` without a parenthesized argument list refers to the range type with the default values of all its parameters, i.e., `range(int, BoundedRangeType.bounded, false)`.

> *Example (rangeVariable.chpl).* The following declaration declares a variable `r` that can represent ranges of 64-bit integers, with both high and low bounds specified, and the ability to have a stride other than 1.
>
> ```
> var r: range(int(64), BoundedRangeType.bounded, stridable=true);
> ```

18.3 Range Values

A range value consists of the range's four primary properties (§18.1): low bound, high bound, stride and alignment.

18.3.1 Range Literals

Range literals are specified with the following syntax.

```
range-literal:
  expression .. expression
  expression ..
  .. expression
  ..
```

The expressions to the left and to the right of `..`, when given, are called the low bound and the high bound expression, respectively.

The type of a range literal is a range with the following parameters:

- `idxType` is determined as follows:
    - If both the low bound and the high bound expressions are given and have the same integral type, then `idxType` is that type.
    - If both the low bound and the high bound expressions are given and an implicit conversion is allowed from each expression's type to the same integral type, then `idxType` is that integral type.
    - If only one bound expression is given and it has an integral type or an implicit conversion is allowed from that expression's type to an integral type, then `idxType` is that integral type.
    - If neither bound expression is given, then `idxType` is `int`.
    - Otherwise, the range literal is not legal.

- `boundedType` is a value of the type `BoundedRangeType` that is determined as follows:
  - `bounded`, if both the low bound and the high bound expressions are given,
  - `boundedLow`, if only the high bound expression is given,
  - `boundedHigh`, if only the low bound expression is given,
  - `boundedNone`, if neither bound expression is given.
- `stridable` is `false`.

The value of a range literal is as follows:

- The low bound is given by the low bound expression, if present, and is -∞ otherwise.
- The high bound is given by the upper bound expression, if present, and is +∞ otherwise.
- The stride is 1.
- The alignment is ambiguous.

18.3.2 Default Values

The default value for a range type depends on the type's `boundedType` parameter as follows:

- `1..0` (an empty range) if `boundedType` is `bounded`
- `1..` if `boundedType` is `boundedLow`
- `..0` if `boundedType` is `boundedHigh`
- `..` if `boundedType` is `boundedNone`

> *Rationale*. We use 0 and 1 to represent an empty range because these values are available for any `idxType`.
>
> We have not found the natural choice of the default value for `boundedLow` and `boundedHigh` ranges. The values indicated above are distinguished by the following property. Slicing the default value for a `boundedLow` range with the default value for a `boundedHigh` range (or visa versa) produces an empty range, matching the default value for a `bounded` range

18.4 Common Operations

18.4.1 Range Assignment

Assigning one range to another results in the target range copying the low and high bounds, stride, and alignment from the source range.

Range assignment is legal when:

- An implicit conversion is allowed from `idxType` of the source range to `idxType` of the destination range type,
- the two range types have the same `boundedType`, and
- either the destination range is stridable or the source range's stride is 1.

18.4.2 Range Comparisons

Ranges can be compared using equality and inequality.

```
proc ==(r1: range(?), r2: range(?)): bool
```

Returns `true` if the two ranges have the same represented sequence or the same four primary properties, and `false` otherwise.

18.4.3 Iterating over Ranges

A range can be used as an iterator expression in a loop. This is legal only if the range is iterable. In this case the loop iterates over the members of the range's represented sequence, in the order defined by the sequence. If the range is empty, no iterations are executed.

Iterating over Unbounded Ranges in Zippered Iterations

When a range with the first index but without the last index is used in a zippered iteration (§11.8.1), it generates as many indices as needed to match the other iterator(s).

Example (zipWithUnbounded.chpl). The code

```
for i in (1..5, 3..) do
  write(i, "; ");
```

produces the output

```
(1, 3); (2, 4); (3, 5); (4, 6); (5, 7);
```

18.5 Range Operators

The following operators can be applied to range expressions and are described in this section: stride (`by`), alignment (`align`), count (`#`) and slicing (`()` or `[]`). Chapel also defines a set of functions that operate on ranges. They are described in §18.6.

range-expression:
  expression
  strided-range-expression
  counted-range-expression
  aligned-range-expression
  sliced-range-expression

18.5.1 By Operator

The `by` operator selects a subsequence of the range's represented sequence, optionally reversing its direction. The operator takes two arguments, a base range and an integral step. It produces a new range whose represented sequence contains each |step|-th element of the base range's represented sequence. The operator reverses the direction of the represented sequence if step<0. If the resulting sequence is increasing, it starts at the base range's aligned low bound, if it exists. If the resulting sequence is decreasing, it starts at the base range's aligned high bound, if it exists. Otherwise, the base range's alignment is used to determine which members of the represented sequence to retain. If the base range's represented sequence is undefined, the resulting sequence is undefined, too.

The syntax of the `by` operator is:

```
strided-range-expression:
  range-expression by step-expression

step-expression:
  expression
```

The type of the step must be a signed integer of the same bit size as the base range's `idxType`, or an implicit conversion must be allowed to that type from the step's type. The type of the step is also allowed to be the base range's `idxType` itself. It is an error for the step to be zero.

> *Future.* We consider allowing the step to be of any integer type, for maximum flexibility.

The type of the result of the `by` operator is the type of the base range, but with the `stridable` parameter set to `true`.

Formally, the result of the `by` operator is a range with the following primary properties:

- The low and upper bounds are the same as those of the base range.
- The stride is the product of the base range's stride and the step, cast to the base range's stride type before multiplying.
- The alignment is:
  - the aligned low bound of the base range, if such exists and the stride is positive;
  - the aligned high bound of the base range, if such exists and the stride is negative;
  - the same as that of the base range, otherwise.

> *Example (rangeByOperator.chpl).* In the following declarations, range `r1` represents the odd integers between 1 and 20. Range `r2` strides `r1` by two and represents every other odd integer between 1 and 20: 1, 5, 9, 13 and 17.
>
> ```
> var r1 = 1..20 by 2;
> var r2 = r1 by 2;
> ```

> *Rationale. Why isn't the high bound specified first if the stride is negative?* The reason for this choice is that the `by` operator is binary, not ternary. Given a range `R` initialized to `1..3`, we want `R by -1` to contain the ordered sequence $3, 2, 1$. But then `R by -1` would be different from `3..1 by -1` even though it should be identical by substituting the value in R into the expression.

18.5.2 Align Operator

The `align` operator can be applied to any range, and creates a new range with the given alignment.

The syntax for the `align` operator is:

aligned−range−expression:
  range−expression **align** *expression*

The type of the resulting range expression is the same as that of the range appearing as the left operand. An implicit conversion from the type of the right operand to the index type of the operand range must be allowed. The resulting range has the same low and high bounds and stride as the source range. The alignment equals the `align` operator's right operand and therefore is not ambiguous.

Example (alignedStride.chpl).

```
var r1 = 0 .. 10 by 3 align 0;
for i in r1 do
  write(" ", i);                        // Produces "0 3 6 9".
writeln();

var r2 = 0 .. 10 by 3 align 1;
for i in r2 do
  write(" ", i);                        // Produces "1 4 7 10".
writeln();
```

When the stride is negative, the same indices are printed in reverse:

Example (alignedNegStride.chpl).

```
var r3 = 0 .. 10 by -3 align 0;
for i in r3 do
  write(" ", i);                        // Produces "9 6 3 0".
writeln();

var r4 = 0 .. 10 by -3 align 1;
for i in r4 do
  write(" ", i);                        // Produces "10 7 4 1".
writeln();
```

To create a range aligned relative to its `first` index, use the `offset` method (§18.6.4).

18.5.3 Count Operator

The `#` operator takes a range and an integral count and creates a new range containing the specified number of indices. The low or high bound of the left operand is preserved, and the other bound adjusted to provide the specified number of indices. If the count is positive, indices are taken from the start of the range; if the count is negative, indices are taken from the end of the range. The count must be less than or equal to the `length` of the range.

counted−range−expression:
  range−expression # *expression*

The *idxType* of the resulting range is the same type that would be obtained by adding the integral count value to a value with the range's *idxType*. The stride and alignment of the resulting range are the same as those of the initial range.

Depending on the sign of the count and the stride, the high or low bound is unchanged and the other bound is adjusted so that it is $c * stride - 1$ units away. Specifically:

- If the count times the stride is positive, the low bound is preserved and the high bound is adjusted to be one less than the low bound plus that product.
- Iff the count times the stride is negative, the high bound is preserved and the low bound is adjusted to be one greater than the high bound plus that product.

> *Rationale*. Following the principle of preserving as much information from the original range as possible, we must still choose the other bound so that exactly *count* indices lie within the range. Making the two bounds lie $count * stride - 1$ apart will achieve this, regardless of the current alignment of the range.
>
> This choice also has the nice symmetry that the alignment can be adjusted without knowing the bounds of the original range, and the same number of indices will be produced:
>
> ```
> r # 4 align 0   // Contains four indices.
> r # 4 align 1   // Contains four indices.
> r # 4 align 2   // Contains four indices.
> r # 4 align 3   // Contains four indices.
> ```

It is an error to apply the count operator with a positive count to a range that has no first index. It is also an error to apply the count operator with a negative count to a range that has no last index. It is an error to apply the count operator to a range that is ambiguously aligned.

> *Example (rangeCountOperator.chpl)*. The following declarations result in equivalent ranges.
>
> ```
> var r1 = 1..10 by -2 # -3;
> var r2 = ..6 by -2 # 3;
> var r3 = -6..6 by -2 # 3;
> var r4 = 1..#6 by -2;
> ```
>
> Each of these ranges represents the ordered set of three indices: 6, 4, 2.

18.5.4 Arithmetic Operators

The following arithmetic operators are defined on ranges and integral types:

```
proc +(r: range, s: integral): range
proc +(s: integral, r: range): range
proc -(r: range, s: integral): range
```

The + and - operators apply the scalar via the operator to the range's low and high bounds, producing a shifted version of the range. If the operand range is unbounded above or below, the missing bounds are ignored. The index type of the resulting range is the type of the value that would result from an addition between the scalar value and a value with the range's index type. The bounded and stridable parameters for the result range are the same as for the input range.

The stride of the resulting range is the same as the stride of the original. The alignment of the resulting range is shifted by the same amount as the high and low bounds. It is permissible to apply the shift operators to a range that is ambiguously aligned. In that case, the resulting range is also ambiguously aligned.

> *Example (rangeAdd.chpl).* The following code creates a bounded, non-stridable range `r` which has an index type of `int` representing the indices $0, 1, 2, 3$. It then uses the `+` operator to create a second range `r2` representing the indices $1, 2, 3, 4$. The `r2` range is bounded, non-stridable, and is represented by indices of type `int`.
>
> ```
> var r = 0..3;
> var r2 = r + 1;    // 1..4
> ```

18.5.5 Range Slicing

Ranges can be *sliced* using other ranges to create new sub-ranges. The resulting range represents the intersection between the two ranges' represented sequences. The stride and alignment of the resulting range are adjusted as needed to make this true. `idxType` and the sign of the stride of the result are determined by the first operand.

Range slicing is specified by the syntax:

```
sliced-range-expression:
  range-expression ( range-expression )
  range-expression [ range-expression ]
```

If either of the operand ranges is ambiguously aligned, then the resulting range is also ambiguously aligned. In this case, the result is valid only if the strides of the operand ranges are relatively prime. Otherwise, an error is generated at run time.

> *Rationale*. If the strides of the two operand ranges are relatively prime, then they are guaranteed to have some elements in their intersection, regardless whether their relative alignment can be determined. In that case, the bounds and stride in the resulting range are valid with respect to the given inputs. The alignment can be supplied later to create a valid range.
>
> If the strides are not relatively prime, then the result of the slicing operation would be completely ambiguous. The only reasonable action for the implementation is to generate an error.

If the resulting represented sequence cannot be expressed as a range of the expected type, an error is generated. This can happen, for example, when the operands represent all odd and all even numbers, or when the first operand is an unbounded range with unsigned `idxType` and the second operand represents only negative numbers.

> *Example (rangeSlicing.chpl).* In the following example, `r` represents the integers from 1 to 20 inclusive. Ranges `r2` and `r3` are defined using range slices and represent the indices from 3 to 20 and the odd integers between 1 and 20 respectively. Range `r4` represents the odd integers between 1 and 20 that are also divisible by 3.

```
var r = 1..20;
var r2 = r[3..];
var r3 = r[1.. by 2];
var r4 = r3[0.. by 3];
```

18.6 Predefined Functions on Ranges

18.6.1 Range Type Parameters

proc *range*.`boundedType : BoundedRangeType`

Returns the `boundedType` parameter of the range's type.

proc *range*.`idxType` : **type**

Returns the `idxType` parameter of the range's type.

proc *range*.`stridable` : **bool**

Returns the `stridable` parameter of the range's type.

18.6.2 Range Properties

Most of the methods in this subsection report on the range properties defined in §18.1. A range's represented sequence can be examined, for example, by iterating over the range in a for loop §11.8.

Open issue. The behavior of the methods that report properties that may be undefined, ∞, or ambiguous, may change.

proc *range*.`aligned` : **bool**

Reports whether the range's alignment is *not* ambiguous.

proc *range*.`alignedHigh : idxType`

Returns the range's aligned high bound. If the aligned high bound is undefined (does not exist), the behavior is undefined.

Example (alignedHigh.chpl). The following code:

```
var r = 0..20 by 3;
writeln(r.alignedHigh);
```

produces the output

```
18
```

proc *range*.`alignedLow : idxType`

Returns the range's aligned low bound. If the aligned low bound is undefined (does not exist), the behavior is undefined.

proc *range*.`alignment : idxType`

Returns the range's alignment. If the alignment is ambiguous, the behavior is undefined. See also `aligned`.

proc *range*.`first : idxType`

Returns the range's first index. If the range has no first index, the behavior is undefined. See also `hasFirst`.

proc *range*.`hasFirst():` **bool**

Reports whether the range has the first index.

proc *range*.`hasHighBound()` **param: bool**

Reports whether the range's high bound is *not* +∞.

proc *range*.`hasLast():` **bool**

Reports whether the range has the last index.

proc *range*.`hasLowBound()` **param: bool**

Reports whether the range's low bound is *not* -∞.

proc *range*.`high : idxType`

Returns the range's high bound. If the high bound is +∞, the behavior is undefined. See also `hasHighBound`.

proc *range*.`isAmbiguous():` **bool**

Reports whether the range is ambiguously aligned.

proc *range*.`last : idxType`

Returns the range's last index. If the range has no last index, the behavior is undefined. See also `hasLast`.

proc *range*.`length : idxType`

Returns the number of indices in the range's represented sequence. If the represented sequence is infinite or is undefined, an error is generated.

proc *range*.`low : idxType`

Returns the range's low bound. If the low bound is -∞, the behavior is undefined. See also `hasLowBound`.

proc *range*.`stride :` **int**`(numBits(idxType))`

Returns the range's stride. This will never return 0. If the range is not stridable, this will always return 1.

18.6.3 Other Queries

proc *range*.`boundsCheck(r2: `**range**`(?)): `**bool**

Returns `false` if either range is ambiguously aligned. Returns `true` if range `r2` lies entirely within this range and `false` otherwise.

proc `ident(r1: `**range**`(?), r2: `**range**`(?)): `**bool**

Returns `true` if the two ranges are the same in every respect: i.e. the two ranges have the same `idxType`, `boundedType`, `stridable`, `low`, `high`, `stride` and `alignment` values.

proc *range*.`indexOrder(i: idxType): idxType`

If `i` is a member of the range's represented sequence, returns an integer giving the ordinal index of `i` within the sequence using 0-based indexing. Otherwise, returns `(-1):idxType`. It is an error to invoke `indexOrder` if the represented sequence is not defined or the range does not have the first index.

Example. The following calls show the order of index 4 in each of the given ranges:

```
(0..10).indexOrder(4) == 4
(1..10).indexOrder(4) == 3
(3..5).indexOrder(4) == 1
(0..10 by 2).indexOrder(4) == 2
(3..5 by 2).indexOrder(4) == -1
```

proc *range*.`member(i: idxType): `**bool**

Returns `true` if the range's represented sequence contains `i`, `false` otherwise. It is an error to invoke `member` if the represented sequence is not defined.

proc *range*.`member(other: `**range**`): `**bool**

Reports whether `other` is a subrange of the receiver. That is, if the represented sequences of the receiver and `other` are defined and the receiver's sequence contains all members of the `other`'s sequence.

18.6.4 Range Transformations

proc *range*.`alignHigh()`

Sets the high bound of this range to its aligned high bound, if it is defined. Generates an error otherwise.

proc *range*.`alignLow()`

Sets the low bound of this range to its aligned low bound, if it is defined. Generates an error otherwise.

proc *range*.`expand(i: idxType)`

Returns a new range whose bounds are extended by i units on each end. If $i < 0$ then the resulting range is contracted by its absolute value. In symbols, given that the operand range is represented by the tuple (l, h, s, a), the result is $(l - i, h + i, s, a)$. The stride and alignment of the original range are preserved. If the operand range is ambiguously aligned, then so is the resulting range.

proc *range*.`exterior(i: idxType)`

Returns a new range containing the indices just outside the low or high bound of the range (low if $i < 0$ and high otherwise). The stride and alignment of the original range are preserved. Let the operand range be denoted by the tuple (l, h, s, a). Then:

if $i < 0$, the result is $(l + i, l - 1, s, a)$,
if $i > 0$, the result is $(h + 1, h + i, s, a)$, and
if $i = 0$, the result is (l, h, s, a).

If the operand range is ambiguously aligned, then so is the resulting range.

proc *range*.`interior(i: idxType)`

Returns a new range containing the indices just inside the low or high bound of the range (low if $i < 0$ and high otherwise). The stride and alignment of the original range are preserved. Let the operand range be denoted by the tuple (l, h, s, a). Then:

if $i < 0$, the result is $(l, l - (i - 1), s, a)$,
if $i > 0$, the result is $(h - (i - 1), h, s, a)$, and
if $i = 0$, the result is (l, h, s, a).

This differs from the behavior of the count operator, in that `interior()` preserves the alignment, and it uses the low and high bounds rather than `first` and `last` to establish the bounds of the resulting range. If the operand range is ambiguously aligned, then so is the resulting range.

proc *range*.`offset(n: idxType)`

Returns a new range whose alignment is this range's first index plus `n`. The new alignment, therefore, is not ambiguous. If the range has no first index, a run-time error is generated.

proc *range*.`translate(i: integral)`

Returns a new range with its `low`, `high` and `alignment` values adjusted by i. The `stride` value is preserved. If the range's alignment is ambiguous, the behavior is undefined.

19 Domains

A *domain* is a first-class representation of an index set. Domains are used to specify iteration spaces, to define the size and shape of arrays (§20), and to specify aggregate operations like slicing. A domain can specify a single- or multi-dimensional rectangular iteration space or represent a set of indices of a given type. Domains can also represent a subset of another domain's index set, using either a dense or sparse representation. A domain's indices may potentially be distributed across multiple locales as described in §27, thus supporting global-view data structures.

In the next subsection, we introduce the key characteristics of domains. In §19.2, we discuss the types and values that can be associated with a base domain. In §19.3, we discuss the types and values of simple subdomains that can be created from those base domains. In §19.4, we discuss the types and values of sparse subdomains. The remaining sections describe the important manipulations that can be perfomed with domains, as well as the predefined operators and functions defined for domains.

19.1 Domain Overview

There are three *kinds* of domain, distinguished by their subset dependencies: *base domains*, *subdomains* and *sparse subdomains*. A base domain describes an index set spanning one or more dimensions. A subdomain creates an index set that is a subset of the indices in a base domain or another subdomain. Sparse subdomains are subdomains which can represent sparse index subsets efficiently. Simple subdomains are subdomains that are not sparse. These relationships can be represented as follows:

```
domain-type:
  base-domain-type
  simple-subdomain-type
  sparse-subdomain-type
```

Domains can be further classified according to whether they are *regular* or *irregular*. A regular domain represents a rectangular iteration space and can have a compact representation whose size is independent of the number of indices. Rectangular domains, with the exception of sparse subdomains, are regular.

An irregular domain can store an arbitrary set of indices of an arbitrary but homogeneous index type. Irregular domains typically require space proportional to the number of indices being represented. All *associative* domain types and their subdomains (including sparse subdomains) are irregular. Sparse subdomains of regular domains are also irregular.

An index set can be either *ordered* or *unordered* depending on whether its members have a well-defined order relationship. All regular and enumerated domains are ordered. All other associative domains are unordered.

The type of a domain describes how a domain is represented and the operations that can be performed upon it, while its value is the set of indices it represents. In addition to storing a value, each domain variable has an identity that distinguishes it from other domains that may have the same type and value. This identity is used to define the domain's relationship with subdomains, index types (§19.5), and arrays (§20.11).

> *Open issue*. In the future, it is likely that we will support a means of creating domain aliases, much as we support array aliases currently.

The runtime representation of a domain is controlled by its domain map. Domain maps are presented in §27.

19.2 Base Domains

Base domain types can be classified as regular or irregular. Dense and strided rectangular domains are regular domains. Irregular base domain types include all of the associative domain types.

base−domain−type:
  rectangular−domain−type
  associative−domain−type

These base domain types are discussed in turn in the following subsections.

19.2.1 Rectangular Domains

Rectangular domains describe multidimensional rectangular index sets. They are characterized by a tensor product of ranges and represent indices that are tuples of an integral type. Because their index sets can be represented using ranges, regular domain values typically require only $O(1)$ space.

Rectangular Domain Types

Rectangular domain types are parameterized by three things:

- `rank` a positive `int` value indicating the number of dimensions that the domain represents;
- `idxType` a type member representing the index type for each dimension; and
- `stridable` a `bool` parameter indicating whether any of the domain's dimensions will be characterized by a strided range.

If `rank` is 1, the index type represented by a rectangular domain is `idxType`. Otherwise, the index type is the homogenous tuple type `rank*idxType`. If unspecified, `idxType` defaults to `int` and `stridable` defaults to `false`.

> *Open issue*. We may represent a rectangular domain as rank*idxType even if rank is 1. This would eliminate a lot of code currently used to support the special (rank == 1) case.

The syntax of a rectangular domain type is summarized as follows:

rectangular−domain−type:
  domain (*named−expression−list*)

where *named-expression-list* permits the values of `rank`, `idxType`, and `stridable` to be specified using standard type signature.

> *Example (typeFunctionDomain.chpl).* The following declarations both create an unintialized rectangular domain with three dimensions, with `int` indices:
>
> ```
> var D1 : domain(rank=3, idxType=int, stridable=false);
> var D2 : domain(3*int);
> ```

Rectangular Domain Values

Each dimension of a rectangular domain is a range of type `range(idxType, BoundedRangeType.bounded, stridable)`. The index set for a rank 1 domain is the set of indices described by its singleton range. The index set for a rank n domain is the set of all `n*idxType` tuples described by the tensor product of its ranges. When expanded (as by an iterator), rectangular domain indices are ordered according to the lexicographic order of their values. That is, the index with the highest rank is listed first and changes most slowly.[1]

> *Future.* Unbounded ranges may be supported.

Literal rectangular domain values are represented by a comma-separated list of range expressions of matching `idxType` enclosed in square brackets:

> *Future.* Will we support domains with heterogeneous index types?

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

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

The type of a rectangular domain literal is defined as follows:

- `rank` = the number of range expressions in the literal;
- `idxType` = the type of the range expressions;
- `stridable` = `true` if any of the range expressions are stridable, otherwise `false`.

If the index types in the ranges differ and all of them can be promoted to the same type, then that type is used as the `idxType`. Otherwise, the domain literal is invalid.

> *Example.* The expression `[1..5, 1..5]` defines a rectangular domain with type `domain(rank=2, idxType=int, strided=false)`. It is a 5×5 domain with the indices:
>
> $$(1,1),(1,2),\ldots,(1,5),(2,1),\ldots(5,5). \tag{19.1}$$

---

[1] This is also known as row-major ordering.

A domain expression may contain bounds which are evaluated at runtime.

Example. In the code

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

`D` is defined as a two-dimensional, nonstridable rectangular domain with an index type of `2*int` and is initialized to contain the set of indices (i, j) for all i and j such that $i \in 1, 2, \ldots, n$ and $j \in 1, 2, \ldots, n$.

The default value of a range type is the `rank` default range values for type:

```
range(idxType, BoundedRangeType.bounded, stridable)
```

Example (rectangularDomain.chpl). The following creates a two-dimensional rectangular domain and then uses this to declare an array. The domain's `dim()` accessor is used to fill the array with some values. Then it is printed out.

```
var D : domain(2) = [1..2, 1..7];
var A : [D] int;
for i in D.dim(1) do
  for j in D.dim(2) do
    A[i,j] = 7 * i**2 + j;
writeln(A);
```

```
8 9 10 11 12 13 14 29 30 31 32 33 34 35
```

19.2.2 Associative Domains

Associative domains represent an arbitrary set of indices of a given type and can be used to describe sets or to create dictionary-style arrays. Indices may be virtually any value type, including integers, reals, objects and enumerators.

Associative Domain Types

An associative domain type is parameterized by `idxType`, the type of the index that it stores. The syntax is as follows:

associative-domain-type:
  domain (*associative-index-type*)
  domain (*enum-type*)
  domain (**opaque**)

associative-index-type:
  type-specifier

Open issue. Should there be an `opt_idxTypeEquals` in the domain type above?

The three expansions of *associative-domain-type* correspond to the three kinds of associative domain listed below.

1. In general, associative domains have an index set whose members are of the associative index type.
2. Enumerated domains are a special case, in which the *associative-index-type* is an enumerated type. Enumerated types are handled specially during initialization, as described below.
3. Opaque domain types are parameterized by the type `opaque`, indicating that the index values are anonymous and have no obvious representational name or value.

When an associative domain is used as the index set of an array, the relation between the indices and the array elements can be thought of as a map between the values of the index set and the elements stored in the array. For enumerated domains, the mapping is between values of the enumeration and the associated array values. For opaque domains, the indices have no inherent names and are therefore anonymous. Opaque domains can be used to represent sets and to build unstructured arrays (similar to pointer-based data structures in conventional languages).

Associative Domain Values

An associative domain's value is simply the set of all index values that the domain describes. The indices of an associative domain are typically unordered.

There is no literal syntax for an associative domain, though a tuple of values of type `idxType` can be used to initialize a variable of associative domain type.

In general, the default value for an associative domain is the empty set.

Enumerated Domain Values Associative domains over enumerated types are ordered according to the order in which the enumeration's identifiers were declared.

Being a special case of associative domains, an enumerated domain can be initialized by tuple of enumerators. All of the enumerators must be members of the enumeration type used to declare the domain.

The default value for an enumerated domain is the set of all identifiers in the enumeration.

> *Rationale*. The decision to have enumerated domains start fully populated was based on the observation that enumerations have a finite, typically small number of values and that it would be common to declare arrays with elements corresponding to each identifier in the enumeration. Further, in terms of usability it is simpler to clear a fully-populated domain than to fully populate an empty one.
>
> In addition, fully-populated constant enumerated domains are an important case for compiler optimizations, particularly if the numeric values of the enumeration are consecutive.

> *Future*. We may generally support a `startPopulated` parameter on associative domains, to unify this capability with other values.

Opaque Domain Values An opaque domain's value is simply the unordered set of anonymous indices that the domain describes.

There is no literal syntax for an opaque domain due to the fact that the indices have no inherent names. The members of an opaque domain can only be created using the `add` function or + operator.

The default value for an opaque domain is the empty set.

19.3 Simple Subdomains

A subdomain is a domain whose indices are guaranteed to be a subset of those described by another domain known as its *parent domain*. A subdomain has the same type as its parent domain, and by default it inherits the domain map of its parent domain. All domain types support subdomains.

Simple subdomains are subdomains which are not sparse. Sparse subdomains are discussed in the following section (§19.4). A simple subdomain inherits its representaion (regular or irregular) from its base domain (or base subdomain). A sparse subdomain is always irregular, even if its base domain is regular.

In all other respects, the two kinds of subdomain behave identically. In this specification, "subdomain" refers to both simple and sparse subdomains, unless it is specifically distinguished as one or the other.

> *Rationale*. Subdomains are provided in Chapel for a number of reasons: to facilitate the ability of the compiler or a reader to reason about the inter-relationship of distinct domain variables; to support the author's ability to omit redundant domain mapping specifications; to support the compiler's ability to reason about the relative alignment of multiple domains; and to improve the compiler's ability to prove away bounds checks for array accesses.

19.3.1 Simple Subdomain Types

A simple subdomain type is specified using the following syntax:

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

This declares that *domain-expression* is the parent domain of this subdomain type. A simple subdomain specifies a subdomain with the same underlying representation as its base domain.

> *Open issue*.
>
> An open semantic issue for subdomains is when a subdomain's subset property should be re-verified once its parent domain is reassigned and whether this should be done aggressively or lazily.

19.3.2 Simple Subdomain Values

> *Open issue*. Needs review: By default, the value of subdomain is an empty index set. A rectangular subdomain can be initialized using a domain literal (§19.2.1). Associative subdomains must be initialized explicitly, by using the `add` method or `+` operator.
>
> It is an error to attempt to add an index to a subdomain that is not also a member of the parent domain.

19.4 Sparse Subdomains

sparse-subdomain-type:
  sparse subdomain$_{opt}$ (*domain-expression*)

This declaration creates a sparse subdomain. *Sparse subdomains* are irregular domains that describe an arbitrary subset of a domain, even if the parent domain is a regular domain. Sparse subdomains are useful in Chapel for defining *sparse arrays* in which a single element value (usually “zero”) occurs frequently enough that it is worthwhile to avoid storing it redundantly. The set difference between a sparse subdomain's index set and that of parent domain is the set of indices for which the sparse array will store this replicated value. See §20.10 for details about sparse arrays.

19.4.1 Sparse Subdomain Types

Each root domain type has a unique corresponding sparse subdomain type. Sparse subdomains whose parent domains are also sparse subdomains share the same type.

19.4.2 Sparse Subdomain Values

A sparse subdomain's value is simply the set of all index values that the domain describes. If the parent domain's indices were ordered, the sparse subdomain's are as well.

There is no literal syntax for an sparse subdomain. However, a variable of a sparse subdomain type can be initilized using a tuple of values of the parent domain's index type.

The default value for a sparse subdomain value is the empty set. This is true even if the parent domain is an enumerated domain.

> *Example*. The following code declares a two-dimensional dense domain `D`, followed by a two dimensional sparse subdomain of `D` named `SpsD`. Since `SpsD` is uninitialized, it will initially describe an empty set of indices from `D`.
>
> ```
> const D: domain(2) = [1..n, 1..n];
> var SpsD: sparse subdomain(D);
> ```

19.5 Domain Index Types

Each domain value has a corresponding compiler-provided *index type* which can be used to represent values belonging to that domain's index set. Index types are described using the following syntax:

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

A variable with a given index type is constrained to take on only values available within the domain on which it is defined. This restriction allows the compiler to prove away the bound checking that code safety considerations might otherwise require. Due to the subset relationship between a base domain and its subdomains, a variable of an index type defined with respect to a subdomain is also necessarily a valid index into the base domain.

Since an index types are known to be legal for a given domain, it may also afford the opportunity to represent that index using an optimized format that doesn't simply store the index variable's value. This fact could be used to support accelerated access to arrays declared over that domain. For example, iteration over an index type could be implemented using memory pointers and strides, rather than explicitly calculating the offset of each index within the domain.

These potential optimizations may make it less expensive to index into arrays using index type variables of their domains or subdomains.

In addition, since an index type is associated with a specific domain or subdomain, it carries more semantic weight than a generic index. For example, one could iterate over a rectangular domain with integer bounds using an `int(n)` as the index variable. However, it would be more precise to use a variable of the domain's index type.

> *Open issue*.
>
> An open issue for index types is what the semantics should be for an index type value that is live across a modification to its domain's index set—particularly one that shrinks the index set. Our hypothesis is that most stored indices will either have short lifespans or belong to constant or monotonically growing domains. But these semantics need to be defined nevertheless.

19.6 Iteration Over Domains

All domains support iteration via standard `for`, `forall`, and `coforall` loops. These loops iterate over all of the indices that the domain describes. If the domain is ordered, then the indices are visited in the prescribed order.

The type of the iterator variable for an iteration over a domain named `D` is that domain's index type, `index(D)`. If the domain's indices are ordered, a `for` loop will traverse the indices in order.

19.7 Domains as Arguments

This section describes the semantics of passing domains as arguments to functions.

19.7.1 Formal Arguments of Domain Type

When a domain value is passed to a formal argument of compatible domain type by blank intent, it is passed by reference in order to preserve the domain's identity.

19.7.2 Domain Promotion of Scalar Functions

Domain values may be passed to a scalar function argument whose type matches the domain's index type. This results in a promotion of the scalar function as defined in §25.3.

> *Example*. Given a function `foo()` that accepts real floating point values and an associative domain `D` of type `domain(real)`, `foo` can be called with `D` as its actual argument which will result in the function being invoked for each value in the index set of `D`.

> *Example*. Given an array `A` with element type `int` declared over a one-dimensional domain `D` with `idxType int`, the array elements can be assigned their corresponding index values by writing:
>
> ```
> A = D;
> ```
>
> This is equivalent to:
>
> ```
> forall (a,i) in (A,D) do
>   a = i;
> ```

19.8 Domain Operations

Chapel supplies predefined operators and functions that can be used to manipulate domains. Unless otherwise noted, these operations are applicable to a domain of any type, whether a base domain or a subdomain.

19.8.1 Domain Assignment

All domain types support domain assignment.

```
domain-expression:
  domain-literal
  domain-name
  domain-assignment-expression
  domain-striding-expression
  domain-slice-expression

domain-assignment-expression:
  domain-name = domain-expression

domain-name:
  identifier
```

Domain assignment is by value and causes the target domain variable to take on the index set of the right-hand side expression. In practice, the right-hand side expression is often another domain value; a tuple of ranges (for regular domains); or a tuple of indices or a loop that enumerates indices (for irregular domains). If the domain variable being assigned was used to declare arrays, these arrays are reallocated as discussed in §20.11.

> *Example*. The following three assignments show ways of assigning indices to a sparse domain, `SpsD`. The first assigns the domain two index values, `(1,1)` and `(n,n)`. The second assigns the domain all of the indices along the diagonal from `(1,1)...(n,n)`. The third invokes an iterator that is written to `yield` indices read from a file named "inds.dat". Each of these assignments has the effect of replacing the previous index set with a completely new set of values.
>
> ```
> SpsD = ((1,1), (n,n));
> SpsD = [i in 1..n] (i,i);
> SpsD = readIndicesFromFile("inds.dat");
> ```

19.8.2 Domain Striding

The `by` operator can be applied to a rectangular domain value in order to create a strided rectangular domain value. The right-hand operand to the `by` operator can either be an integral value or an integral tuple whose size matches the domain's rank.

domain−striding−expression:
  domain−expression **by** *expression*

The type of the resulting domain is the same as the original domain but with `stridable` set to true. In the case of an integer stride value, the value of the resulting domain is computed by applying the integer value to each range in the value using the `by` operator. In the case of a tuple stride value, the resulting domain's value is computed by applying each tuple component to the corresponding range using the `by` operator.

19.8.3 Domain Slicing

In Chapel, slicing is the application of an index set to another variable using either parentheses or square brackets.

domain−slice−expression:
  domain−slice−expression [*domain−expression*]
  domain−slice−expression (*domain−expression*)

Domain-based Slicing

The index set used to express a slice can be represented using a domain value.

Slicing an array results in an alias to a subset of the array's elements as described in §20.6.

Slicing a domain evaluates to a new domain value whose index set is the intersection of the domain's index set and the slicing index set. The type and domain map of the result match the domain being sliced.

> *Open issue*. Can we say that it is an alias in the case of sparse/associative?

Range-based Slicing

When slicing rectangular domains or arrays, the slice can be expressed as a list of `rank` ranges. These ranges can either be bounded or unbounded. When unbounded, they inherit their bounds from the domain or array being sliced.

> *Example*. The following code declares a two dimensional rectangular domain `D`, and then a number of subdomains of `D` by slicing into `D` using bounded and unbounded ranges. The `InnerD` domain describes the inner indices of D, `Col2OfD` describes the 2nd column of `D`, and `AllButLastRow` describes all of `D` except for the last row.
>
> ```
> const D: domain(2) = [1..n, 1..n],
>       InnerD = D[2..n-1, 2..n-1],
>       Col2OfD = D[.., 2..2],
>       AllButLastRow = D[..n-1, ..];
> ```

Rank-Change Slicing

For multidimensional rectangular domains and arrays, substituting integral values for one or more of the ranges in a range-based slice will result in a domain or array of lower rank.

The result of a rank-change slice on an array is an alias to a subset of the array's elements as described in §20.6.1.

The result of rank-change slice on a domain is a subdomain of the domain being sliced. The resulting subdomain's type will be the same as the original domain, but with a `rank` equal to the number of dimensions that were sliced by ranges rather than integers.

19.8.4 Count Operator

The `#` operator can be applied to dense rectangular domains with a tuple argument whose size matches the rank of the domain (or optionally an integer in the case of a 1D domain). The operator is equivalent to applying the `#` operator to the component ranges of the domain and then using them to slice the domain as in Section 19.8.3.

19.8.5 Adding and Removing Domain Indices

All irregular domain types support the ability to incrementally add and remove indices from their index sets. This can either be done using `add(i:idxType)` and `remove(i:idxType)` methods on a domain variable or by using the `+=` and `-=` assignment operators. It is legal to add the same index to an irregular domain's index set twice, but illegal to remove an index that does not belong to the domain's index set.

> *Open issue*. These remove semantics seem dangerous in a parallel context; maybe add flags to both the method versions of the call that say whether they should balk or not? Or add exceptions...

As with normal domain assignments, arrays declared in terms of a domain being modified in this way will be reallocated as discussed in §20.11.

19.9 Predefined Methods on Domains

This section gives a brief description of the library functions provided for Domains. These are categorized by the type of domain to which they apply: all, regular or irregular. Within each subsection, entries are listed in alphabetical order.

19.9.1 Methods on All Domain Types

The methods in this subsection can be applied to any domain.

proc *Domain*`.clear()`

Resets this domain's index set to the empty set.

Example (emptyEnumeratedDomain). In the case of an enumerated domain, this function provides a way to produce an empty index set.

```
enum Counter { one, two, three };
var D : domain ( Counter );
writeln("D has ", D.numIndices, " indices.");
D.clear();
writeln("D has ", D.numIndices, " indices.");
```

D has 3 indices. D has 0 indices.

proc *Domain*`.idxType` **type**

Returns the domain type's `idxType`. This function is not available on opaque domains.

proc *Domain*`.indexOrder(i:` **index**`(`*Domain*`)): idxType`

If `i` is a member of the domain, returns the ordinal value of `i` using a total ordering of the domain's indices using 0-based indexing. Otherwise, it returns `(-1):idxType`. For rectangular domains, this ordering will be based on a row-major ordering of the indices; for other domains, the ordering may be implementation-defined and unstable as indices are added and removed from the domain.

proc `isEnumDom(d:` **domain**`)` **param**

Returns a param `true` if the given domain is enumerated, false otherwise.

proc `isIrregularDom(d:` **domain**`)` **param**

Returns a param `true` if the given domain is irregular, false otherwise.

proc `isOpaqueDom(d:` **domain**`)` **param**

Returns a param `true` if the given domain is opaque, false otherwise.

proc `isRectangularDom(d:` **domain**`)` **param**

Returns a param `true` if the given domain is rectangular, false otherwise.

proc `isSparseDom(d:` **domain**`)` **param**

Returns a param `true` if the given domain is sparse, false otherwise.

proc *Domain*`.member(i)`

Returns true if the given index `i` is a member of this domain's index set, and false otherwise.

Open issue. We would like to call the type of i above idxType, but it's not true for rectangular domains. That observation provides some motivation to normalize the behavior.

proc *Domain*`.numIndices: capType`

Returns the number of indices in the domain as a value of the capacity type.

19.9.2 Methods on Regular Domains

The methods described in this subsection can be applied to regular domains only.

proc *Domain*`.dim(d:` **int**`):` **range**

Returns the range of indices described by dimension `d` of the domain.

Example. In the code

```
for i in D.dim(1) do
  for j in D.dim(2) do
    writeln(A(i,j));
```

domain `D` is iterated over by two nested loops. The first dimension of `D` is iterated over in the outer loop. The second dimension is iterated over in the inner loop.

proc *Domain*`.dims(): rank*`**range**

Returns a tuple of ranges describing the dimensions of the domain.

proc *Domain*`.expand(off: integral):` **domain**
proc *Domain*`.expand(off: rank*integral):` **domain**

Returns a new domain that is the current domain expanded in dimension `d` if `off` or `off(d)` is positive or contracted in dimension `d` if `off` or `off(d)` is negative.

proc *Domain*`.exterior(off: integral):` **domain**
proc *Domain*`.exterior(off: rank*integral):` **domain**

Returns a new domain that is the exterior portion of the current domain with `off` or `off(d)` indices for each dimension `d`. If `off` or `off(d)` is negative, compute the exterior from the low bound of the dimension; if positive, compute the exterior from the high bound.

proc *Domain*`.high:` **index**`(`*Domain*`)`

Returns the high index of the domain as a value of the domain's index type.

```
proc Domain.interior(off: integral): domain
proc Domain.interior(off: rank*integral): domain
```

Returns a new domain that is the interior portion of the current domain with `off` or `off(d)` indices for each dimension `d`. If `off` or `off(d)` is negative, compute the interior from the low bound of the dimension; if positive, compute the interior from the high bound.

```
proc Domain.low: index(Domain)
```

Returns the low index of the domain as a value of the domain's index type.

```
proc Domain.rank param : int
```

Returns the rank of the domain.

```
proc Domain.stridable param : bool
```

Returns whether or not the domain is stridable.

```
proc Domain.stride: int(numBits(idxType)) where rank == 1
proc Domain.stride: rank*int(numBits(idxType))
```

Returns the stride of the domain as the domain's stride type (for 1D domains) or a tuple of the domain's stride type (for multidimensional domains).

```
proc Domain.translate(off: integral): domain
proc Domain.translate(off: rank*integral): domain
```

Returns a new domain that is the current domain translated by `off` or `off(d)` for each dimension `d`.

19.9.3 Methods on Irregular Domains

The following methods are available only on irregular domain types.

```
proc +(d: domain, i: index(d))
proc +(i, d: domain) where i: index(d)
```

Adds the given index to the given domain. If the given index is already a member of that domain, it is ignored.

```
proc +(d1: domain, d2: domain)
```

Merges the index sets of the two domain arguments.

```
proc -(d: domain, i: index(d))
```

Removes the given index from the given domain. It is an error if the domain does not contain the given index.

```
proc -(d1: domain, d2: domain)
```

Removes the indices in domain `d2` from those in `d1`. It is an error if `d2` contains indices which are not also in `d1`.

20 Arrays

An *array* is a map from a domain's indices to a collection of variables of homogenous type. Since Chapel domains support a rich variety of index sets, Chapel arrays are also richer than the traditional linear or rectilinear array types in conventional languages. Like domains, arrays may be distributed across multiple locales without explicitly partitioning them using Chapel's Domain Maps (§27).

20.1 Array Types

An array type is specified by the identity of the domain that it is declared over and the element type of the array. Array types are given by the following syntax:

array-type:
[*domain-expression*] *type-specifier*

The *domain-expression* must specify a domain that the array can be declared over. If it is a domain literal, the duplicate square brackets around the domain literal may be omitted.

Example (decls.chpl). In the code

```
const D: domain(2) = [1..10, 1..10];
var A: [D] real;
```

`A` is declared to be an arithmetic array over rectangular domain `D` with elements of type `real`. As a result, it represents a 2-dimensional 10×10 real floating point variables indexed using the indices $(1,1), (1,2), \ldots, (1,10), (2,1), \ldots, (10,10)$.

An array's element type can be referred to using the member symbol `eltType`.

Example (eltType.chpl). In the following example, `x` is declared to be of type `real` since that is the element type of array `A`.

```
var A: [D] real;
var x: A.eltType;
```

20.2 Array Values

An array's value is the collection of its elements' values. Assignments between array variables are performed by value as described in §20.5. Chapel semantics are defined so that the compiler will never need to insert temporary arrays of the same size as a user array variable.

Arrays do not have a literal format in Chapel, but arithmetic array variables can be initialized using tuple values. These tuples must match the size and shape of the array itself.

Example (decl-with-anon-domain.chpl). The following example declares a 2×3 array `A` using an anonymous domain value and initializes the elements of the array using a 2-tuple of 3-tuples which matches the array's size and shape.

```
var A: [1..2, 1..3] real = ((1.1, 1.2, 1.3), (2.1, 2.2, 2.3));
```

An array's default value is to have its elements all initialized to the default values for their types.

20.2.1 Runtime Representation of Array Values

The runtime representation of an array in memory is controlled by its domain's domain map. Through this mechanism, users can reason about and control the runtime representation of an array's elements. See §27 for more details.

20.3 Array Indexing

Arrays can be indexed using index values from the domain over which they are declared. Array indexing is expressed using either parenthesis or square brackets. This results in a reference to the element that corresponds to the index value.

Example (array-indexing.chpl). Given:

```
var A: [1..10] real;
```

the first two elements of A can be assigned the value 1.2 and 3.4 respectively using the assignment:

```
A(1) = 1.2;
A[2] = 3.4;
```

If an array is indexed using an index that is not part of its domain's index set, the reference is considered out-of-bounds and a runtime error will occur, halting the program.

20.3.1 Rectangular Array Indexing

Since the indices for multidimensional rectangular domains are tuples, for convenience, rectangular arrays can be indexed using the list of integer values that make up the tuple index. This is semantically equivalent to creating a tuple value out of the integer values and using that tuple value to index the array. For symmetry, 1-dimensional rectangular arrays can be accessed using 1-tuple indices even though their index type is an integral value. This is semantically equivalent to de-tupling the integral value from the 1-tuple and using it to index the array.

Example (array-indexing-2.chpl). Given:

```
var A: [1..5, 1..5] real;
var ij: 2*int = (1, 1);
```

the elements of array A can be indexed using any of the following idioms:

```
A(ij) = 1.1;
A((1, 2)) = 1.2;
A(1, 3) = 1.3;
A[ij] = -1.1;
A[(1, 4)] = 1.4;
A[1, 5] = 1.5;
```

Example (index-using-var-arg-tuple.chpl). The code

```
proc f(A: [], is...)
  return A(is);
```

defines a function that takes an array as the first argument and a variable-length argument list. It then indexes into the array using the tuple that captures the actual arguments. This function works even for one-dimensional arrays because one-dimensional arrays can be indexed into by 1-tuples.

20.4 Iteration over Arrays

All arrays support iteration via for, forall and coforall loops. These loops iterate over all of the array elements as described by its domain. A loop of the form:

```
[co]for[all] a in A
  ...a...
```

is semantically equivalent to:

```
[co]for[all] i in A.domain
  ...A(i)...
```

Thus, the iterator variable for an array traversal is a reference to the array element type.

20.5 Array Assignment

Array assignment is by value. Arrays can be assigned arrays, ranges, domains, iterators, or tuples.

Example (assign.chpl). If `A` is an lvalue of array type and `B` is an expression of either array, range, or domain type, or an iterator, then the assignment

```
A = B;
```

is equivalent to

```
forall (a,b) in (A,B) do
  a = b;
```

> If the zipper iteration is illegal, then the assignment is illegal. Notice that the assignment is implemented with the semantics of a `forall` loop.

Arrays can be assigned tuples of values of their element type if the tuple contains the same number of elements as the array. For multidimensional arrays, the tuple must be a nested tuple such that the nesting depth is equal to the rank of the array and the shape of this nested tuple must match the shape of the array. The values are assigned element-wise.

Arrays can also be assigned single values of their element type. In this case, each element in the array is assigned this value.

> *Example (assign-2.chpl).* If `e` is an expression of the element type of the array or a type that can be implicitly converted to the element type of the array, then the assignment
>
> ```
> A = e;
> ```
>
> is equivalent to
>
> ```
> forall a in A do
>   a = e;
> ```

20.6 Array Slicing

An array can be sliced using a domain that has the same type as the domain over which it was declared. The result of an array slice is an alias to the subset of the array elements from the original array corresponding to the slicing domain's index set.

> *Example (slicing.chpl).* Given the definitions
>
> ```
> var OuterD: domain(2) = [0..n+1, 0..n+1];
> var InnerD: domain(2) = [1..n, 1..n];
> var A, B: [OuterD] real;
> ```
>
> the assignment given by
>
> ```
> A[InnerD] = B[InnerD];
> ```
>
> assigns the elements in the interior of `B` to the elements in the interior of `A`.

20.6.1 Rectangular Array Slicing

A rectangular array can be sliced by any rectangular domain that is a subdomain of the array's defining domain. If the subdomain relationship is not met, an out-of-bounds error will occur. The result is a subarray whose indices are those of the slicing domain and whose elements are an alias of the original array's.

Rectangular arrays also support slicing by ranges directly. If each dimension is indexed by a range, this is equivalent to slicing the array by the rectangular domain defined by those ranges. These range-based slices may also be expressed using partially unbounded or completely unbounded ranges. This is equivalent to slicing the array's defining domain by the specified ranges to create a subdomain as described in §20.6 and then using that subdomain to slice the array.

20.6.2 Rectangular Array Slicing with a Rank Change

For multidimensional rectangular arrays, slicing with a rank change is supported by substituting integral values within a dimension's range for an actual range. The resulting array will have a rank less than the rectangular array's rank and equal to the number of ranges that are passed in to take the slice.

> *Example (array-decl.chpl).* Given an array
>
> ```
> var A: [1..n, 1..n] int;
> ```
>
> the slice `A[1..n, 1]` is a one-dimensional array whose elements are the first column of `A`.

20.7 Count Operator

The `#` operator can be applied to dense rectangular arrays with a tuple argument whose size matches the rank of the array (or optionally an integer in the case of a 1D array). The operator is equivalent to applying the `#` oeprator to the array's domain and using the result to slice the array as described in Section 20.6.1.

20.8 Array Arguments to Functions

Arrays are passed to functions by reference. Formal arguments that receive arrays are aliases of the actual arguments.

When a formal argument has array type, the element type of the array can be omitted and/or the domain of the array can be queried or omitted. In such cases, the argument is generic and is discussed in §22.1.6.

If a non-queried domain is specified in the array type of a formal argument, the domain must match the domain of the actual argument. This is verified at runtime. There is an exception if the domain is a rectangular domain, described in §20.8.1.

20.8.1 Formal Arguments of Rectangular Array Type

Formal arguments of rectangular array type allow a rectangular domain to be specified that does not match the rectangular domain of the actual rectangular 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 (actual-domain-strided.chpl).* In the code
>
> ```
> proc foo(X: [1..5] int) { X = c; }
> var A: [1..10 by 2] int;
> foo(A);
> ```
>
> the array `A` is strided and its elements can be indexed by the odd integers between one and nine. In the function `foo`, the array `X` references array `A` and the same elements can be indexed by the integers between one and five.

20.8.2 Array Promotion of Scalar Functions

Array promotion of a scalar function is defined over the array type and the element type of the array. The domain of the returned array, if an array is captured by the promotion, is the domain of the array that promoted the function. In the event of zipper promotion over multiple arrays, the promoted function returns an array with a domain that is equal to the domain of the first argument to the function that enables promotion. If the first argument is an iterator or a range, the result is a one-based one-dimensional array.

> *Example (whole-array-ops.chpl).* Whole array operations is a special case of array promotion of scalar functions. In the code
>
> ```
> A = B + C;
> ```
>
> if `A`, `B`, and `C` are arrays, this code assigns each element in `A` the element-wise sum of the elements in `B` and `C`.

20.9 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:
  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 (reindexing.chpl).* In the code
>
> ```
> var A: [1..5, 1..5] int;
> var AA: [0..2, 0..2] => A[2..4, 2..4];
> ```
>
> an array alias `AA` is created to alias the interior of array `A` given by the slice `A[2..4, 2..4]`. The reindexing expression changes the indices defined by the domain of the alias to be zero-based in both dimensions. Thus `AA(1,1)` is equivalent to `A(3,3)`.

20.10 Sparse Arrays

Sparse arrays in Chapel are those whose domain is a sparse array. A sparse array differs from other array types in that it stores a single value corresponding to multiple indices. This value is commonly referred to as the *zero value*, but we refer to it as the *implicitly replicated value* or *IRV* since it can take on any value of the array's element type in practice including non-zero numeric values, a class reference, a record or tuple value, etc.

An array declared over a sparse domain can be indexed using any of the indices in the sparse domain's parent domain. If it is read using an index that is not part of the sparse domain's index set, the IRV value is returned. Otherwise, the array element corresponding to the index is returned.

Sparse arrays can only be written at locations corresponding to indices in their domain's index set. In general, writing to other locations corresponding to the IRV value will result in a runtime error.

By default a sparse array's IRV is defined as the default value for the array's element type. The IRV can be set to any value of the array's element type by assigning to a pseudo-field named `IRV` in the array.

> *Example (sparse-error.chpl).* The following code example declares a sparse array, `SpsA` using the sparse domain `SpsD` (For this example, assume that `n`>1). Line 2 assigns two indices to `SpsD`'s index set and then lines 3–4 store the values 1.1 and 9.9 to the corresponding values of `SpsA`. The IRV of `SpsA` will initially be 0.0 since its element type is `real`. However, the fifth line sets the IRV to be the value 5.5, causing `SpsA` to represent the value 1.1 in its low corner, 9.9 in its high corner, and 5.5 everywhere else. The final statement is an error since it attempts to assign to `SpsA` at an index not described by its domain, `SpsD`.

```
var SpsD: sparse subdomain(D);
var SpsA: [SpsD] real;
SpsD = ((1,1), (n,n));
SpsA(1,1) = 1.1;
SpsA(n,n) = 9.9;
SpsA.IRV = 5.5;
SpsA(1,n) = 0.0;  // ERROR!
```

20.11 Association of Arrays to Domains

When an array is declared, it is linked during execution to the domain identity over which it was declared. This linkage is invariant for the array's lifetime and cannot be changed.

When indices are added or removed from a domain, the change impacts the arrays declared over this particular domain. In the case of adding an index, an element is added to the array and initialized to the IRV for sparse arrays, and to the default value for the element type for dense arrays. In the case of removing an index, the element in the array is removed.

When a domain is reassigned a new value, its arrays are also impacted. Values that correspond to indices in the intersection of the old and new domain are preserved in the arrays. Values that could only be indexed by the old domain are lost. Values that can only be indexed by the new domain have elements added to the new array, initialized to the IRV for sparse arrays, and to the element type's default value for other array types.

For performance reasons, there is an expectation that a method will be added to domains to allow non-preserving assignment, *i.e.*, all values in the arrays associated with the assigned domain will be lost. Today this can be achieved by assigning the array's domain an empty index set (causing all array elements to be deallocated) and then re-assigning the new index set to the domain.

An array's domain can only be modified directly, via the domain's name or an alias created by passing it to a function via blank intent. In particular, the domain may not be modified via the array's `.domain` method, nor by using the domain query syntax on a function's formal array argument (§22.1.6).

> *Rationale*. When multiple arrays are declared using a single domain, modifying the domain affects all of the arrays. Allowing an array's domain to be queried and then modified suggests that the change should only affect that array. By requiring the domain to be modified directly, the user is encouraged to think in terms of the domain distinctly from a particular array.
>
> In addition, this choice has the beneficial effect that arrays declared via an anonymous domain have a constant domain. Constant domains are considered a common case and have potential compilation benefits such as eliminating bounds checks. Therefore making this convenient syntax support a common, optimizable case seems prudent.

20.12 Predefined Functions and Methods on Arrays

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

proc *Array*.`eltType` **type**

> Returns the element type of the array.

proc *Array*.`rank` **param**

> Returns the rank of the array.

proc *Array*.**domain**: `this`.**domain**

> Returns the domain of the given array. This domain is constant, implying that the domain cannot be resized by assigning to its domain field, only by modifying the domain directly.

proc *Array*.`numElements: this.`**domain**`.dim_type`

> Returns the number of elements in the array.

proc `reshape(A:` *Array*`, D:` *Domain*`):` *Array*

> Returns a copy of the array containing the same values but in the shape of the new domain. The number of indices in the domain must equal the number of elements in the array. The elements of the array are copied into the new array using the default iteration orders over both arrays.

21 Iterators

An *iterator* is a function that can generate, or *yield*, multiple values (consecutively or in parallel) via its yield statements.

> *Open issue*. The parallel iterator story is under development. It is expected that the specification will be expanded regarding parallel iterators soon.

21.1 Iterator Definitions

The syntax to declare an iterator is given by:

iterator-declaration-statement:
  iter *iterator-name argument-list*$_{opt}$ *return-intent*$_{opt}$ *return-type*$_{opt}$ *where-clause*$_{opt}$
  iterator-body

iterator-name:
  identifier

iterator-body:
  block-statement
  yield-statement

The syntax of an iterator declaration is similar to a procedure declaration, with some key differences:

- The keyword `iter` is used instead of the keyword `proc`.
- The name of the iterator cannot overload any operator.
- `yield` statements may appear in the body of an iterator, but not in a procedure.
- A `return` statement in the body of an iterator is not allowed to have an expression.

21.2 The Yield Statement

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

yield-statement:
  yield *expression* ;

When an iterator is executed and a `yield` is encountered, the value of the yield expression is returned. However, the state of execution of the iterator is saved. On its next invocation, execution resumes from the point immediately following that `yield` statement and with the saved state of execution.

When a `return` is encountered, the iterator finishes without yielding another index value. The `return` statements appearing in an iterator are not permitted to have a return expression. An iterator also completes after the last statement in the iterator is executed. An iterator need not contain any yield statements.

21.3 Iterator Calls

Iterators are invoked using regular call expressions:

> *iteratable−call−expression*:
>   *call−expression*

All details of iterator calls, including argument passing, function resolution, the use of parentheses versus brackets to delimit the parameter list, and so on, are identical to procedure calls as described in Chapter 13.

However, the result of an iterator call depends upon its context, as described below.

21.3.1 Iterators in For and Forall Loops

When an iterator is accessed via a for or forall loop, the iterator is evaluated alongside the loop body in an interleaved manner. For each iteration, the iterator yields a value and the body is executed.

21.3.2 Iterators as Arrays

If an iterator function is captured into a new variable declaration or assigned to an array, the iterator is iterated over in total and the expression evaluates to a one-dimensional arithmetic array that contains the values returned by the iterator on each iteration.

Example (as-arrays.chpl). Given an iterator

```
iter squares(n: int): int {
  for i in 1..n do
    yield i * i;
}
```

1 4 9 16 25

the expression `squares(5)` evaluates to the array `1, 4, 9, 16, 25`.

21.3.3 Iterators and Generics

An iterator call expression can be passed to a generic function argument that has neither a declared type nor default value (§22.1.3). In this case the iterator is passed without being evaluated. Within the generic function the corresponding formal argument can be used as an iterator, e.g. in for loops. The arguments to the iterator call expression, if any, are evaluated at the call site, i.e. prior to passing the iterator to the generic function.

21.3.4 Recursive Iterators

Recursive iterators are allowed. A recursive iterator invocation is typically made by iterating over it in a loop.

> *Example (recursive.chpl).* A post-order traversal of a tree data structure could be written like this:
>
> ```
> iter postorder(tree: Tree): string {
>   if tree != nil {
>     for child in postorder(tree.left) do
>       yield child;
>     for child in postorder(tree.right) do
>       yield child;
>     yield tree.data;
>   }
> }
> ```
>
> By contrast, using calls `postorder(tree.left)` and `postorder(tree.right)` as stand-alone statements would result in generating temporary arrays containing the outcomes of these recursive calls, which would then be discarded.

21.4 Parallel Iterators

Iterators used in explicit forall-statements or -expressions must be parallel iterators. Reductions, scans and promotion over serial iterators will be serialized.

Parallel iterators are defined for standard constructs in Chapel such as ranges, domains, and arrays, thereby allowing these constructs to be used with forall-statements and -expressions.

The left-most iteratable expression in a forall-statement or -expression determines the number of tasks, the iterations each task executes, and the locales on which these tasks execute. For ranges, default domains, and default arrays, these values can be controlled via configuration constants (§25.5).

Domains and arrays defined using distributed domain maps will typically implement forall loops with multiple tasks on multiple locales. For ranges, default domains, and default arrays, all tasks are executed on the current locale.

A more detailed definition of parallel iterators is forthcoming. The following paper sketches out the parallel iterator interface:

> B. L. Chamberlain, S. Choi, S. J. Deitz, and A. Navarro. **Composable Parallel Iterators in Chapel**. 2011.

22 Generics

Chapel supports generic functions and types that are parameterizable over both types and parameters. The generic functions and types look similar to non-generic functions and types already discussed.

22.1 Generic Functions

A function is generic if any of the following conditions hold:

- Some formal argument is specified with an intent of `type` or `param`.
- Some formal argument has no specified type and no default value.
- Some formal argument is specified with a queried type.
- The type of some formal argument is a generic type, e.g., `List`. Queries may be inlined in generic types, e.g., `List(?eltType)`.
- The type of some formal argument is an array type where either the element type is queried or omitted or the domain is queried or omitted.

These conditions are discussed in the next sections.

22.1.1 Formal Type Arguments

If a formal argument is specified with intent `type`, then a type must be passed to the function at the call site. A copy of the function is instantiated for each unique type that is passed to this function at a call site. The formal argument has the semantics of a type alias.

Example (build2tuple.chpl). 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.

```
proc build2Tuple(type t, type tt) {
  var x1: t;
  var x2: tt;
  return (x1, x2);
}
```

This function is instantiated with "normal" function call syntax where the arguments are types:

```
var t2 = build2Tuple(int, string);
t2 = (1, "hello");
```

22.1.2 Formal Parameter Arguments

If a formal argument is specified with intent `param`, then a parameter must be passed to the function at the call site. A copy of the function is instantiated for each unique parameter that is passed to this function at a call site. The formal argument is a parameter.

> *Example (fillTuple.chpl).* 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`.

```
proc fillTuple(param p: int, x: int) {
  var result: p*int;
  for param i in 1..p do
    result(i) = x;
  return result;
}
```

> The function call `fillTuple(3, 3)` returns a 3-tuple where each component contains the value `3`.

22.1.3 Formal Arguments without Types

If the type of a formal argument is omitted, the type of the formal argument is taken to be the type of the actual argument passed to the function at the call site. A copy of the function is instantiated for each unique actual type.

> *Example (fillTuple2.chpl).* 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`:

```
proc fillTuple(param p: int, x) {
  var result: p*x.type;
  for param i in 1..p do
    result(i) = x;
  return result;
}
```

> In this function, the type of the tuple is taken to be the type of the actual argument. The call `fillTuple(3, 3.14)` returns a 3-tuple of real values `(3.14, 3.14, 3.14)`. The return type is `(real, real, real)`.

22.1.4 Formal Arguments with Queried Types

If the type of a formal argument is specified as a queried type, the type of the formal argument is taken to be the type of the actual argument passed to the function at the call site. A copy of the function is instantiated for each unique actual type. The queried type has the semantics of a type alias.

> *Example (fillTuple3.chpl).* The example from the previous section can be rewritten to use a queried type for clarity:

```
proc fillTuple(param p: int, x: ?t) {
  var result: p*t;
  for param i in 1..p do
    result(i) = x;
  return result;
}
```

22.1.5 Formal Arguments of Generic Type

If the type of a formal argument is a generic type, the type of the formal argument is taken to be the type of the actual argument passed to the function at the call site with the constraint that the type of the actual argument is an instantiation of the generic type. A copy of the function is instantiated for each unique actual type.

> *Example*. The following code defines a function `writeTop` that takes an actual argument that is a generic stack (see §22.6) and outputs the top element of the stack. The function is generic on the type of its argument.
>
> ```
> proc writeTop(s: Stack) {
>   write(s.top.item);
> }
> ```

Types and parameters may be queried from the top-level types of formal arguments as well. In the example above, the formal argument's type could also be specified as `Stack(?type)` in which case the symbol `type` is equivalent to `s.itemType`.

Note that generic types which have default values for all of their generic fields, *e.g. range*, are not generic when simply specified and require a query to mark the argument as generic. For simplicity, the identifier may be omitted.

> *Example*. The following code defines a class with a type field that has a default value. Function `f` is defined to take an argument of this class type where the type field is instantiated to the default. Function `g`, on the other hand, is generic on its argument because of the use of the question mark.
>
> ```
> class C {
>   type t = int;
> }
> proc f(c: C) {
>   // c.type is always int
> }
> proc g(c: C(?)) {
>   // c.type may not be int
> }
> ```

The generic type may be specified with some queries and some exact values. These exact values result in *implicit where clauses* for the purpose of function resolution.

> *Example*. Given the class definition

```
class C {
  type t;
  type tt;
}
```

then the function definition

```
proc f(c: C(?t,real)) {
  // body
}
```

is equivalent to

```
proc f(c: C(?t,?tt)) where tt == real {
  // body
}
```

For tuples with query arguments, an implicit where clause is always created to ensure that the size of the actual tuple matches the implicitly specified size of the formal tuple.

Example. The function definition

```
proc f(tuple: (?t,real)) {
  // body
}
```

is equivalent to

```
proc f(tuple: (?t,?tt)) where tuple.size == 2 && tt == real {
  // body
}
```

Example (query.chpl). Type queries can also be used to constrain the types of other function arguments and/or the return type. In this example, the type query on the first argument establishes type constraints on the other arguments and also determines the return type.

```
writeln(sumOfThree(1,2,3));
writeln(sumOfThree(4.0,5.0,3.0));

proc sumOfThree(x: ?t, y:t, z:t):t {
   var sum: t;

   sum = x + y + z;
   return sum;
}
```

6 12.0

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

22.1.6 Formal Arguments of Generic Array Types

If the type of a formal argument is an array where either the domain or the element type is queried or omitted, the type of the formal argument is taken to be the type of the actual argument passed to the function at the call site. If the domain is omitted, the domain of the formal argument is taken to be the domain of the actual argument.

A queried domain may not be modified via the name to which it is bound (see §20.11 for rationale).

22.2 Function Visibility in Generic Functions

Function visibility in generic functions is altered depending on the instantiation. When resolving function calls made within generic functions, the visible functions are taken from any call site at which the generic function is instantiated for each particular instantiation. The specific call site chosen is arbitrary and it is referred to as the *point of instantiation*.

For function calls that specify the module explicitly (§12.4.1), an implicit use of the specified module exists at the call site.

> *Example (point-of-instantiation.chpl).* Consider the following code which defines a generic function `bar`:
>
> ```
> module M1 {
>   record R {
>     var x: int;
>     proc foo() { }
>   }
> }
>
> module M2 {
>   proc bar(x) {
>     x.foo();
>   }
> }
>
> module M3 {
>   use M1, M2;
>   proc main() {
>     var r: R;
>     bar(r);
>   }
> }
> ```
>
> In the function `main`, the variable `r` is declared to be of type `R` defined in module `M1` and a call is made to the generic function `bar` which is defined in module `M2`. This is the only place where `bar` is called in this program and so it becomes the point of instantiation for `bar` when the argument `x` is of type `R`. Therefore, the call to the `foo` method in `bar` is resolved by looking for visible functions from within `main` and going through the use of module `M1`.

If the generic function is only called indirectly through dynamic dispatch, the point of instantiation is defined as the point at which the derived type (the type of the implicit `this` argument) is defined or instantiated (if the derived type is generic).

> *Rationale*. Visible function lookup in Chapel's generic functions is handled differently than in C++'s template functions in that there is no split between dependent and independent types.
>
> Also, dynamic dispatch and instantiation is handled differently. Chapel supports dynamic dispatch over methods that are generic in some of its formal arguments.
>
> Note that the Chapel lookup mechanism is still under development and discussion. Comments or questions are appreciated.

22.3 Generic Types

Generic types are generic classes and generic records. A class or record is generic if it contains one or more generic fields. A generic field is one of:

- a specified or unspecified type alias,
- a parameter field, or
- a `var` or `const` field that has no type and no initialization expression.

For each generic field, the class or record is parameterized over:

- the type bound to the type alias,
- the value of the parameter field, or
- the type of the `var` or `const` field, respectively.

Correspondingly, the class or record is instantiated with a set of types and parameter values, one type or value per generic field.

22.3.1 Type Aliases in Generic Types

If a class or record defines a type alias, the class or record is generic over the type that is bound to that alias. Such a type alias is accessed as if it were a field; similar to a parameter field, it cannot be assigned except in its declaration.

The type alias becomes an argument with intent `type` to the compiler-generated constructor (§22.3.6) for its class or record. This makes the compiler-generated constructor generic. The type alias also becomes an argument with intent `type` to the type constructor (§22.3.4). If the type alias declaration binds it to a type, that type becomes the default for these arguments, otherwise they have no defaults.

The class or record is instantiated by binding the type alias to the actual type passed to the corresponding argument of a user-defined (§22.3.7) or compiler-generated constructor or type constructor. If that argument has a default, the actual type can be omitted, in which case the default will be used instead.

Example (NodeClass.chpl). The following code defines a class called `Node` that implements a linked list data structure. It is generic over the type of the element contained in the linked list.

```
class Node {
  type eltType;
  var data: eltType;
  var next: Node(eltType);
}
```

The call `new Node(real, 3.14)` creates a node in the linked list that contains the value `3.14`. The `next` field is set to nil. The type specifier `Node` is a generic type and cannot be used to define a variable. The type specifier `Node(real)` denotes the type of the `Node` class instantiated over `real`. Note that the type of the `next` field is specified as `Node(eltType)`; the type of `next` is the same type as the type of the object that it is a field of.

22.3.2 Parameters in Generic Types

If a class or record defines a parameter field, the class or record is generic over the value that is bound to that field. The parameter becomes an argument with intent `param` to the compiler-generated constructor (§22.3.6) for that class or record. This makes the compiler-generated constructor generic. The parameter also becomes an argument with intent `param` to the type constructor (§22.3.4). If the parameter declaration has an initialization expression, that expression becomes the default for these arguments, otherwise they have no defaults.

The class or record is instantiated by binding the parameter to the actual value passed to the corresponding argument of a user-defined (§22.3.7) or compiler-generated constructor or type constructor. If that argument has a default, the actual value can be omitted, in which case the default will be used instead.

Example (IntegerTuple.chpl). The following code defines a class called `IntegerTuple` that is generic over an integer parameter which defines the number of components in the class.

```
class IntegerTuple {
  param size: int;
  var data: size*int;
}
```

The call `new IntegerTuple(3)` creates an instance of the `IntegerTuple` class that is instantiated over parameter `3`. The field `data` becomes a 3-tuple of integers. The type of this class instance is `IntegerTuple(3)`. The type specified by `IntegerTuple` is a generic type.

22.3.3 Fields without Types

If a `var` or `const` field in a class or record has no specified type or initialization expression, the class or record is generic over the type of that field. The field becomes an argument with blank intent to the compiler-generated constructor (§22.3.6). That argument has no specified type and no default value. This makes the compiler-generated constructor generic. The field also becomes an argument with `type` intent and no default to the type constructor (§22.3.4). Correspondingly, an actual value must always be passed to the default constructor argument and an actual type to the type constructor argument.

The class or record is instantiated by binding the type of the field to the type of the value passed to the corresonding argument of a user-defined (§22.3.7) or compiler-generated constructor (§22.3.6). When the type constructor is invoked, the class or record is instantiated by binding the type of the field to the actual type passed to the corresponding argument.

> *Example (fieldWithoutType.chpl).* The following code defines another class called `Node` that implements a linked list data structure. It is generic over the type of the element contained in the linked list. This code does not specify the element type directly in the class as a type alias but rather omits the type from the `data` field.
>
> ```
> class Node {
>   var data;
>   var next: Node(data.type) = nil;
> }
> ```
>
> A node with integer element type can be defined in the call to the constructor. The call `new Node(1)` defines a node with the value `1`. The code
>
> ```
> var list = new Node(1);
> list.next = new Node(2);
> ```
>
> defines a two-element list with nodes containing the values `1` and `2`. The type of each object could be specified as `Node(int)`.

22.3.4 The Type Constructor

A type constructor is automatically created for each class or record. A type constructor is a type function (§13.9) that has the same name as the class or record. It takes one argument per the class's or record's generic field, including fields inherited from the superclasses, if any. The formal argument has intent `type` for a type alias field and is a parameter for a parameter field. It accepts the type to be bound to the type alias and the value to be bound to the parameter, respectively. For a generic `var` or `const` field, the corresponding formal argument also has intent `type`. It accepts the type of the field, as opposed to a value as is the case for a parameter field. The formal arguments occur in the same order as the fields are declared and have the same names as the corresponding fields. Unlike the compiler-generated constructor, the type constructor has only those arguments that correspond to generic fields.

A call to a type constructor accepts actual types and parameter values and returns the type of the class or record that is instantiated appropriately for each field (§22.3.1, §22.3.2, §22.3.3). Such an instantiated type must be used as the type of a variable, array element, non-generic formal argument, and in other cases where uninstantiated generic class or record types are not allowed.

When a generic field declaration has an initialization expression or a type alias is specified, that initializer becomes the default value for the corresponding type constructor argument. Uninitialized fields, including all generic `var` and `const` fields, and unspecified type aliases result in arguments with no defaults; actual types or values for these arguments must always be provided when invoking the type constructor.

22.3.5 Generic Methods

All methods bound to generic classes or records, including constructors, are generic over the implicit `this` argument. This is in addition to being generic over any other argument that is generic.

22.3.6 The Compiler-Generated Constructor

If no user-defined constructors are supplied for a given generic class, the compiler generates one following in a manner similar to that for concrete classes (§15.3.2). However, the compiler-generated constructor for a generic class or record (§15.3.2) is generic over each argument that corresponds to a generic field, as specified above. The argument has intent `type` for a type alias field and is a parameter for a parameter field. It accepts the type to be bound to the type alias and the value to be bound to the parameter, respectively. This is the same as for the type constructor. For a generic `var` or `const` field, the corresponding formal argument has the blank intent and accepts the value for the field to be initialized with. The type of the field is inferred automatically to be the type of the initialization value.

The default values for the generic arguments of the compiler-generated constructor are the same as for the type constructor (§22.3.4). For example, the arguments corresponding to the generic `var` and `const` fields, if any, never have defaults, so the corresponding actual values must always be provided.

22.3.7 User-Defined Constructors

If a generic field of a class does not have an initialization expression or a type alias is unspecified, each user-defined constructor for that class must provide a formal argument whose name matches the name of the field.

If the name of a formal argument in a user-defined constructor matches the name of a generic field that does not have an initialization expression, is a type alias, or is a parameter field, the field is automatically initialized at the beginning of the constructor invocation to the actual value of that argument. This is done by passing that formal argument to the implicit invocation of the compiler-generated constructor during default-initialization (§15.2.8).

Example (constructorsForGenericFields.chpl). In the following code:

```
class MyGenericClass {
  type t1;
  param p1;
  const c1;
  var v1;
  var x1: t1; // this field is not generic

  type t5 = real;
  param p5 = "a string";
  const c5 = 5.5;
  var v5 = 555;
  var x5: t5; // this field is not generic

  proc MyGenericClass(c1, v1, type t1, param p1) { }
  proc MyGenericClass(type t5, param p5, c5, v5, x5,
                      type t1, param p1, c1, v1, x1) { }
}  // class MyGenericClass

var g1 = new MyGenericClass(11, 111, int, 1);
var g2 = new MyGenericClass(int, "this is g2", 3.3, 333, 3333,
                            real, 2, 222, 222.2, 22);
```

The arguments `t1`, `p1`, `c1`, and `v1` are required in all constructors for `MyGenericClass`. They can appear in any order. Both `MyGenericClass` constructors initialize the corresponding fields implicitly because these fields do not have initialization expressions. The second constructor also initializes implicitly the fields `t5` and `p5` because they are a type field and a parameter field. It does not initialize the fields `c5` and `v5` because they have initialization expressions, or the fields `x1` and `x5` because they are not generic fields (even though they are of generic types).

Open issue. The design of constructors, especially for generic classes, is under development, so the above specification may change.

22.4 Where Expressions

The instantiation of a generic function can be constrained by *where clauses*. A where clause is specified in the definition of a function (§13.2). When a function is instantiated, the expression in the where clause must be a parameter expression and must evaluate to either `true` or `false`. If it evaluates to `false`, the instantiation is rejected and the function is not a possible candidate for function resolution. Otherwise, the function is instantiated.

Example (whereClause.chpl). Given two overloaded function definitions

```
proc foo(x) where x.type == int { writeln("int"); }
proc foo(x) where x.type == real { writeln("real"); }
```

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

22.5 User-Defined Compiler Diagnostics

The special compiler diagnostic function calls `compilerError` and `compilerWarning` generate compiler diagnostic of the indicated severity if the function containing these calls may be called when the program is executed and the function call is not eliminated by parameter folding.

The compiler diagnostic is defined by the actual arguments which must be string parameters. The diagnostic points to the spot in the Chapel program from which the function containing the call is called. Compilation halts if a `compilerError` is encountered whereas it will continue after encountering a `compilerWarning`.

Note that when a variable function is called in a context where the implicit `setter` argument is true or false, both versions of the variable function are resolved by the compiler. Consequently, the `setter` argument cannot be effectively used to guard a compiler diagnostic statements.

Example (compilerDiagnostics.chpl). The following code shows an example of using user-defined compiler diagnostics to generate warnings and errors:

```
proc foo(x, y) {
  if (x.type != y.type) then
    compilerError("foo() called with non-matching types: ",
                  typeToString(x.type), " != ", typeToString(y.type));
  writeln("In 2-argument foo...");
}
```

```
proc foo(x) {
  compilerWarning("1-argument version of foo called");
  writeln("In generic foo!");
}
```

The first routine generates a compiler error whenever the compiler encounters a call to it where the two arguments have different types. It prints out an error message indicating the types of the arguments. The second routine generates a compiler warning whenver the compiler encounters a call to it.

Thus, if the program foo.chpl contained the following calls:

```
1 foo(3.4);
2 foo("hi");
3 foo(1, 2);
4 foo(1.2, 3.4);
5 foo("hi", "bye");
6 foo(1, 2.3);
7 foo("hi", 2.3);
```

compiling the program would generate output like:

```
foo.chpl:1: warning: 1-argument version of foo called with type: real
foo.chpl:2: warning: 1-argument version of foo called with type: string
foo.chpl:6: error: foo() called with non-matching types: int != real
```

22.6 Example: A Generic Stack

Example (genericStack.chpl).

```
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

  proc push(item: itemType) {
    top = new MyNode(itemType, item, top);
  }

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

  proc isEmpty return top == nil;
}
```

23 Input and Output

Chapel provides a predefined `file` class to handle input and output to files using the functions `read`, `readln`, `write`, and `writeln`.

23.1 The *file* type

The file class contains the following fields:

- The `filename` field is a `string` that contains the name of the file.
- The `mode` field is a `FileAccessMode` enum value that indicates whether the file is being read or written.
- The `path` field is a `string` that contains the path of the file.
- The `style` field can be set to `text` or `binary` to specify that reading from or writing to the file should be done with text or binary formats.

These fields can be modified any time that the file is closed.

The `mode` field supports the following `FileAccessMode` values:

- `FileAccessMode.read` The file can be read.
- `FileAccessMode.write` The file can be written.

The file type supports the following methods:

- The `open()` method opens the file for reading and/or writing.
- The `close()` method closes the file for reading and/or writing.
- The `isOpen` method returns true if the file is open for reading and/or writing, and otherwise returns false.
- The `eof` method returns true if the file is at its end-of-file position and returns false otherwise.
- The `flush()` method flushes the file, finishing outstanding reading and writing.

Additionally, the file type supports the methods `read`, `readln`, `write`, and `writeln` for input and output as discussed in §23.5 and §23.6.

23.2 Standard files *stdout*, *stdin*, and *stderr*

The files `stdout`, `stdin`, and `stderr` are predefined and map to standard output, standard input, and standard error as implemented in a platform dependent fashion.

23.3 The *write*, *writeln*, *read*, and *readln* functions

The predefined function `write` takes an arbitrary number of arguments and prints each out in turn to `stdout`. The predefined function `writeln` is identical to `write` except that it outputs an additional *end-of-line* character after writing out the argument expressions. Both of these functions will generate their output atomically with respect to other calls to these functions from other tasks.

The predefined function `read` takes an arbitrary number of variable expressions and reads into each in turn from `stdin`. Any whitespace is skipped over and is used only to separate one argument from the next. The predefined function `readln` is identical except that upon reading all of its arguments it scans ahead in the input stream until just after the next *end-of-line* character.

The `read` and `readln` functions are also defined to take an arbitrary number of types as arguments. In this case, the functions read an expression of each argument type. In the event that a single type is specified, the return value is the value that was read; if multiple types are specified, a tuple of the values is returned.

These functions are wrappers for the methods on files described next.

> *Example (writeln.chpl).* The `writeln` wrapper function allows for a simple implementation of the *Hello-World* program:
>
> ```
> writeln("Hello, World!");
> ```
>
> Hello, World!

> *Example.* The following code shows three ways to read values into a pair of variables `x` and `y`:
>
> ```
> var x: int;
> var y: real;
>
> /* reading into variable expressions */
> read(x, y);
>
> /* reading via a single type argument */
> x = read(int);
> y = read(real);
>
> /* reading via multiple type arguments */
> (x, y) = read(int, real);
> ```

23.4 User-Defined *writeThis* methods

To define the output for a given type, the user must define a method called `writeThis` on that type that takes a single argument of `Writer` type. If such a method does not exist, a default method is created.

23.5 The *write* and *writeln* method on files

The `file` type supports methods `write` and `writeln` for output. These methods are defined to take an arbitrary number of arguments. Each argument is written in turn by calling the `writeThis` method on that argument. Default `writeThis` methods are bound to any type that the user does not explicitly create one for.

A lock is used to ensure that output is serialized across multiple tasks.

23.5.1 The *write* and *writeln* method on strings

The `write` and `writeln` methods can also be called on strings to write the output to a string instead of a file.

23.5.2 Generalized *write* and *writeln*

The `Writer` class contains no arguments and serves as a base class to allow user-defined classes to be written to. If a class is defined to be a subclass of Writer, it must override the `writeIt` method that takes a `string` as an argument.

Example (Writer.chpl). 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;
  proc writeIt(s: string) {
    data += s.substring(1);
  }
  proc writeThis(x: Writer) {
    x.write(data);
  }
}

var c = new C();
c.write(41, 32, 23, 14);
writeln(c);
```

```
4321
```

The `C` class filters the arguments sent to it, printing out only the first letter. The output to the above is thus `4321`.

23.6 The *read* and *readln* methods on files

The `file` type supports `read` and `readln` methods. The `read` method takes an arbitrary number of arguments, reading in each argument from file. The `readln` method also takes an arbitrary number of arguments, reading in each argument from a single line or multiple lines in the file and advancing the file pointer to the next line after the last argument is read.

The `file` type also supports overloaded methods `read` and `readln` that take an arbitrary number of types as arguments. These methods read values of the specified types from the file and return them in a tuple. If only one type is read, the value is not returned in a tuple, but is returned directly.

> *Example*. The following line of code reads a value of type `int` from `stdin` and uses it to initialize variable `x` (causing `x` to have an inferred type of `int`):
>
> ```
> var x = stdin.read(int);
> ```

23.7 Default *read* and *write* methods

Default `write` methods are created for all types for which a user `write` method is not defined. They have the following semantics:

- **arrays** Outputs the elements of the array in row-major order where rows are separated by line-feeds and blank lines are used to separate other dimensions.
- **domains** Outputs the dimensions of the domain enclosed by `[` and `]`.
- **ranges** Outputs the lower bound of the range followed by `..` followed by the upper bound of the range. If the stride of the range is not one, the output is additionally followed by the word `by` followed by the stride of the range.
- **tuples** Outputs the components of the tuple in order delimited by `(` and `)`, and separated by commas.
- **classes** Outputs the values within the fields of the class prefixed by the name of the field and the character `=`. Each field is separated by a comma. The output is delimited by `{` and `}`.
- **records** Outputs the values within the fields of the class prefixed by the name of the field and the character `=`. Each field is separated by a comma. The output is delimited by `(` and `)`.

Default `read` methods are created for all types for which a user `read` method is not defined. The default `read` methods are defined to read in the output of the default `write` method.

24 Task Parallelism and Synchronization

Chapel supports both task parallelism and data parallelism. This chapter details task parallelism as follows:

- §24.1 introduces tasks and task parallelism.
- §24.2 describes the begin statement, an unstructured way to introduce concurrency into a program.
- §24.3 describes synchronization variables, an unstructured mechanism for synchronizing tasks.
- §24.4 describes the cobegin statement, a structured way to introduce concurrency into a program.
- §24.5 describes the coforall loop, another structured way to introduce concurrency into a program.
- §24.6 describes the sync statement, a structured way to control parallelism.
- §24.7 describes the serial statement, a structured way to suppress parallelism.
- §24.8 describes the atomic statement, a construct to support atomic transactions.

24.1 Tasks and Task Parallelism

A Chapel *task* is a distinct context of execution that maybe running concurrently with other tasks. Chapel provides a simple construct, the begin statement, to create tasks, introducing concurrency into a program in an unstructured way. In addition, Chapel introduces two type qualifiers, `sync` and `single`, for synchronization between tasks.

Chapel provides two constructs, the cobegin and coforall statements, to introduce concurrency in a more structured way. These constructs create multiple tasks but do not continue until the tasks have completed. In addition, Chapel provides two constructs, the sync- and serial statements, to suppress parallelism and insert synchronization. All four of these constructs can be implemented through judicious uses of the unstructured task-parallel constructs described in the previous section.

24.2 The Begin Statement

The begin statement creates a task to execute a statement. The syntax for the begin statement is given by

begin−statement:
  begin *statement*

Control continues concurrently with the statement following the begin statement.

Example (beginUnordered.chpl). The code

```
begin writeln("output from spawned task");
writeln("output from main task");
```

executes two `writeln` statements that output the strings to the terminal, but the ordering is purposely unspecified. There is no guarantee as to which statement will execute first. When the begin statement is executed, a new task is created that will execute the `writeln` statement within it. However, execution will continue immediately after task creation with the next statement.

Yield and return statements are not allowed in begin blocks. Break and continue statements may not be used to exit a begin block.

24.3 Synchronization Variables

Synchronization variables have a logical state associated with the value. The state of the variable is either *full* or *empty*. Normal reads of a synchronization variable cannot proceed until the variable's state is full. Normal writes of a synchronization variable cannot proceed until the variable's state is empty.

Chapel supports two types of synchronization variables: sync and single. Both types behave similarly, expect that a single variable may only be written once. Consequently, when a sync variable is read, its state transitions to empty, where as when a single variable is read, its state does not change. When either type of synchronization variable is written, its state transitions to full.

Sync and single are type qualifiers and preceed the variable's type in the declaration. Sync and single are only supported for all Chapel primitive types (§7.1) except strings and complex.

If a task attempts to read or write a synchronization variable that is not in the correct state, the task is suspended. When the variable transitions to the correct state, the task is resumed. If there are multiple tasks blocked waiting for the state transition, one is non-deterministically selected to proceed and the others continue to wait.

A synchronization variable is specified with a sync or single type given by the following syntax:

sync−type:
  sync *type−specifier*

single−type:
  single *type−specifier*

If a synchronization variable declaration has an initialization expression, then the variable is initially full, otherwise it is initially empty.

Example (beginWithSyncVar.chpl). The code

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

  proc sum():int {
    if (isLeaf) then
       return value;
```

```
      var x$: sync int;
      begin x$ = left.sum();
      var y = right.sum();
      return x$+y;
    }
  }
```

the sync variable `x$` is assigned by an asynchronous task created with the begin statement. The task returning the sum waits on the reading of `x$` until it has been assigned. By convention, synchronization variables end in **$** to provide a visual cue to the programmer indicating that the task may block.

Example (syncCounter.chpl). Sync variables are useful for tallying data from multiple tasks as well. A sync variable of type `int` is read and then written during an update so the full-empty semantics make these updates atomic. The code

```
var count$: sync int = 0;
begin count$ = count$ + 1;
begin count$ = count$ + 1;
begin count$ = count$ + 1;
```

creates three tasks to increment `count$`. If `count$` was not a sync variable, this code would be unsafe because between the points at which one task reads `count$` and writes `count$`, another task may increment it.

Example (singleVar.chpl). The following code implements a simple split-phase barrier using a single variable.

```
var count$: sync int = n; // counter which also serves as a lock
var release$: single bool; // barrier release

forall t in 1..n do begin {
  work(t);
  var myc = count$; // read the count, set state to empty
  if myc!=1 {
    write(".");
    count$ = myc-1; // update the count, set state to full
    // we could also do some work here before blocking
    release$;
  } else {
    release$ = true; // last one here, release everyone
    writeln("done");
  }
}
```

In each iteration of the forall loop after the work is completed, the task reads the count**$** variable, which is used to tally the number of tasks that have arrived. All tasks except the last task to arrive will block while trying to read the variable release**$**. The last task to arrive will write to release**$**, setting its state to full at which time all the other tasks can be unblocked and run.

If a formal argument is a synchronization type, the actual is passed by reference and the argument itself is a valid lvalue. The unqualified types `sync` or `single` can also be used to specify a generic formal argument. In this case, the actual must be a sync or single variable and it is passed by reference.

For generic formal arguments with unspecified types (§22.1.5), an actual that is `sync` or `single` is read before being passed to the function and the generic formal argument's type is set to the base type of the actual.

24.3.1 Predefined Single and Sync Methods

The following methods are defined for variables of sync and single type.

```
proc (sync t).readFE(): t
```

> Returns the value of the sync variable. This method blocks until the sync variable is full. The state of the sync variable is set to empty when this method completes.

```
proc (sync t).readFF(): t
proc (single t).readFF(): t
```

> Returns the value of the sync or single variable. This method blocks until the sync or single variable is full. The state of the sync or single variable remains full when this method completes.

```
proc (sync t).readXX(): t
proc (single t).readXX(): t
```

> Returns the value of the sync or single variable. This method is non-blocking and the state of the sync or single variable is unchanged when this method completes.

```
proc (sync t).writeEF(v: t)
proc (single t).writeEF(v: t)
```

> Assigns `v` to the value of the sync or single variable. This method blocks until the sync or single variable is empty. The state of the sync or single variable is set to full when this method completes.

```
proc (sync t).writeFF(v: t)
```

> Assigns `v` to the value of the sync variable. This method blocks until the sync variable is full. The state of the sync variable remains full when this method completes.

```
proc (sync t).writeXF(v: t)
```

> Assigns `v` to the value of the sync variable. This method is non-blocking and the state of the sync variable is set to full when this method completes.

```
proc (sync t).reset()
```

> Assigns the default value of type `t` to the value of the sync variable. This method is non-blocking and the state of the sync variable is set to empty when this method completes.

```
proc (sync t).isFull: bool
proc (single t).isFull: bool
```

> Returns `true` if the sync or single variable is full and `false` otherwise. This method is non-blocking and the state of the sync or single variable is unchanged when this method completes.

Note that `writeEF` and `readFE`/`readFF` methods (for `sync` and `single` variables, respectively) are implicitly invoked for writes and reads of synchronization variables.

Example (syncMethods.chpl). Given the following declarations

```
{ // }
var x$: sync int;
var y$: single int;
var z: int;
```

the code

```
x$ = 5;
y$ = 6;
z = x$ + y$;
```

is equivalent to

```
x$.writeEF(5);
y$.writeEF(6);
z = x$.readFE() + y$.readFF();
```

24.4 The Cobegin Statement

The cobegin statement is used to introduce concurrency within a block. The `cobegin` statement syntax is

cobegin-statement:
  cobegin *block-statement*

A new task is created for each statement in the block. Control continues when all of the tasks have finished.

Return statements are not allowed in cobegin blocks. Yield statement may only be lexically enclosed in cobegin blocks in parallel iterators (§21.4). Break and continue statements may not be used to exit a cobegin block.

Example (cobeginAndEquivalent.chpl). The cobegin statement

```
cobegin {
  stmt1();
  stmt2();
  stmt3();
}
```

is equivalent to the following code that uses only begin statements and single variables to introduce concurrency and synchronize:

```
var s1$, s2$, s3$: single bool;
begin { stmt1(); s1$ = true; }
begin { stmt2(); s2$ = true; }
begin { stmt3(); s3$ = true; }
s1$; s2$; s3$;
```

Each begin statement is executed concurrently but control does not continue past the final line above until each of the single variables is written, thereby ensuring that each of the functions has finished.

24.5 The Coforall Loop

The coforall loop is a variant of the cobegin statement in loop form. The syntax for the coforall loop is given by

coforall−statement:
  coforall *index−var−declaration* **in** *iteratable−expression* **do** *statement*
  coforall *index−var−declaration* **in** *iteratable−expression block−statement*
  coforall *iteratable−expression* **do** *statement*
  coforall *iteratable−expression block−statement*

The `coforall` loop creates a separate task for each iteration of the loop. Control continues with the statement following the `coforall` loop after all tasks corresponding to the iterations of the loop have completed.

Return statements are not allowed in coforall blocks. Yield statement may only be lexically enclosed in coforall blocks in parallel iterators (§21.4). Break and continue statements may not be used to exit a coforall block.

Example (coforallAndEquivalent.chpl). The coforall statement

```
coforall i in iterator() {
  body();
}
```

is equivalent to the following code that uses only begin statements and sync and single variables to introduce concurrency and synchronize:

```
var runningCount$: sync int = 1;
var finished$: single bool;
for i in iterator() {
  runningCount$ += 1;
  begin {
    body();
    var tmp = runningCount$;
    runningCount$ = tmp-1;
    if tmp == 1 then finished$ = true;
  }
}
var tmp = runningCount$;
runningCount$ = tmp-1;
if tmp == 1 then finished$ = true;
finished$;
```

Each call to `body()` executes concurrently because it is in a begin statement. The sync variable `runningCount$` is used to keep track of the number of executing tasks plus one for the main task. When this variable reaches zero, the single variable `finished$` is used to signal that all of the tasks have completed. Thus control does not continue past the last line until all of the tasks have completed.

24.6 The Sync Statement

The sync statement acts as a join of all dynamically encountered begins from within a statement. The syntax for the sync statement is given by

sync-statement:
sync *statement*
sync *block-statement*

Return statements are not allowed in sync statement blocks. Yield statement may only be lexically enclosed in sync statement blocks in parallel iterators (§21.4). Break and continue statements may not be used to exit a sync statement block.

Example (syncStmt1.chpl). The sync statement can be used to wait for many dynamically created tasks.

```
sync for i in 1..n do begin work();
```

The for loop is within a sync statement and thus the tasks created in each iteration of the loop must complete before the continuing past the sync statement.

Example (syncStmt2.chpl). The sync statement

```
sync {
  begin stmt1();
  begin stmt2();
}
```

is similar to the following cobegin statement

```
cobegin {
  stmt1();
  stmt2();
}
```

except that if begin statements are dynamically encountered when `stmt1()` or `stmt2()` are executed, then the former code will wait for these begin statements to complete whereas the latter code will not.

24.7 The Serial Statement

The `serial` statement can be used to dynamically disable parallelism. The syntax is:

serial-statement:
serial *expression* **do** *statement*
serial *expression block-statement*

where the expression evaluates to a bool type. Independent of that value, the *statement* is evaluated. If the expression is true, any dynamically encountered code that would result in new tasks is executed without creating any new tasks. In effect, execution is serialized.

Example (serialStmt1.chpl). In the code

```
proc f(i) {
  serial i<13 {
    cobegin {
      work(i);
      work(i);
    }
  }
}

for i in lo..hi {
  f(i);
}
```

the serial statement in procedure f() inhibits concurrent execution of work() if the variable i is less than 13.

Example (serialStmt2.chpl). The code

```
serial true {
  begin stmt1();
  cobegin {
    stmt2();
    stmt3();
  }
  coforall i in 1..n do stmt4();
}
```

is equivalent to

```
stmt1();
{
  stmt2();
  stmt3();
}
for i in 1..n do stmt4();
```

because the expression evaluated to determine whether to serialize always evaluates to true.

24.8 Atomic Statements

Open issue. This section describes a feature that is a work-in-progress. We seek feedback and collaboration in this area from the broader community.

The *atomic statement* is used to specify that a statement should appear to execute atomically from other tasks' point of view. In particular, no task will see memory in a state that would reflect that the atomic statement had begun executing but had not yet completed.

Open issue. This definition of the atomic statement provides a notion of *strong atomicity* since the action will appear atomic to any task at any point in its execution. For performance reasons, it could be more practical to support *weak atomicity* in which the statement's atomicity is only guaranteed with respect to other atomic statements. We may also pursue using atomic type qualifiers as a means of marking data that should be accessed atomically inside or outside an atomic section.

The syntax for the atomic statement is given by:

atomic-statement:
  atomic *statement*

Example. The following code illustrates the use of an atomic statement to perform an insertion into a doubly-linked list:

```
proc Node.insertAfter(newNode: Node) {
  atomic {
    newNode.prev = this;
    newNode.next = this.next;
    if this.next then this.next.prev = newNode;
    this.next = newNode;
  }
}
```

The use of the atomic statement in this routine prevents other tasks from viewing the list in a partially-updated state in which the pointers might not be self-consistent.

25 Data Parallelism

Chapel provides two explicit data-parallel constructs (the forall-statement and the forall-expression) and several idioms that support data parallelism implicitly (whole-array assignment, function and operator promotion, reductions, and scans).

This chapter details data parallelism as follows:

- §25.1 describes the forall statement.
- §25.2 describes forall expressions
- §25.3 describes promotion.
- §25.4 describes reductions and scans
- §25.5 describes the configuration constants for controlling default data parallelism.

25.1 The Forall Statement

The forall statement is a concurrent variant of the for statement described in §11.8.

25.1.1 Syntax

The syntax of the forall statement is given by

forall-statement:
  forall *index-var-declaration* **in** *iteratable-expression* **do** *statement*
  forall *index-var-declaration* **in** *iteratable-expression block-statement*
  forall *iteratable-expression* **do** *statement*
  forall *iteratable-expression block-statement*
  [*index-var-declaration* **in** *iteratable-expression*] *statement*
  [*iteratable-expression*] *statement*

As with the for statement, the indices may be omitted if they are unnecessary and the `do` keyword may be omitted before a block statement. The square bracketed form is a syntactic convenience.

25.1.2 Execution and Serializability

The forall statement evaluates the loop body once for each element yielded by the *iteratable−expression*. Each instance of the forall loop's body may be executed concurrently with the others, but this is not guaranteed. In particular, the loop must be serializable. Details regarding concurrency and iterator implementation are described in 21.4.

This differs from the semantics of the `coforall` loop, discussed in §24.5, where each iteration is guaranteed to run using distinct tasks. The `coforall` loop thus has potentially higher overhead than a forall loop, but in cases where concurrency is required for correctness, it is essential.

Control continues with the statement following the forall loop only after every iteration has been completely evaluated. At this point, all data accesses within the body of the forall loop will be guaranteed to be completed.

The following statements may not be lexically enclosed in forall statements: break statements, and return statements. Yield statement may only be lexically enclosed in forall statements in parallel iterators (§21.4).

> *Example (forallStmt.chpl).* In the code
>
> ```
> forall i in 1..N do
>   a(i) = b(i);
> ```
>
> the user has stated that the element-wise assignments can execute concurrently. This loop may be executed serially with a single task, or by using a distinct task for every iteration, or by using a number of tasks where each task executes a number of iterations. This loop can also be written as
>
> ```
> [i in 1..N] a(i) = b(i);
> ```

25.1.3 Zipper Iteration

Zipper iteration has the same semantics as described in §11.8.1 and §21.4 for parallel iteration.

25.2 The Forall Expression

The forall expression is a concurrent variant of the for expression described in §10.21.

25.2.1 Syntax

The syntax of a forall expression is given by

forall−expression:
  forall *index−var−declaration* **in** *iteratable−expression* **do** *expression*
  forall *iteratable−expression* **do** *expression*
  [*index−var−declaration* **in** *iteratable−expression*] *expression*
  [*iteratable−expression*] *expression*

As with the for expression, the indices may be omitted if they are unnecessary. The `do` keyword is always required in the keyword-based notation. The bracketed form is a syntactic convenience.

25.2.2 Execution and Serializability

The forall expression executes a forall loop (§25.1), evaluates the body expression on each iteration of the loop, and returns the resulting values as a collection. The size and shape of that collection are determined by the iteratable-expression.

> *Example (forallExpr.chpl).* The code
>
> ```
> writeln(+ reduce [i in 1..10] i**2);
> ```
>
> applies a reduction to a forall-expression that evaluates the square of the indices in the range `1..10`.

The forall expression follows the semantics of the forall statement as described in 25.1.2.

25.2.3 Zipper Iteration

Forall expression also support zippered iteration semantics as described in §11.8.1 and §21.4 for parallel iteration.

25.2.4 Filtering Predicates in Forall Expressions

A filtering predicate is an if expression that is immediately enclosed by a forall expression. A filtering predicate does not require an else clause. Such an if expression filters the iterations of the forall expression. The iterations for which the condition does not hold are not reflected in the result of the forall expression.

> *Example (forallFilter.chpl).* The following expression returns every other element starting with the first:
>
> ```
> [i in 1..s.numElements] if i % 2 == 1 then s(i)
> ```

25.3 Promotion

A function that expects one or more scalar arguments but is called with one or more arrays, domains, ranges, or iterators is promoted if the element types of the arrays, the index types of the domains and/or ranges, or the yielded types of the iterators can be resolved to the type of the argument. The rules of when an overloaded function can be promoted are discussed in §13.14.

In addition to scalar functions, operators and casts are also promoted.

> *Example (promotion.chpl).* Given the array
>
> ```
> var A: [1..5] int = [i in 1..5] i;
> ```

and the function

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

then the call `square(A)` results in the promotion of the `square` function over the values in the array `A`. The result is an iterator that returns the values `1`, `4`, `9`, `16`, and `25`.

Whole array operations are a form of promotion as applied to operators rather than functions.

25.3.1 Zipper Promotion

Promotion also supports zippered iteration semantics as described in §11.8.1 and §21.4 for parallel iteration.

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 (zipper-promotion.chpl). Given a function defined as

```
proc foo(i: int, j: int) {
  return (i,j);
}
```

and a call to this function written

```
writeln(foo(1..3, 4..6));
```

then the output is

```
(1, 4) (2, 5) (3, 6)
```

25.3.2 Whole Array Assignment

Whole array assignment is a considered a degenerate case of promotion and is implicitly parallel. The assignment statement

```
LHS = RHS;
```

is equivalent to

```
forall (e1,e2) in (LHS,RHS) do
  e1 = e2;
```

25.3.3 Evaluation Order

The semantics of whole array assignment and promotion are different from most array programming languages. Specifically, the compiler does not insert array temporaries for such operations if any of the right-hand side array expressions alias the left-hand side expression.

> *Example*. If `A` is an array declared over the indices `1..5`, then the following codes are not equivalent:
>
> ```
> A[2..4] = A[1..3] + A[3..5];
> ```
>
> and
>
> ```
> var T = A[1..3] + A[3..5];
> A[2..4] = T;
> ```
>
> This follows because, in the former code, some of the new values that are assigned to `A` may be read to compute the sum depending on the number of tasks used to implement the data parallel statement.

25.4 Reductions and Scans

Chapel provides reduction and scan expressions that apply operators to aggregate expressions in stylized ways. Reduction expressions collapse the aggregate's values down to a summary value. Scan expressions compute an aggregate of results where each result value stores the result of a reduction applied to all of the elements in the aggregate up to that expression. Chapel provides a number of predefined reduction and scan operators, and also supports a mechanism for the user to define additional reductions and scans (Chapter 28).

25.4.1 Reduction Expressions

A reduction expression applies a reduction operator to an aggregate expression, collapsing the aggregate's dimensions down into a result value (typically a scalar or summary expression that is independent of the input aggregate's size). For example, a sum reduction computes the sum of all the elements in the input aggregate expression.

The syntax for a reduction expression is given by:

reduce−expression:
  reduce−scan−operator **reduce** *expression*
  class−type **reduce** *expression*

reduce−scan−operator: *one of*
  + ∗ && || & | ˆ **min max minloc maxloc**

Chapel's predefined reduction operators are defined by *reduce−scan−operator* above. In order, they are: sum, product, logical-and, logical-or, bitwise-and, bitwise-or, bitwise-exclusive-or, minimum, maximum, minimum-with-location, and maximum-with-location. The minimum reduction returns the minimum value as defined by the < operator. The maximum reduction returns the maximum value as defined by the > operator. The minimum-with-location reduction returns the lowest index position with the minimum value (as defined by the < operator). The maximum-with-location reduction returns the lowest index position with the maximum value (as defined by the > operator).

The expression on the right-hand side of the `reduce` keyword can be of any type that can be iterated over and to which the reduction operator can be applied. For example, the bitwise-and operator can be applied to arrays of boolean or integral types to compute the bitwise-and of all the values in the array.

The minimum-with-location and maximum-with-location reductions take a 2-tuple of arguments where the first tuple element is the collection of values for which the minimum/maximum value is to be computed. The second tuple element is a collection of indices with the same size and shape that provides names for the locations of the values in the first argument. The reduction returns a tuple containing the minimum/maximum value in the first position and the location of the value in the second position.

> *Example (reduce-loc.chpl).* The first line below computes the smallest element in an array `A` as well as its index, storing the results in `minA` and `minALoc`, respectively. It then computes the largest element in a forall expression making calls to a function `foo()`, storing the value and its number in `maxVal` and `maxValNum`.
>
> ```
> var (minA, minALoc) = minloc reduce (A, A.domain);
> var (maxVal, maxValNum) = maxloc reduce ([i in 1..n] foo(i), 1..n);
> ```

User-defined reductions are specified by preceding the keyword `reduce` by the class type that implements the reduction interface as described in §28.

25.4.2 Scan Expressions

A scan expression applies a scan operator to an aggregate expression, resulting in an aggregate expression of the same size and shape. The output values represent the result of the operator applied to all elements up to and including the corresponding element in the input.

The syntax for a scan expression is given by:

scan-expression:
  reduce-scan-operator **scan** *expression*
  class-type **scan** *expression*

The predefined scans are defined by *reduce-scan-operator*. These are identical to the predefined reductions and are described in §25.4.1.

The expression on the right-hand side of the scan can be of any type that can be iterated over and to which the operator can be applied.

> *Example.* Given an array
>
> ```
> var A: [1..3] int = 1;
> ```
>
> that is initialized such that each element contains one, then the code
>
> ```
> writeln(+ scan A);
> ```

outputs the results of scanning the array with the sum operator. The output is

```
1 2 3
```

User-defined scans are specified by preceding the keyword `scan` by the class type that implements the scan interface as described in Chapter 28.

25.5 Configuration Constants for Default Data Parallelism

The following configuration constants are provided to control the degree of data parallelism over ranges, default domains, and default arrays:

Config Const	**Type**	**Default**
`dataParTasksPerLocale`	`int`	Number of cores per locale
`dataParIgnoreRunningTasks`	`bool`	`true`
`dataParMinGranularity`	`int`	`1`

The configuration constant `dataParTasksPerLocale` specifies the number of tasks to use when executing a forall loop over a range, default domain, or default array. The actual number of tasks may be fewer depending on the other two configuration constants. A value of zero results in using the default value.

The configuration constant `dataParIgnoreRunningTasks`, when true, has no effect on the number of tasks to use to execute the forall loop. When false, the number of tasks per locale is decreased by the number of tasks that are already running on the locale, with a minimum value of one.

The configuration constant `dataParMinGranularity` specifies the minimum number of iterations per task created. The number of tasks is decreased so that the number of iterations per task is never less than the specified value.

For distributed domains and arrays that have these same configuration constants (*e.g.*, Block and Cyclic distributions), these same global configuration constants are used to specify their default behavior within each locale (§33.1, §33.2).

26 Locales

Chapel provides high-level abstractions that allow programmers to exploit locality by controlling the affinity of both data and tasks to abstract units of processing and storage capabilities called *locales*. The *on-statement* allows for the migration of tasks to *remote* locales.

Throughout this section, the term *local* will be used to describe the locale on which a task is running, the data located on this locale, and any tasks running on this locale. The term *remote* will be used to describe another locale, the data on another locale, and the tasks running on another locale.

26.1 Locales

A *locale* is a portion of the target parallel architecture that has processing and storage capabilities. Chapel implementations should typically define locales for a target architecture such that tasks running within a locale have roughly uniform access to values stored in the locale's local memory and longer latencies for accessing the memories of other locales. As an example, a cluster of multicore nodes or SMPs would typically define each node to be a locale. In contrast a pure shared memory machine would be defined as a single locale.

26.1.1 Locale Types

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

26.1.2 Locale Methods

The locale type supports the following methods:

```
proc locale.callStackSize: uint(64);
```

Returns the per-task call stack size used when creating tasks on the locale in question. A value of 0 indicates that the call stack size is determined by the system.

```
proc locale.id: int;
```

Returns a unique integer for each locale, from 0 to the number of locales less one.

```
proc locale.name: string;
```

Returns the name of the locale.

```
proc locale.numCores: int;
```

Returns the number of processor cores available on a given locale.

```
use Memory;
proc locale.physicalMemory(unit: MemUnits=MemUnits.Bytes, type retType=int(64)): retType;
```

Returns the amount of physical memory available on a given locale in terms of the specified memory units (Bytes, KB, MB, or GB) using a value of the specified return type.

26.1.3 The Predefined Locales Array

Chapel provides a predefined environment that stores information about the locales used during program execution. This *execution environment* contains definitions for the array of locales on which the program is executing (`Locales`), a domain for that array (`LocaleSpace`), and the number of locales (`numLocales`).

```
config const numLocales: int;
const LocaleSpace: domain(1) = [0..numLocales-1];
const Locales: [LocaleSpace] locale;
```

When a Chapel program starts, a single task executes `main` on `Locales(0)`.

Note that the Locales array is typically defined such that distinct elements refer to distinct resources on the target parallel architecture. In particular, the Locales array itself should not be used in an oversubscribed manner in which a single processor resource is represented by multiple locale values (except during development). Oversubscription should instead be handled by creating an aggregate of locale values and referring to it in place of the Locales array.

> *Rationale*. This design choice encourages clarity in the program's source text and enables more opportunities for optimization.
>
> For development purposes, oversubscription is still very useful and this should be supported by Chapel implementations to allow development on smaller machines.

> *Example*. The code
>
> ```
> const MyLocales: [loc in 0..numLocales*4] locale = Locales(loc%numLocales);
> on MyLocales(i) ...
> ```
>
> defines a new array `MyLocales` that is four times the size of the `Locales` array. Each locale is added to the `MyLocales` array four times in a round-robin fashion.

26.1.4 The *here* Locale

A predefined constant locale `here` can be used anywhere in a Chapel program. It refers to the locale that the current task is running on.

> *Example*. The code
>
> ```
> on Locales(1) {
>   writeln(here.id);
> }
> ```
>
> results in the output `1` because the `writeln` statement is executed on locale 1.

The identifier `here` is not a keyword and can be overridden.

26.1.5 Querying the Locale of an Expression

The locale associated with an expression (where the expression is stored) is queried using the following syntax:

locale-access-expression:
  expression . **locale**

When the expression is a class, the access returns the locale on which the class object exists rather than the reference to the class. If the expression is a value, it is considered local. The implementation may warn about this behavior. If the expression is a locale, it is returned directly.

Example. Given a class C and a record R, the code

```
on Locales(1) {
  var x: int;
  var c: C;
  var r: R;
  on Locales(2) {
    on Locales(3) {
      c = new C();
      r = new R();
    }
    writeln(x.locale.id);
    writeln(c.locale.id);
    writeln(r.locale.id);
  }
}
```

results in the output

```
1
3
1
```

The variable `x` is declared and exists on `Locales(1)`. The variable `c` is a class reference. The reference exists on `Locales(1)` but the object itself exists on `Locales(3)`. The locale access returns the locale where the object exists. Lastly, the variable `r` is a record and has value semantics. It exists on `Locales(1)` even though it is assigned a value on a remote locale.

Global (non-distributed) constants are replicated across all locales.

Example. For example, the following code:

```
const c = 10;
for loc in Locales do on loc do
    writeln(c.locale.id);
```

outputs

```
0
1
2
3
4
```

when running on 5 locales.

26.2 The On Statement

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

on−statement:
  on *expression* **do** *statement*
  on *expression block−statement*

The locale of the expression is automatically queried as described in §26.1.5. Execution of the statement occurs on this specified locale and then continues after the `on-statement`.

Return statements may not be lexically enclosed in on statements. Yield statement may only be lexically enclosed in on statements in parallel iterators §21.4.

26.2.1 Remote Variable Declarations

By default, when new variables and data objects are created, they are created in the locale where the task is running. Variables can be defined within an *on−statement* to define them on a particular locale such that the scope of the variables is outside the *on−statement*. This is accomplished using a similar syntax but omitting the `do` keyword and braces. The syntax is given by:

remote−variable−declaration−statement:
  on *expression variable−declaration−statement*

27 Domain Maps

A domain map specifies the implementation of the domains and arrays that are *mapped* using it. That is, it defines how domain indices and array elements are mapped to locales, how they are stored in memory, and how operations such as accesses, iteration, and slicing are performed. Each domain and array is mapped using some domain map.

A domain map is either a *layout* or a *distribution*. A layout describes domains and arrays that exist on a single locale, whereas a distribution describes domains and arrays that are partitioned across multiple locales.

A domain map is represented in the program with an instance of a *domain map class*. Chapel provides a set of standard domain map classes. Users can create domain map classes as well.

Domain maps are presented as follows:

- domain maps for domain types §27.1, domain values §27.2, and arrays §27.3
- domain maps are not retained upon domain assignment §27.4
- standard layouts §34
- standard distributions §33
- user-defined domain maps §29

27.1 Domain Maps for Domain Types

Each domain type has a domain map associated with it. This domain map is used to map all domain values of this type (§27.2).

If a domain type does not have a domain map specified for it explicitly as described below, a default domain map is provided by the Chapel implementation. Such a domain map will typically be a layout that maps the entire domain to the locale on which the domain value is created or the domain or array variable is declared.

> *Cray's Chapel Implementation*. The default domain map provided by the Cray Chapel compiler is such a layout. The storage for the representation of a domain's index set is placed on the locale where the domain variable is declared. The storage for the elements of arrays declared over domains with the default map is placed on the locale where the array variable is declared. Arrays declared over rectangular domains with this default map are laid out in memory in row-major order.

A domain map can be specified explicitly by providing a *dmap value* in a `dmapped` clause:

mapped-domain-type:
  domain-type **dmapped** *dmap-value*

dmap-value:
  expression

A dmap value consists of an instance of a domain map class wrapped in an instance of the predefined record `dmap`. The domain map class is chosen and instantiated by the user. `dmap` behaves like a generic record with a single generic field, which holds the domain map instance.

Example. The code

```
use BlockDist;
var MyBlockDist: dmap(Block(rank=2));
```

declares a variable capable of storing dmap values for a two-dimensional Block distribution. The Block distribution is described in more detail in §33.1.

Example. The code

```
use BlockDist;
var MyBlockDist: dmap(Block(rank=2)) = new dmap(new Block([1..5,1..6]));
```

creates a dmap value wrapping a two-dimensional Block distribution with a bounding box of `[1..5, 1..6]` over all of the locales.

Example. The code

```
use BlockDist;
var MyBlockDist = new dmap(new Block([1..5,1..6]));
type MyBlockedDom = domain(2) dmapped MyBlockDist;
```

defines a two-dimensional rectangular domain type that is mapped using a Block distribution.

The following syntactic sugar is provided within the `dmapped` clause. If a `dmapped` clause starts with the name of a domain map class, it is considered to be a constructor expression as if preceeded by `new`. The resulting domain map instance is wrapped in a newly-created instance of `dmap` implicitly.

Example. The code

```
use BlockDist;
type BlockDom = domain(2) dmapped Block([1..5,1..6]);
```

is equivalent to

```
use BlockDist;
type BlockDom = domain(2) dmapped new dmap(new Block([1..5,1..6]));
```

27.2 Domain Maps for Domain Values

A domain value is always mapped using the domain map of that value's type. The type inferred for a domain literal (§19.2.1) has a default domain map.

Example. In the following code

```
use BlockDist;
var MyDomLiteral = [1..2,1..3];
var MyBlockedDom: domain(2) dmapped Block([1..5,1..6]) = MyDomLiteral;
```

`MyDomLiteral` is given the inferred type of the domain literal and so will be mapped using a default map. MyBlockedDom is given a type explicitly, in accordance to which it will be mapped using a Block distribution.

A domain value's map can be changed explicitly with a `dmapped` clause, in the same way as a domain type's map.

mapped-domain-expression:
  domain-expression **dmapped** *dmap-value*

Example. In the following code

```
use BlockDist;
var MyBlockedDomLiteral1 = [1..2,1..3] dmapped new dmap(new Block([1..5,1..6]));
var MyBlockedDomLiteral2 = [1..2,1..3] dmapped Block([1..5,1..6]);
```

both `MyBlockedDomLiteral1` and `MyBlockedDomLiteral2` will be mapped using a Block distribution.

27.3 Domain Maps for Arrays

Each array is mapped using the domain map of the domain over which the array was declared.

Example. In the code

```
use BlockDist;
var Dom: domain(2) dmapped Block([1..5,1..6]) = [1..5,1..6];
var MyArray: [Dom] real;
```

the domain map used for `MyArray` is the Block distribution from the type of `Dom`.

27.4 Domain Maps Are Not Retained upon Domain Assignment

Domain assignment (§19.8.1) transfers only the index set of the right-hand side expression. The implementation of the left-hand side domain expression, including its domain map, is determined by its type and so does not change upon a domain assignment.

Example. In the code

```
use BlockDist;
var Dom1: domain(2) dmapped Block([1..5,1..6]) = [1..5,1..6];
var Dom2: domain(2) = Dom1;
```

`Dom2` is mapped using the default distribution, despite `Dom1` having a Block distribution.

Example. In the code

```
use BlockDist;
var Dom1: domain(2) dmapped Block([1..5,1..6]) = [1..5,1..6];
var Dom2 = Dom1;
```

`Dom2` is mapped using the same distribution as `Dom1`. This is because the declaration of `Dom2` lacks an explicit type specifier and so its type is defined to be the type of its initialization expression, `Dom1`. So in this situation the effect is that the domain map does transfer upon an initializing assignment.

28 User-Defined Reductions and Scans

User-defined reductions and scans are supported via class definitions where the class implements a structural interface. The definition of this structural interface is forthcoming. The following paper sketched out such an interface:

> S. J. Deitz, D. Callahan, B. L. Chamberlain, and L. Snyder. **Global-view abstractions for user-defined reductions and scans**. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

29 User-Defined Domain Maps

This chapter is forthcoming. As a placeholder, please refer to doc/technotes/README.dsi in the Chapel release.

30 Memory Consistency Model

> *Open issue*. This chapter is a work-in-progress and represents an area where we are particularly interested in feedback from, and collaboration with, the broader community.

Chapel's memory consistency model is well-defined for programs that are *data-race-free*. Such programs are sequentially consistent. For other programs, no specific guarantees can be made about the program's execution.

> *Rationale*. Chapel presents a memory consistency model that is less strict than Java's. It does so because it does not strive to provide the same dynamic security requirements as Java does.

Accessing a synchronization variable (`sync` or `single`, §24.3) is the only way to correctly synchronize between two Chapel tasks. Such reads and writes serve as memory fences, preventing reordering of reads and writes to traditional variables across the synchronization varible's access. When the same memory location is written by one task and read by another, the ordering of the read relative to the write is undefined unless there is an intervening access to a synchronization variable.

Example. In this example, a synchronization variable is used to (a) ensure that all writes to an array of unsynchronized variables are complete, (b) signal that fact to a second task, and (c) pass along the number of values that are valid for reading.

```
var A: [1..100] real;
var done$: sync int;           // initially empty
cobegin {
  { // Reader task
    const numToRead = done$;   // block until writes are complete
    for i in 1..numToRead do
      writeln("A[", i, "] = ", A[i]);
  }
  {  // Writer task
    const numToWrite = 23;     // an arbitrary number
    for i in 1..numToWrite do
      A[i] = i/10.0;
    done$ = numToWrite;        // fence writes to A and signal done
  }
}
```

Example (syncSpinWait.chpl). One consequence of Chapel's memory consistency model is that a task cannot spin-wait on a variable waiting for another task to write to that variable. The behavior of the following code is undefined:

```
var x: int;
cobegin {
  while x != 1 do ;  // spin wait
  x = 1;
}
```

In contrast, spinning on a synchronization variable has well-defined behavior:

```
var x$: sync int;
cobegin {
  while x$.readXX() != 1 do ;  // spin wait
  x$.writeXF(1);
}
```

In this code, the first statement in the cobegin statement executes a loop until the variable is set to one. The second statement in the cobegin statement sets the variable to one. Neither of these statements block.

Future. Upon completion, Chapel's atomic statement (§24.8) will serve as an additional means of correctly synchronizing between tasks.

31 Interoperability

Chapel's interoperability features support cooperation between Chapel and other languages. They provide the ability to create software systems that incorporate both Chapel and non-Chapel components. Thus, they support the reuse of existing software components while leveraging the unique features of the Chapel language.

Interoperability can be broken down in terms of the exchange of types, variables and procedures, and whether these are imported or exported. An overview of procedure importing and exporting is provided in §31.1. Details on sharing types, variables and procedures are supplied in §31.2. For information on specific packages which have been integrated with Chapel, consult the associated README file under `CHPL_HOME/doc/technotes`.

> *Future.*
>
> At present, the backend language for Chapel is C, which makes it relatively easy to call C libraries from Chapel and vice versa. To support a variety of platforms without requiring recompilation, it may be desiriable to move to an intermediate-language model.
>
> In that case, each supported platform must minimally support that virtual machine. However, in addition to increased portability, a virtual machine model may expose elements of the underlying machine's programming model (hardware task queues, automated garbage collection, etc.) that are not easily rendered in C. In addition, the virtual machine model can support run-time task migration.

The remainder of this chapter documents Chapel support of interoperability through the existing C-language backend.

31.1 Interoperability Overview

The following two subsections provide an overview of calling externally-defined (C) routines in Chapel, and setting up Chapel routines so they can be called from external (C) code.

31.1.1 Calling External Functions

To use an external function in a Chapel program, it is necessary to inform the Chapel compiler of that routine's signature through an external function declaration. This permits Chapel to bind calls to that function signature during function resolution. The user must also supply a definition for the referenced function by naming a C source file, an object file or an object library on the `chpl` command line.

An external procedure declaration has the following syntax:

external-procedure-declaration-statement:
  extern *external-name*$_{opt}$ **proc** *function-name argument-list return-intent*$_{opt}$ *return-type*$_{opt}$

Chapel will call the external function using the parameter types supplied in the `extern` declaration. Therefore, in general, the type of each argument in the supplied *argument-list* must be the Chapel equivalent of the corresponding external type.

The return value of the function can be used by Chapel only if its type is declared using the optional *return-type* specifier. If it is omitted, Chapel assumes that no value is returned, or equivalently that the function returns `void`.

At present, external iterators are not supported.

> *Future*. The overloading of function names is also not supported directly in the compiler. However, one can use the *external-name* syntax to supply a name to be used by the linker. In this way, function overloading can be implemented "by hand". This syntax also supports calling external C++ routines: The *external-name* to use is the mangled function name generated by the external compilation environment[1].

> *Future*. Dynamic dispatch (polymorphism) is also unsupported in this version. But this is not ruled out in future versions. Since Chapel already supports type-based procedure declaration and resolution, it is a small step to translate a type-relative extern method declaration into a virtual method table entry. The mangled name of the correct external function must be supplied for each polymorphic type available. However, most likely the generation of `.chpl` header files from C and C++ libraries can be fully automated.

There are three ways to supply to the Chapel compiler the definition of an external function: as a C source file (`.c` or `.h`), as an object file and as an object library. It is platform-dependent whether static libraries (archives), dynamic libraries or both are supported. See the `chpl` man page for more information on how these file types are handled.

31.1.2 Calling Chapel Functions

To call a Chapel procedure from external code, it is necessary to expose the corresponding function symbol to the linker. This is done by adding the `export` linkage specifier to the function definition. The `export` specifier ensures that the corresponding procedure will be resolved, even if it is not called within the Chapel program or library being compiled.

An exported procedure declaration has the following syntax:

```
exported-procedure-declaration-statement:
  export external-name_opt proc function-name argument-list return-intent_opt return-type_opt
    function-body
```

If the optional *external-name* is supplied, then it is used verbatim as the exported function symbol. Otherwise, the Chapel name of the procedure is exported. The rest of the procedure declaration is the same as for a non-exported function. An exported procedure can be called from within Chapel as well. Currently, iterators cannot be exported.

---

[1] In UNIX-like programming environments, `nm` and `grep` can be used to find the mangled name of a given function within an object file or object library.

> *Future*. Currently, exported functions cannot have generic, `param` or type arguments. This is because such functions actually represent a family of functions, specific versions of which are instantiated as need during function resolution.
>
> Instantiating all possible versions of a template function is not practical in general. However, if explicit instantiation were supported in Chapel, an explicit instantiation with the export linkage specifier would clearly indicate that the matching template function was to be instantiated with the given `param` values and argument types.

31.2 Shared Language Elements

This section provides details on how to share Chapel types, variables and procedures with external code. It is written assuming that the intermediate language is C.

31.2.1 Shared Types

This subsection discusses how specific types are shared between Chapel and external code.

Referring to Standard C Types

In Chapel code, all standard C types must be expressed in terms of their Chapel equivalents. This is true, whether the entity is exported, imported or private. Standard C types and their corresponding Chapel types are shown in the following table.

C Type	Chapel Type	C Type	Chapel Type	C Type	Chapel Type
`int8_t`	`int(8)`	`uint8_t`	`uint(8)`	`_real32`	`real(32)`
`int16_t`	`int(16)`	`uint16_t`	`uint(16)`	`_real64`	`real(64)`
`int32_t`	`int(32)`	`uint32_t`	`uint(32)`	`_imag32`	`imag(32)`
`int64_t`	`int(64)`	`uint64_t`	`uint(64)`	`_imag64`	`imag(64)`
`chpl_bool`	`bool`	`const char*`	`string`		
`_complex64`	`complex(64)`	`_complex128`	`complex(128)`		

Standard C types are built-in. Their Chapel equivalents do not have to be declared using the `extern` keyword.

In C, the "colloquial" integer type names `char`, `signed char`, `unsigned char`, (`signed`) `short` (`int`), `unsigned short` (`int`), (`signed`) `int`, `unsigned int`, (`signed`) `long` (`int`), `unsigned long` (`int`), (`signed`) `long long` (`int`) and `unsigned long long` (`int`) may have an implementation-defined width.[2]. When referring to C types in a Chapel program, the burden of making sure the type sizes agree is on the user. A Chapel implementation must ensure that all of the C equivalents in the above table are defined and have the correct representation with respect to the corresponding Chapel type.

---

[2] However, most implementations have settled on using 8, 16, 32, and 64 bits (respectively) to represent `char`, `short`, `int` and `long`, and `long long` types

Referring to External C Types

An externally-defined type can be referenced using a external type declaration with the following syntax.

external−type−alias−declaration−statement:
  extern type *type−alias−declaration−list* ;

In each *type−alias−declaration*, if the *type−specifier* part is supplied, then Chapel uses the supplied type specifier internally. Otherwise, it treats the named type as an opaque type. The definition for an external type must be supplied by a C header file named on the `chpl` command line.

Fixed-size C array types can be described within Chapel using its homogenous tuple type. For example, the C typedef

```
typedef double vec[3];
```

can be described in Chapel using

```
extern type vec = 3*real(64);
```

Referring to External C Structs

External C struct types can be referred to within Chapel by prefixing a Chapel `record` definition with the `extern` keyword.

external−record−declaration−statement:
  extern *simple−record−declaration−statement*

For example, consider an external C structure defined in `foo.h` called `fltdbl`.

```
typedef struct _fltdbl {
  float x;
  double y;
} fltdbl;
```

This type could be referred to within a Chapel program using

```
extern record fltdbl {
  var x: real(32);
  var y: real(64);
}
```

and defined by supplying `foo.h` on the `chpl` command line.

Within the Chapel declaration, some or all of the fields from the C structure may be omitted. The order of these fields need not match the order they were specified within the C code. Any fields that are not specified (or that cannot be specified because there is no equivalent Chapel type) cannot be referenced within the Chapel code.

A C header file containing the struct's definition in C must be specified on the chpl compiler command line. Note that currently only typdef'd C structures are supported. That is, in the C header file, the `struct` must be supplied with a type name through a `typedef` declaration.

Referring to External Structs Through Pointers

An external type which is a pointer to a `struct` can be referred to from Chapel using an external `class` declaration. External class declarations have the following syntax.

> *external-class-declaration-statement*:
>   **extern** *simple-class-declaration-statement*

External class declarations are similar to external record declarations as discussed above, but place additional requirements on the C code.

For example, given the declaration

```
extern class D {
  var x: real;
}
```

the requirements on the corresponding C code are:

1. There must be a struct type that is typedef'd to have the name `_D`.
2. A pointer-to-`_D` type must be typedef'd to have the name `D`.
3. The `_D` struct type must contain a field named `x` of type `double`.

Like external records/structs, it may also contain other fields that will simply be ignored by the Chapel compiler.

The following C typedef would fulfill the external Chapel class declaration shown above:

```
typedef struct __D {
  double x;
  int y;
} _D, *D;
```

where the Chapel compiler would not know about the 'y' field and therefore could not refer to it or manipulate it.

Opaque Types

It is possible refer to external pointer-based C types that cannot be described in Chapel by using the "opaque" keyword. As the name implies, these types are opaque as far as Chapel is concerned and cannot be used for operations other than argument passing and assignment.

For example, Chapel could be used to call an external C function that returns a pointer to a structure (that can't or won't be described as an external class) as follows:

```
extern proc returnStructPtr(): opaque;

var structPtr: opaque = returnStructPtr();
```

However, because the type of `structPtr` is opaque, it can be used only in assignments and the arguments of functions expecting the same underlying type.

```
var copyOfStructPtr = structPtr;

extern proc operateOnStructPtr(ptr: opaque);
operateOnStructPtr(structPtr);
```

Like a `void*` in C, Chapel's `opaque` carries no information regarding the underlying type. It therefore subverts type safety, and should be used with caution.

31.2.2 Shared Data

This subsection discusses how to access external variables and constants.

A C variable or constant can be referred to within Chapel by prefixing its declaration with the extern keyword. For example:

```
extern var bar: foo;
```

would tell the Chapel compiler about an external C variable named `bar` of type `foo`. Similarly,

```
extern const baz: int(32);
```

would refer to an external 32-bit integer constant named `baz` in the C code. In practice, external consts can be used to provide Chapel definitions for #defines and enum symbols in addition to traditional C constants.

> *Cray's Chapel Implementation.* Note that since params must be known to Chapel at compile-time (and because the Chapel compiler doesn't have the ability to parse C code), external params are not supported.

31.2.3 Shared Procedures

This subsection provides additional detail and examples for calling external procedures from Chapel and for exporting Chapel functions for external use.

Calling External C Functions

To call an external C function, a prototype of the routine must appear in the Chapel code. This is accomplished by providing the Chapel signature of the function preceded by the `extern` keyword. For example, for a C function foo() that takes no arguments and returns nothing, the prototype would be:

```
extern proc foo();
```

To refer to the return value of a C function, its type must be supplied through a *return-type* clause in the prototype.[3]

If the above function returns a C `double`, it would be declared as:

---

[3] The return type cannot be inferred, since an `extern` procedure declaration has no body.

```
extern proc foo(): real;
```

Similarly, for external functions that expect arguments, the types of those arguments types may be declared in Chapel using explicit argument type specifiers.

The types of function arguments may be omitted from the external procedure declaration, in which case they are inferred based on the Chapel callsite. For example, the Chapel code

```
extern proc foo(x: int, y): real;
var a, b: int;
foo(a, b);
```

would imply that the external function foo takes two 32-bit integer values and returns a 64-bit real. External function declarations with omitted type arguments can also be used call external C macros.

External function arguments can be declared using the *default-expression* syntax. In this case, the default argument will be supplied by the Chapel compiler if the corrspondinging actual argument is omitted at the callsite. For example:

```
extern proc foo(x: int, y = 1.2): real;
foo(0);
```

Would cause external function foo() to be invoked with the arguments 0 and 1.2.

C varargs functions can be declared using Chapel's *variable-argument-expression* syntax (`...`). For example, the C `printf` function can be declared in Chapel as

```
extern proc printf(fmt: string, vals...?numvals): int;
```

External C functions or macros that accept type arguments can also be prototyped in Chapel by declaring the argument as a type. For example:

```
extern foo(type t);
```

Calling such a routine with a Chapel type will cause the type identifier (e.g., 'int') to be passed to the routine.[4]

31.2.4 Calling Chapel Procedures Externally

To call a Chapel procedure from external code, the procedure name must be exported using the `export` keyword. An exported procedure taking no arguments and returning void can be declared as:

```
export proc foo();
```

If the procedure body is omitted, the procedure declaration is a prototype; the body of the procedure must be supplied elsewhere. In a prototype, the return type must be declared; otherwise, it is assumed to be `void`. If the body is supplied, the return type of the exported procedure is inferred from the type of its return expression(s).

If the optional *external-name* is supplied, that is the name used in linking with external code. For example, if we declare

---

[4] In practice, this will typically only be useful if the external function is a macro or built-in that can handle type identifiers.

```
export "myModule_foo" proc foo();
```

then the name `foo` is used to refer to the procedure within chapel code, whereas a call to the same function from C code would appear as `myModule_foo();`. If the external name is omitted, then its internal name is also used externally.

When a procedure is exported, all of the types and functions on which it depends are also exported. Iterators cannot be explicitly exported. However, they are inlined in Chapel code which uses them, so they are exported in effect.

31.2.5 Argument Passing

The manner in which arguments are passed to an external function can be controlled using argument intents. The following table shows the correspondence between Chapel intents and C argument type declarations. These correspondences pertain to both imported and exported function signatures.

Chapel	C
`T`	`const T`
`in T`	`T`
`inout T`	`T*`
`out T`	`T*`
`param`	
`type`	`char*`

Currently, `param` arguments are not allowed in an extern function declaration, and `type` args are passed as a string containing the name of the actual type being passed. Note that the level of indirection is changed when passing arguments to a C function using `inout` or `out` intent. The C code implementing that functions must dereference the argument to extract its value.

> *Future*. The `ref` intent has not yet been implemented, but will correspond with the C type `T*` when it is.

32 Standard Modules

Standard modules provide standard library support and are available to every Chapel program. The functions and other definitions of automatic modules are always available to a Chapel program. Optional modules can be accessed via use statements (§12.4).

The automatic modules are as follows:

`Math`	Math routines
`Base`	Basic routines
`Types`	Routines related to primitive types

The optional modules include:

`AdvancedIters`	Advanced iterator functions
`BitOps`	Bit manipulation routines
`Norm`	Routines for computing vector and matrix norms
`Random`	Random number generation routines
`Search`	Generic searching routines
`Sort`	Generic sorting routines
`Time`	Types and routines related to time
`UtilMath`	Math-related utilities

There is an expectation that each of these modules will be extended and that more standard modules will be defined over time.

32.1 Automatic Modules

Automatic modules are used by a Chapel program automatically. There is currently no way to avoid their use by a program, although we anticipate adding such a capability in the future.

32.1.1 Math

The module `Math` defines routines for mathematical computations. This module is used by default; there is no need to explicitly use this module. The Math module defines routines that are derived from and implemented via the standard C routines defined in `math.h`.

```
proc abs(i: int(?w)): int(w)
proc abs(i: uint(?w)): uint(w)
proc abs(param i: integral) param
proc abs(x: real): real
proc abs(x: real(32)): real(32)
proc abs(x: complex): real
```

Returns the absolute value of the argument.

```
proc acos(x: real): real
proc acos(x: real(32)): real(32)
```

Returns the arc cosine of the argument. It is an error if `x` is less than -1 or greater than 1.

```
proc acosh(x: real): real
proc acosh(x: real(32)): real(32)
```

Returns the inverse hyperbolic cosine of the argument. It is an error if `x` is less than 1.

```
proc asin(x: real): real
proc asin(x: real(32)): real(32)
```

Returns the arc sine of the argument. It is an error if `x` is less than -1 or greater than 1.

```
proc asinh(x: real): real
proc asinh(x: real(32)): real(32)
```

Returns the inverse hyperbolic sine of the argument.

```
proc atan(x: real): real
proc atan(x: real(32)): real(32)
```

Returns the arc tangent of the argument.

```
proc atan2(y: real, x: real): real
proc atan2(y: real(32), x: real(32)): real(32)
```

Returns the arc tangent of the two arguments. This is equivalent to the arc tangent of `y / x` except that the signs of `y` and `x` are used to determine the quadrant of the result.

```
proc atanh(x: real): real
proc atanh(x: real(32)): real(32)
```

Returns the inverse hyperbolic tangent of the argument. It is an error if `x` is less than -1 or greater than 1.

```
proc cbrt(x: real): real
proc cbrt(x: real(32)): real(32)
```

Returns the cube root of the argument.

```
proc ceil(x: real): real
proc ceil(x: real(32)): real(32)
```

Returns the value of the argument rounded up to the nearest integer.

```
proc conjg(a: complex(?w)): complex(w)
```

Returns the conjugate of `a`.

```
proc cos(x: real): real
proc cos(x: real(32)): real(32)
```

Returns the cosine of the argument.

```
proc cosh(x: real): real
proc cosh(x: real(32)): real(32)
```

Returns the hyperbolic cosine of the argument.

```
proc divceil(m: integral, n: integral)
proc divceil(param m: integral, param n: integral) param
```

Returns $\lceil \frac{m}{n} \rceil$, i.e., the fraction m/n rounded up to the nearest integer.

```
proc divfloor(m: integral, n: integral)
proc divfloor(param m: integral, param n: integral) param
```

Returns $\lfloor \frac{m}{n} \rfloor$, i.e., the fraction m/n rounded down to the nearest integer.

```
proc erf(x: real): real
proc erf(x: real(32)): real(32)
```

Returns the error function of the argument defined as

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

for the argument x.

```
proc erfc(x: real): real
proc erfc(x: real(32)): real(32)
```

Returns the complementary error function of the argument. This is equivalent to `1.0 - erf(x)`.

```
proc exp(x: real): real
proc exp(x: real(32)): real(32)
```

Returns the value of e raised to the power of the argument.

```
proc exp2(x: real): real
proc exp2(x: real(32)): real(32)
```

Returns the value of 2 raised to the power of the argument.

```
proc expm1(x: real): real
proc expm1(x: real(32)): real(32)
```

Returns one less than the value of e raised to the power of the argument.

```
proc floor(x: real): real
proc floor(x: real(32)): real(32)
```

Returns the value of the argument rounded down to the nearest integer.

```
const INFINITY: real(64);
```

Supplies the floating point constant for infinity.

```
proc isfinite(x: real): real
proc isfinite(x: real(32)): real(32)
```

Returns whether or not a particular value is finite.

```
proc isinf(x: real): real
proc isinf(x: real(32)): real(32)
```

Returns whether or not a particular value is infinite.

```
proc isnan(x: real): real
proc isnan(x: real(32)): real(32)
```

Returns whether or not a particular value is not a number.

```
proc lgamma(x: real): real
proc lgamma(x: real(32)): real(32)
```

Returns the natural logarithm of the absolute value of the gamma function of the argument.

```
proc log(x: real): real
proc log(x: real(32)): real(32)
```

Returns the natural logarithm of the argument. It is an error if the argument is less than or equal to zero.

```
proc log10(x: real): real
proc log10(x: real(32)): real(32)
```

Returns the base 10 logarithm of the argument. It is an error if the argument is less than or equal to zero.

```
proc log1p(x: real): real
proc log1p(x: real(32)): real(32)
```

Returns the natural logarithm of `x+1`. It is an error if `x` is less than or equal to -1.

```
proc log2(i: int(?w)): int(w)
proc log2(i: uint(?w)): uint(w)
proc log2(x: real): real
proc log2(x: real(32)): real(32)
```

Returns the base 2 logarithm of the argument. It is an error if the argument is less than or equal to zero.

```
proc mod(x: real(?w), y: real(w)): real(w)
proc mod(m: integral, n: integral)
proc mod(param m: integral, param n: integral) param
```

Computes the mod operator on the two numbers, defined as `mod(x,y)` $= x - y\lfloor \frac{m}{n} \rfloor$.

```
const NAN: real(64);
```

Supplies the floating point constant for not a number.

```
proc nearbyint(x: real): real
proc nearbyint(x: real(32)): real(32)
```

Returns the rounded integral value of the argument determined by the current rounding direction.

```
proc rint(x: real): real
proc rint(x: real(32)): real(32)
```

Returns the rounded integral value of the argument determined by the current rounding direction.

```
proc round(x: real): real
proc round(x: real(32)): real(32)
```

Returns the rounded integral value of the argument. Cases halfway between two integral values are rounded towards zero.

```
proc sin(x: real): real
proc sin(x: real(32)): real(32)
```

Returns the sine of the argument.

```
proc sinh(x: real): real
proc sinh(x: real(32)): real(32)
```

Returns the hyperbolic sine of the argument.

```
proc sqrt(x: real): real
proc sqrt(x: real(32)): real(32)
```

Returns the square root of the argument. It is an error if the argument is less than zero.

```
proc tan(x: real): real
proc tan(x: real(32)): real(32)
```

Returns the tangent of the argument.

```
proc tanh(x: real): real
proc tanh(x: real(32)): real(32)
```

Returns the hyperbolic tangent of the argument.

```
proc tgamma(x: real): real
proc tgamma(x: real(32)): real(32)
```

Returns the gamma function of the argument defined as

$$\int_0^\infty t^{x-1} e^{-t} dt$$

for the argument x.

```
proc trunc(x: real): real
proc trunc(x: real(32)): real(32)
```

Returns the nearest integral value to the argument that is not larger than the argument in absolute value.

32.1.2 Base

proc `ascii(s: `**string**`):` **int**

Returns the ASCII code number of the first letter in the argument `s`.

proc `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.

proc `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.

proc **complex**`.re:` **real**

Returns the real component of the complex number.

proc **complex**`.im:` **real**

Returns the imaginary component of the complex number.

proc **complex**`.=re(f: `**real**`)`

Sets the real component of the complex number to `f`.

proc **complex**`.=im(f: `**real**`)`

Sets the imaginary component of the complex number to `f`.

proc `exit(status: `**int**`)`

Exits the program with code `status`.

proc `halt()`

Exits the program and prints to standard error the location in the Chapel code of the call to `halt`.

proc `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.

proc **string**`.length:` **int**

Returns the number of characters in the base expression of type string.

proc `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 conversions.

proc `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 conversions.

proc **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.

proc `typeToString(`**type** `t)` **param** **:** **string**

Returns a string parameter that represents the name of the type `t`.

32.1.3 Types

`proc numBits(type t) param : int`

Returns the number of bits used to store the values of type `t`. This is implemented for all numeric types and fixed-width `bool` types. It is not implemented for default-width `bool`.

`proc numBytes(type t) param : int`

Returns the number of bytes used to store the values of type `t`. This is implemented for all numeric types and fixed-width `bool` types. It is not implemented for default-width `bool`.

`proc max(type t): t`

Returns the maximum value that can be stored in type `t`. This is implemented for all numeric types.

`proc min(type t): t`

Returns the minimum value that can be stored in type `t`. This is implemented for all numeric types.

32.2 Optional Modules

Optional modules can be used by a Chapel program via the `use` keyword (§12.4).

32.2.1 AdvancedIters

The `AdvancedIters` module contains several iterators that can be used to drive a `forall` loop by performing dynamic and adaptive splitting of a range's iterations.

`iter dynamic(r:range(?), chunkSize:int, numTasks:int=0) {`

This iterator is equivalent to the dynamic scheduling approach of OpenMP: Given an input range `r`, each task (of `numTasks`) is assigned chunks of size `chunkSize` from `r` (or the remaining iterations if there are fewer than `chunkSize`). This continues until there are no remaining iterations in `r`. If `numTasks` has the value 0, it will use the value indicated by `dataParTasksPerLocale` (§25.5).

`iter guided(r:range(?), numTasks:int = 0)`

This iterator is equivalent to the guided policy of OpenMP: Given an input range `r`, each task (of `numTasks`) is assigned chunks of variable size, until there are no remaining iterations in `r`. The size of each chunk is the number of unassigned iterations divided by the number of tasks, `numTasks`. The size decreases approximately exponentially to 1. The splitting strategy is therefore adaptive. If `numTasks` has the value 0, it will use the value indicated by `dataParTasksPerLocale` (§25.5).

`iter adaptive(r:range(?), numTasks:int = 0)`

This iterator implements a naive adaptive binary splitting work-stealing strategy: Initially the leader iterator distributes the range to split, `r`, evenly among the `numTasks` tasks. If `numTasks` has the value 0, it will use the value indicated by `dataParTasksPerLocale` (§25.5).

Then, each task performs adaptive splitting on its local sub-range's iterations. When a task exhausts its local iterations, it steals and splits from the range of another task (the victim). The splitting method on the local range and on the victim range is binary: i.e. the size of each chunk is computed as the number of unassigned iterations divided by 2. There are three stealing strategies that can be selected at compile time using the config param `methodStealing`:

- `methodStealing`=0 : Each task without work tries to steal from its neighbor range until it exhausts that range. Then the task continues with the next neighbor range, and so on until there is no more work. This is the default policy.
- `methodStealing`=1: Each task without work tries to steal once from its neighbor range, next from the following neighbor range and so on in a round-robin way until there is no more work.
- `methodStealing`=2: Similar to method 0, but now the splitting in the victim range is performed from its tail.

32.2.2 BitOps

The module `BitOps` defines routines that manipulate the bits of values of integral types.

```
proc bitPop(i: integral): int
```

Returns the number of bits set to one in the integral argument `i`.

```
proc 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.

```
proc 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.

```
proc 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.

32.2.3 Norm

The module `Norm` supports the computation of standard vector and matrix norms on Chapel arrays. The current interface is minimal and should be expected to grow and evolve over time.

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

An enumerated type indicating the different types of norms supported by this module: 1-norm, 2-norm, infinity norm and Frobenius norm, respectively.

```
proc norm(x: [], p: normType) where x.rank == 1 || x.rank == 2
```

Compute the norm indicated by `p` on the 1D or 2D array `x`.

```
proc norm(x: [])
```

Compute the default norm on array `x`. For a 1D array this is the 2-norm, for a 2D array, this is the Frobenius norm.

32.2.4 Random

The module `Random` supports the generation of pseudo-random values and streams of values. The current interface is minimal and should be expected to grow and evolve over time. In particular, we expect to support other pseudo-random number generation algorithms, more random value types (*e.g.*, int), and both serial and parallel iterators over the RandomStream class.

```
class RandomStream
proc RandomStream(seed: int(64), param parSafe: bool = true)
proc RandomStream(seedGenerator: SeedGenerator = SeedGenerator.currentTime,
                  param parSafe: bool = true)
```

Implements a pseudo-random stream of values based on a seed value. The current implementation generates the values using a linear congruential generator. In future versions of this module, the RandomStream class will offer a wider variety of algorithms for generating pseudo-random values.

To construct a RandomStream class, the seed may be explicitly passed. It must be an odd integer between 1 and $2^{46} - 1$. Alternatively, the RandomStream class can be constructed by passing a value of the enumerated type SeedGenerator to choose an algorithm to use to set the seed. If neither a seed nor a SeedGenerator value is passed to the RandomStream class, the seed will be initialized based on the current time in microseconds (rounded via modular arithmetic to the nearest odd integer between 1 and $2^{46} - 1$.

The parSafe parameter defaults to true and allows for safe use of this class by concurrent tasks. This can be overridden when calling methods to make them safe when called by concurrent tasks. This mechanism allows for lower overhead calls when there is no threat of concurrent calls, but correct calls when there is.

```
enum SeedGenerator { currentTime };
```

Values of this enumerated type may be used to choose a method for initializing the seed in the RandomStream class. The only value supported at present is `currentTime` which can be used to initialize the seed based on the current time in microseconds (rounded via modular arithmetic to the nearest odd integer between 1 and $2^{46} - 1$.

```
proc RandomStream.fillRandom(x:[?D], param parSafe = this.parSafe)
```

Fill the argument array, `x`, with the next |`D`| values of the pseudo-random stream in row-major order. The array must be an array of real(64), imag(64), or complex(128) elements. For complex arrays, each complex element is initialized with two values from the stream of random numbers.

```
proc RandomStream.skipToNth(in n: integral, param parSafe = this.parSafe)
```

Skips ahead or back to the `n`-th value in the random stream. The value of n is assumed to be positive, such that `n` == 1 represents the initial value in the stream.

```
proc RandomStream.getNext(param parSafe = this.parSafe): real
```

Returns the next value in the random stream as a real.

```
proc RandomStream.getNth(n: integral, param parSafe = this.parSafe): real
```

Returns the `n`-th value in the random stream as a real. Equivalent to calling `skipToNth(n)` followed by `getNext()`.

proc `fillRandom(x:[], initseed: int(64))`

A routine provided for convenience to support the functionality of the fillRandom method (above) without explicitly constructing an instance of the `RandomStream` class. This is useful for filling a single array or multiple arrays which require no coherence between them. The `initseed` argument corresponds to the `seed` member of the `RandomStream` class. If unspecified, the default for the class will be used.

32.2.5 Search

The `Search` module is designed to support standard search routines. The current interface is minimal and should be expected to grow and evolve over time.

proc `LinearSearch(Data: [?Dom], val): (bool, index(Dom))`

Searches through the pre-sorted array `Data` looking for the value `val` using a sequential linear search. Returns a tuple indicating (1) whether or not the value was found and (2) the location of the value if it was found, or the location where the value should have been if it was not found.

proc `BinarySearch(Data: [?Dom], val, in lo = Dom.low, in hi = Dom.high)`

Searches through the pre-sorted array `Data` looking for the value `val` using a sequential binary search. If provided, only the indices `lo` through `hi` will be considered, otherwise the whole array will be searched. Returns a tuple indicating (1) whether or not the value was found and (2) the location of the value if it was found, or the location where the value should have been if it was not found.

32.2.6 Sort

The `Sort` module is designed to support standard sorting routines. The current interface is minimal and should be expected to grow and evolve over time.

proc `BubbleSort(Data: [?Dom]) where Dom.rank == 1`

Sorts the 1D array `Data` in-place using a sequential bubble sort algorithm.

proc `InsertionSort(Data: [?Dom]) where Dom.rank == 1`

Sorts the 1D array `Data` in-place using a sequential insertion sort algorithm.

proc `MergeSort(Data: [?Dom], minlen=16) where Dom.rank == 1`

Sorts the 1D array `Data` using a parallel merge sort algorithm. The optional minlen argument is the minimum sized array that will be sorted using merge sort. If the size is less than minlen, the insertion sort algorithm will be used.

proc `QuickSort(Data: [?Dom], minlen=16) where Dom.rank == 1`

Sorts the 1D array `Data` in-place using a sequential implementation of the QuickSort algorithm. The optional minlen argument is the minimum sized array that will be sorted using QuickSort. If the size is less than minlen, the insertion sort algorithm will be used.

proc `SelectionSort(Data: [?Dom]) where Dom.rank == 1`

Sorts the 1D array `Data` in-place using a sequential selection sort algorithm.

32.2.7 Time

The module `Time` defines routines that query the system time and a record `Timer` that is useful for timing portions of code.

record `Timer`

A timer is used to time portions of code. Its semantics are similar to a stopwatch.

enum `TimeUnits { microseconds, milliseconds, seconds, minutes, hours };`

The enumeration TimeUnits defines units of time. These units can be supplied to routines in this module to specify the desired time units.

enum `Day { sunday=0, monday, tuesday, wednesday, thursday, friday, saturday };`

The enumeration Day defines the days of the week, with Sunday defined to be 0.

proc `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.

proc `getCurrentDayOfWeek(): Day`

Returns the current day of the week.

proc `getCurrentTime(unit: TimeUnits = TimeUnits.seconds):` **real**

Returns the elapsed time since midnight in the units specified.

proc `Timer.clear()`

Clears the elapsed time stored in the Timer.

proc `Timer.elapsed(unit: TimeUnits = TimeUnits.seconds):` **real**

Returns the cumulative elapsed time, in the units specified, between calls to `start` and `stop`. If the timer is running, the elapsed time since the last call to `start` is added to the return value.

proc `Timer.start()`

Starts the timer. It is an error to start a timer that is already running.

proc `Timer.stop()`

Stops the timer. It is an error to stop a timer that is not running.

proc `sleep(t:` **uint**`)`

Delays a task for `t` seconds.

32.2.8 UtilMath

The optional module `UtilMath` provides various math-related functions. It supplements the automatic `Math` module (§32.1.1).

```
proc divceilpos(m: int(?), n: int(?))
```

> Computes $\lceil \frac{n}{m} \rceil$ for two strictly positive integers $m > 0$, $n > 0$. For other m and n the behavior is undefined. This function is expected to be faster than `divceil`.

```
proc divfloorpos(m: int(?), n: int(?))
```

> Computes $\lfloor \frac{n}{m} \rfloor$ for two strictly positive integers $m > 0$, $n > 0$. For other m and n the behavior is undefined. This function is expected to be faster than `divfloor`.

33 Standard Distributions

The following table lists distributions standard to the Chapel language:

Distribution	**Module**	**Supported Domain Types**
`block`	`BlockDist`	Rectangular
`cyclic`	`CyclicDist`	Rectangular
`block-cyclic`	`BlockCycDist`	Rectangular
`replicated`	`ReplicatedDist`	Rectangular

> *Rationale*. Why supply any standard distributions? A main design goal of Chapel requires that the standard distributions be defined using the same mechanisms available to Chapel programmers wishing to define their own distributions or layouts (§29). That way there shouldn't be a necessary performance cost associated with user-defined domain maps. Nevertheless, distributions are an integral part of the Chapel language which would feel incomplete without a good set of standard distributions. It is hoped that many distributions will begin as user-defined domain maps and later become part of the standard set of distributions.

33.1 The Block Distribution

The standard Block distribution, defined in the module `BlockDist`, maps indices to locales by partitioning the indices into blocks according to a *bounding box* argument. It is parameterized by the rank and index type of the domains it supports. Thus domains of different ranks or different index types must be distributed with different Block distributions.

For Block distributions of rank d, given a bounding box

$$[l_1..h_1,\ \ldots,\ l_d..h_d]$$

and an array of target locales defined over the domain

$$[0..n_1\text{-}1,\ \ldots,\ 0..n_d-1]$$

then a Block distribution maps an index i to a locale by computing the *kth* component of an index j into the array of target locales from the *kth* component of i using the following formula:

$$j_k = \begin{cases} 0 & \text{if } i_k < l_k \\ \left\lfloor \dfrac{n_k(i_k - l_k)}{h_k - l_k + 1} \right\rfloor & \text{if } i_k \geq l_k \text{ and } i_k \leq h_k \\ n_k - 1 & \text{if } i_k > h_k \end{cases}$$

The Block class constructor is defined as follows:

```
proc Block(boundingBox: domain,
           targetLocales: [] locale = Locales,
           dataParTasksPerLocale = value in global config const of the same name,
           dataParIgnoreRunningTasks = value in global config const of the same name,
           dataParMinGranularity = value in global config const of the same name,
           param rank = boundingBox.rank,
           type idxType = boundingBox.dim(1).eltType)
```

The argument `boundingBox` is a non-distributed domain defining a bounding box used to partition the space of all indices across an array of target locales.

The argument `targetLocales` is a non-distributed array containing the target locales to which this distribution maps indices and data. The rank of `targetLocales` must match the rank of the distribution, or be one. If the rank of `targetLocales` is one, a greedy heuristic is used to reshape the array of target locales so that it matches the rank of the distribution and each dimension contains an approximately equal number of indices.

The arguments `dataParTasksPerLocale`, `dataParIgnoreRunningTasks`, and `dataParMinGranularity` set the knobs that are used to control intra-locale data parallelism for Block-distributed domains and arrays in the same way that the global configuration constants of these names control data parallelism for ranges and default-distributed domains and arrays §25.5.

The `rank` and `idxType` arguments are inferred from the `boundingBox` argument unless explicitly set.

> *Example*. The following code declares a Block distribution with a bounding box equal to the domain `Space` and declares an array, `A`, over a domain declared over this distribution. The computation in the `forall` loop sets each array element to the ID of the locale to which it is mapped.
>
> ```
> use BlockDist;
>
> const Space = [1..8, 1..8];
> const D: domain(2) dmapped Block(boundingBox=Space) = Space;
> var A: [D] int;
>
> forall a in A do
>   a = a.locale.id;
>
> writeln(A);
> ```
>
> When run on 6 locales, the output is:
>
> ```
> 0 0 0 0 1 1 1 1
> 0 0 0 0 1 1 1 1
> 0 0 0 0 1 1 1 1
> 2 2 2 2 3 3 3 3
> 2 2 2 2 3 3 3 3
> 2 2 2 2 3 3 3 3
> 4 4 4 4 5 5 5 5
> 4 4 4 4 5 5 5 5
> ```

33.2 The Cyclic Distribution

The standard Cyclic distribution, defined in the module `CyclicDist`, maps indices to locales in a round-robin pattern according to a *start index* argument. It is parameterized by the rank and index type of the domains it supports. Thus domains of different ranks or different index types must be distributed with different Cyclic distributions.

For cyclic distributions of rank d, given a start index

```
(s1, ..., sd)
```

and an array of target locales defined over the domain

```
[0..n1-1, ..., 0..nd-1]
```

then a Cyclic distribution maps an index i to a locale by computing the *kth* component of an index j into the array of target locales from the *kth* component of i using the following formula:

$$j_k = i_k - s_k \pmod{n_k}$$

The Cyclic class constructor is defined as follows:

```
proc Cyclic(startIdx,
            targetLocales: [] locale = Locales,
            dataParTasksPerLocale = value in global config const of the same name,
            dataParIgnoreRunningTasks = value in global config const of the same name,
            dataParMinGranularity = value in global config const of the same name,
            param rank: int = rank inferred from startIdx,
            type idxType = index type inferred from startIdx)
```

The argument `startIdx` is a tuple of integers defining an index that will be distributed to the first locale in `targetLocales`. For a single dimensional distribution `startIdx` can be an integer or a tuple with a single element.

The argument `targetLocales` is a non-distributed array containing the target locales to which this distribution maps indices and data. The rank of `targetLocales` must match the rank of the distribution, or be one. If the rank of `targetLocales` is one, a greedy heuristic is used to reshape the array of target locales so that it matches the rank of the distribution and each dimension contains an approximately equal number of indices.

The arguments `dataParTasksPerLocale`, `dataParIgnoreRunningTasks`, and `dataParMinGranularity` set the knobs that are used to control intra-locale data parallelism for Cyclic-distributed domains and arrays in the same way that the global configuration constants of these names control data parallelism for ranges and default-distributed domains and arrays §25.5.

The `rank` and `idxType` arguments are inferred from the `startIdx` argument unless explicitly set.

> *Example*. The following code declares a Cyclic distribution with a start index of `(1,1)` and declares an array, `A`, over a domain declared over this distribution. The computation in the `forall` loop sets each array element to the ID of the locale to which it is mapped.
>
> ```
> use CyclicDist;
>
> const Space = [1..8, 1..8];
> const D: domain(2) dmapped Cyclic(startIdx=Space.low) = Space;
> var A: [D] int;
>
> forall a in A do
>   a = a.locale.id;
>
> writeln(A);
> ```

When run on 6 locales, the output is:

```
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3
4 5 4 5 4 5 4 5
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3
4 5 4 5 4 5 4 5
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3
```

33.3 The Block-Cyclic Distribution

The standard Block-Cyclic distribution, defined in the module `BlockCycDist`, maps blocks of indices to locales in a round-robin pattern according to *block size* and *start index* arguments. It is parameterized by the rank and index type of the domains it supports. Thus domains of different ranks or different index types must be distributed with different instances of the BlockCyclic distributions.

For Block-Cyclic distributions of rank d, given a start index of

(s_1, ..., s_d)

and a block size of

(b_1, ..., b_d)

and an array of target locales defined over the domain

[0..n_1-1, ..., 0..n_d-1]

an index i is mapped to a locale by computing the *kth* component of an index j into the array of target locales from the *kth* component of i using the following formula:

$$j_k = ((i_k - s_k)/b_k) \pmod{n_k}$$

The Block-Cyclic class constructor is defined as follows:

```
proc BlockCyclic(startIdx: rank*idxType,
                 blocksize: rank*int(32),
                 targetLocales: [] locale = Locales,
                 param rank: int = rank inferred from startIdx,
                 type idxType = index type inferred from startIdx)
```

The argument `startIdx` is a tuple of integers defining an index that will be distributed to the first locale in `targetLocales`. For a single dimensional distribution `startIdx` can be an integer or a tuple with a single element.

The argument `blocksize` is a tuple of 32-bit integers defining the block size of indices that will be used in dealing out indices to the locales.

The argument `targetLocales` is a non-distributed array containing the target locales to which this distribution maps indices and data. The rank of `targetLocales` must match the rank of the distribution, or be one. If the rank of `targetLocales` is one, a greedy heuristic is used to reshape the array of target locales so that it matches the rank of the distribution and each dimension contains an approximately equal number of indices.

The `rank` and `idxType` arguments are inferred from the `startIdx` argument unless explicitly set.

> *Example*. The following code declares a Block-Cyclic distribution with a start index of `(1,1)` and a blocksize of `(2,3)` and declares an array, `A`, over a domain declared over this distribution. The computation in the `forall` loop sets each array element to the ID of the locale to which it is mapped.
>
> ```
> use BlockCycDist;
>
> const Space = [1..8, 1..8];
> const D: domain(2) dmapped BlockCyclic(startIdx=Space.low,
>                                        blocksize=(2,3))
>       = Space;
> var A: [D] int;
>
> forall a in A do
>   a = a.locale.id;
>
> writeln(A);
> ```
>
> When run on 6 locales, the output is:
>
> ```
> 0 0 0 1 1 1 0 0
> 0 0 0 1 1 1 0 0
> 2 2 2 3 3 3 2 2
> 2 2 2 3 3 3 2 2
> 4 4 4 5 5 5 4 4
> 4 4 4 5 5 5 4 4
> 0 0 0 1 1 1 0 0
> 0 0 0 1 1 1 0 0
> ```

33.4 The Replicated Distribution

The standard Replicated distribution is defined in the module `ReplicatedDist`. It causes a domain and its arrays to be replicated across the locales. An array receives a distinct set of elements – a *replicand* – allocated on each locale. In other words, mapping a domain with the replicated distribution gives it an implicit additional dimension – over the locales, making it behave as if there is one copy of its indices per locale.

Consistency among the replicands is *not* preserved automatically. That is, changes to one replicand of an array are never propagated to the other replicands by the distribution implementation.

Replication is observable only in some cases, as described below.

> *Future*. This behavior may change in the future. In particular, we are considering changing it so that replication is never observable. For example, only the local replicand would be accessed in all cases.

Replication over locales is observable in the following cases:

- when iterating over a replicated domain or array
- when printing them out with write() and similar functions
- when zippering and the replicated domain/array is the first among the zippered items
- when assigning into the replicated array (each replicand gets a copy)
- when inquiring about the domain's `numIndices` or the array's `numElements`

Replication is not observable, i.e., only the replicand *on the current locale* is accessed in the following cases:

- when examining certain domain properties: `dim(d)`, `dims()`, `low`, `high`, `stride`; but not `numIndices`
- when indexing into an array
- when slicing an array
- when zippering and the first zippered item is not replicated
- when assigning to a non-replicated array, i.e. the replicated array is on the right-hand side of the assignment
- when there is only a single locale (trivially: there is only one replicand in this case)

E.g. when iterating, the number of iterations will be (*the number of locales involved*) times (*the number of iterations over this domain if it were distributed with the default distribution*). Note that serial iteration visits the domain indices and array elements of all the replicands *from the current locale*.

> *Example*. The following code declares a replicated domain, `Drepl`, and a replicated array, `Arepl`, and accesses it in different ways:
>
> ```
> const Dbase = [1..5];
> const Drepl: domain(1) dmapped ReplicatedDist() = Dbase;
> var Abase: [Dbase] int;
> var Arepl: [Drepl] int;
>
> // only the current locale's replicand is accessed
> Arepl[3] = 4;
>
> // these iterate over Dbase, so
> // only the current locale's replicand is accessed
> forall (b,r) in (Abase,Arepl) b = r;
> Abase = Arepl;
>
> // these iterate over Drepl;
> // each replicand of Arepl will be zippered against
> // (and copied from) the entire Abase
> forall (r,b) in (Arepl,Abase) r = b;
> Arepl = Abase;
>
> // sequential zippering will detect difference in sizes
> // (if multiple locales)
> for (b,r) in (Abase,Arepl) ... // error
> for (r,b) in (Arepl,Abase) ... // error
> ```

34 Standard Layouts

This chapter is forthcoming.

A Collected Lexical and Syntax Productions

This appendix collects the syntax productions listed throughout the specification. There are no new syntax productions in this appendix. The productions are listed both alphabetically and in depth-first order for convenience.

A.1 Alphabetical Lexical Productions

binary-digit: *one of*
0 1

binary-digits:
binary-digit
binary-digit binary-digits

bool-literal: *one of*
true false

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

digits:
digit
digit digits

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

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

hexadecimal-digit: *one of*
0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

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

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

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

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

integer-literal:
digits
0x *hexadecimal-digits*
0X *hexadecimal-digits*
0b *binary-digits*
0B *binary-digits*

legal-identifier-char:
letter
digit
_
$

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

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

real-literal:
digits$_{opt}$. *digits exponent-part*$_{opt}$
digits .$_{opt}$ *exponent-part*

sign: *one of*
+ −

simple-escape-character: *one of*
\’ \” \? \a \b \f \n \r \t \v

single-quote-delimited-characters:
string-character single-quote-delimited-characters$_{opt}$
” *single-quote-delimited-characters*$_{opt}$

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

string-literal:
” *double-quote-delimited-characters*$_{opt}$ ”
’ *single-quote-delimited-characters*$_{opt}$ ’

A.2 Alphabetical Syntax Productions

aligned-range-expression:
range-expression **align** *expression*

argument−list:
(*formals*$_{opt}$)

array−alias−declaration:
identifier reindexing−expression$_{opt}$ => *array−expression* ;

array−expression:
expression

array−type:
[*domain−expression*] *type−specifier*

assignment−operator: one of
= += −= ∗= /= %= ∗∗= &= |= ˆ= &&= ||= <<= >>=

assignment−statement:
lvalue−expression assignment−operator expression

associative−domain−type:
domain (*associative−index−type*)
domain (*enum−type*)
domain (**opaque**)

associative−index−type:
type−specifier

atomic−statement:
atomic *statement*

base−domain−type:
rectangular−domain−type
associative−domain−type

begin−statement:
begin *statement*

binary−expression:
expression binary−operator expression

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

block−statement:
{ *statements*$_{opt}$ }

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

call−expression:
expression (*named−expression−list*)
expression [*named−expression−list*]
parenthesesless−function−identifier

cast−expression:
expression : *type−specifier*

class−declaration−statement:
external−class−declaration−statement
simple−class−declaration−statement

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

class−name:
identifier

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

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

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

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

cobegin−statement:
cobegin *block−statement*

coforall−statement:
coforall *index−var−declaration* **in** *iteratable−expression* **do** *statement*
coforall *index−var−declaration* **in** *iteratable−expression block−statement*
coforall *iteratable−expression* **do** *statement*
coforall *iteratable−expression block−statement*

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

config−or−extern: *one of*
config extern

constructor−call−expression:
new *class−name* (*argument−list*)

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

counted−range−expression:
range−expression # *expression*

dataparallel−type:
range−type
domain−type
mapped−domain−type
array−type
index−type

default−expression:
= *expression*

delete−statement:
delete *expression* ;

dmap−value:
expression

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

domain−assignment−expression:
domain−name = *domain−expression*

domain−expression:
domain−literal
domain−name
domain−assignment−expression
domain−striding−expression
domain−slice−expression

domain−literal:
[*range−expression−list*]

domain−name:
identifier

domain−slice−expression:
domain−slice−expression [*domain−expression*]
domain−slice−expression (*domain−expression*)

domain−striding−expression:
domain−expression **by** *expression*

domain−type:
base−domain−type
simple−subdomain−type
sparse−subdomain−type

else−part:
else *statement*

empty−statement:
;

enum−constant−expression:
enum−type . *identifier*

enum−constant−list:
enum−constant
enum−constant , *enum−constant−list*$_{opt}$

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

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

enum−type:
identifier

exported−procedure−declaration−statement:
export *external−name*$_{opt}$ **proc** *function−name argument−list return−intent*$_{opt}$ *return−type*$_{opt}$
function−body

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

expression−statement:
expression ;

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

external−class−declaration−statement:
extern *simple−class−declaration−statement*

external−procedure−declaration−statement:
extern *external−name*$_{opt}$ **proc** *function−name argument−list return−intent*$_{opt}$ *return−type*$_{opt}$

external−record−declaration−statement:
extern *simple−record−declaration−statement*

external−type−alias−declaration−statement:
extern type *type−alias−declaration−list* ;

field−access−expression:
receiver−clause$_{opt}$ *identifier*

for−expression:
for *index−var−declaration* **in** *iteratable−expression* **do** *expression*
for *iteratable−expression* **do** *expression*

for−statement:
for *index−var−declaration* **in** *iteratable−expression* **do** *statement*
for *index−var−declaration* **in** *iteratable−expression block−statement*
for *iteratable−expression* **do** *statement*
for *iteratable−expression block−statement*

forall−expression:
forall *index−var−declaration* **in** *iteratable−expression* **do** *expression*
forall *iteratable−expression* **do** *expression*
[*index−var−declaration* **in** *iteratable−expression*] *expression*
[*iteratable−expression*] *expression*

forall−statement:
forall *index−var−declaration* **in** *iteratable−expression* **do** *statement*
forall *index−var−declaration* **in** *iteratable−expression block−statement*
forall *iteratable−expression* **do** *statement*
forall *iteratable−expression block−statement*
[*index−var−declaration* **in** *iteratable−expression*] *statement*
[*iteratable−expression*] *statement*

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

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

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

formals:
formal
formal , *formals*

function−body:
block−statement
return−statement

function−name:
identifier
operator−name

identifier−list:
identifier
identifier , *identifier−list*
tuple−grouped−identifier−list
tuple−grouped−identifier−list , *identifier−list*

if−expression:
if *expression* **then** *expression* **else** *expression*
if *expression* **then** *expression*

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

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

init−part:
= *expression*

initialization−part:
= *expression*

integer−parameter−expression:
expression

iteratable−call−expression:
call−expression

iteratable−expression:
expression

iterator−body:
block−statement
yield−statement

iterator−declaration−statement:
iter *iterator−name* *argument−list*$_{opt}$ *return−intent*$_{opt}$ *return−type*$_{opt}$ *where−clause*$_{opt}$
iterator−body

iterator−name:
identifier

label−statement:
label *identifier* *statement*

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

linkage-specifier:
inline
export *identifier*$_{opt}$
extern

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

locale-access-expression:
expression . **locale**

locale-type:
locale

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

mapped-domain-expression:
domain-expression **dmapped** *dmap-value*

mapped-domain-type:
domain-type **dmapped** *dmap-value*

member-access-expression:
field-access-expression
method-call-expression

method-call-expression:
receiver-clause$_{opt}$ *expression* (*named-expression-list*)
receiver-clause$_{opt}$ *expression* [*named-expression-list*]
receiver-clause$_{opt}$ *parenthesesless-function-identifier*

method-declaration-statement:
linkage-specifier$_{opt}$ *proc-or-iter param-clause*$_{opt}$ *type-binding function-name argument-list*$_{opt}$
return-intent$_{opt}$ *return-type*$_{opt}$ *where-clause*$_{opt}$ *function-body*

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

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

module-identifier-list:
module-identifier
module-identifier . *module-identifier-list*

module-identifier:
  identifier

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

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

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

named-expression:
  expression
  identifier = *expression*

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

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

param-clause:
  param

param-for-statement:
  for param *identifier* **in** *param-iteratable-expression* **do** *statement*
  for param *identifier* **in** *param-iteratable-expression block-statement*

param-iteratable-expression:
  range-literal
  range-literal **by** *integer-literal*

parenthesesless-function-identifier:
  identifier

parenthesized-expression:
  (*expression*)

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

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

proc−or−iter:
proc
iter

procedure−declaration−statement:
linkage−specifier$_{opt}$ **proc** *function−name argument−list*$_{opt}$ *return−intent*$_{opt}$ *return−type*$_{opt}$ *where−clause*$_{opt}$ *function−body*

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

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

range−expression:
expression
strided−range−expression
counted−range−expression
aligned−range−expression
sliced−range−expression

range−literal:
expression .. *expression*
expression ..
.. *expression*
..

range−type:
range (*named−expression−list*)

receiver−clause:
expression .

record−declaration−statement:
simple−record−declaration−statement
external−record−declaration−statement

record−inherit−list:
: *record−type−list*

record−statement−list:
record−statement
record−statement record−statement−list

record−statement:
type−declaration−statement
method−declaration−statement
variable−declaration−statement
empty−statement

record−type−list:
record−type
record−type , *record−type−list*

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

rectangular−domain−type:
domain (*named−expression−list*)

reduce−expression:
reduce−scan−operator **reduce** *expression*
class−type **reduce** *expression*

reduce−scan−operator: *one of*
+ ∗ && || & | ˆ **min max minloc maxloc**

reindexing−expression:
[*domain−expression*]

remote−variable−declaration−statement:
on *expression variable−declaration−statement*

return−intent: *one of*
var const param type

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

return−type:
: *type−specifier*

scan−expression:
reduce−scan−operator **scan** *expression*
class−type **scan** *expression*

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

serial−statement:
serial *expression* **do** *statement*
serial *expression block−statement*

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

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

simple−subdomain−type:
subdomain (*domain−expression*)

single−type:
single *type−specifier*

sliced−range−expression:
range−expression (*range−expression*)
range−expression [*range−expression*]

sparse−subdomain−type:
sparse subdomain$_{opt}$ (*domain−expression*)

statement:
block−statement
expression−statement
assignment−statement
swap−statement
conditional−statement
select−statement
while−do−statement
do−while−statement
for−statement
label−statement
break−statement
continue−statement
param−for−statement
use−statement
type−select−statement
empty−statement
return−statement
yield−statement
module−declaration−statement
procedure−declaration−statement
external−procedure−declaration−statement
exported−procedure−declaration−statement
iterator−declaration−statement
method−declaration−statement
type−declaration−statement
variable−declaration−statement
remote−variable−declaration−statement
on−statement
cobegin−statement
coforall−statement
begin−statement
sync−statement
serial−statement
atomic−statement
forall−statement
delete−statement

statements:
statement
statement statements

step−expression:
expression

strided−range−expression:
range−expression **by** *step−expression*

structured−type:
class−type
record−type
union−type
tuple−type

swap−operator:
<=>

swap−statement:
lvalue−expression swap−operator lvalue−expression

sync−statement:
sync *statement*
sync *block−statement*

sync−type:
sync *type−specifier*

synchronization−type:
sync−type
single−type

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

tuple−expression:
(*expression* , *expression−list*)

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

tuple−type:
(*type−specifier* , *type−list*)

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

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

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

type−binding:
identifier .

type−declaration−statement:
enum−declaration−statement
class−declaration−statement
record−declaration−statement
union−declaration−statement
type−alias−declaration−statement

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

type-part:
: *type-specifier*

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

type-specifier:
primitive-type
enum-type
locale-type
structured-type
dataparallel-type
synchronization-type

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

type-when-statements:
type-when-statement
type-when-statement type-when-statements

unary-expression:
unary-operator expression

unary-operator: *one of*
+ -~ !

union-declaration-statement:
extern$_{opt}$ **union** *identifier* { *union-statement-list* }

union-statement-list:
union-statement
union-statement union-statement-list

union-statement:
type-declaration-statement
procedure-declaration-statement
iterator-declaration-statement
variable-declaration-statement
empty-statement

union-type:
identifier

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

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

variable−declaration−list:
variable−declaration
variable−declaration , *variable−declaration−list*

variable−declaration−statement:
config−or−extern$_{opt}$ *variable−kind variable−declaration−list* ;

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

variable−expression:
identifier

variable−kind: *one of*
param const var

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

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

where−clause:
where *expression*

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

yield−statement:
yield *expression* ;

A.3 Depth-First Lexical Productions

bool−literal: *one of*
true false

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

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

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

legal−identifier−char:
letter
digit
_
$

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

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

real−literal:
digits$_{opt}$. *digits exponent−part*$_{opt}$
digits .$_{opt}$ *exponent−part*

digits:
digit
digit digits

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

sign: *one of*
+ −

integer−literal:
digits
0x *hexadecimal−digits*
0X *hexadecimal−digits*
0b *binary−digits*
0B *binary−digits*

hexadecimal−digits:
hexadecimal−digit
hexadecimal−digit hexadecimal−digits

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

binary−digits:
binary−digit
binary−digit binary−digits

binary−digit: *one of*
0 1

string−literal:
” *double−quote−delimited−characters*$_{opt}$ ”
’ *single−quote−delimited−characters*$_{opt}$ ’

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

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

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

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

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

A.4 Depth-First Syntax Productions

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

module−identifier:
identifier

block−statement:
{ *statements*$_{opt}$ }

statements:
statement
statement statements

statement:
block−statement
expression−statement
assignment−statement
swap−statement
conditional−statement
select−statement
while−do−statement
do−while−statement
for−statement
label−statement
break−statement
continue−statement
param−for−statement
use−statement
type−select−statement
empty−statement
return−statement
yield−statement
module−declaration−statement
procedure−declaration−statement
external−procedure−declaration−statement
exported−procedure−declaration−statement
iterator−declaration−statement

method−declaration−statement
type−declaration−statement
variable−declaration−statement
remote−variable−declaration−statement
on−statement
cobegin−statement
coforall−statement
begin−statement
sync−statement
serial−statement
atomic−statement
forall−statement
delete−statement

expression−statement:
expression ;

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

literal−expression:
bool−literal
integer−literal
real−literal
imaginary−literal
string−literal
range−literal
domain−literal

range−literal:
expression .. *expression*
expression ..
.. *expression*

..

domain−literal:
[*range−expression−list*]

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

range−expression:
expression
strided−range−expression
counted−range−expression
aligned−range−expression
sliced−range−expression

strided−range−expression:
range−expression **by** *step−expression*

step−expression:
expression

counted−range−expression:
range−expression # *expression*

aligned−range−expression:
range−expression **align** *expression*

sliced−range−expression:
range−expression (*range−expression*)
range−expression [*range−expression*]

variable−expression:
identifier

enum−constant−expression:
enum−type . *identifier*

enum−type:
identifier

call−expression:
expression (*named−expression−list*)
expression [*named−expression−list*]
parenthesesless−function−identifier

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

named−expression:
expression
identifier = *expression*

parenthesesless−function−identifier:
identifier

iteratable−call−expression:
call−expression

member−access−expression:
field−access−expression
method−call−expression

field−access−expression:
receiver−clause$_{opt}$ *identifier*

receiver−clause:
expression .

method−call−expression:
receiver−clause$_{opt}$ *expression* (*named−expression−list*)
receiver−clause$_{opt}$ *expression* [*named−expression−list*]
receiver−clause$_{opt}$ *parenthesesless−function−identifier*

constructor−call−expression:
new *class−name* (*argument−list*)

class−name:
identifier

argument−list:
(*formals*$_{opt}$)

formals:
formal
formal , *formals*

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

default−expression:
= *expression*

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

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

identifier−list:
identifier
identifier , *identifier−list*
tuple−grouped−identifier−list
tuple−grouped−identifier−list , *identifier−list*

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

type−specifier:
primitive−type
enum−type
locale−type
structured−type
dataparallel−type
synchronization−type

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

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

integer−parameter−expression:
expression

locale−type:
locale

structured−type:
class−type
record−type
union−type
tuple−type

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

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

union−type:
identifier

tuple−type:
(*type−specifier* , *type−list*)

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

dataparallel−type:
range−type
domain−type
mapped−domain−type
array−type
index−type

range−type:
range (*named−expression−list*)

domain−type:
base−domain−type
simple−subdomain−type
sparse−subdomain−type

base−domain−type:
rectangular−domain−type
associative−domain−type

rectangular−domain−type:
domain (*named−expression−list*)

associative−domain−type:
domain (*associative−index−type*)
domain (*enum−type*)
domain (**opaque**)

associative−index−type:
type−specifier

simple−subdomain−type:
subdomain (*domain−expression*)

domain−expression:
domain−literal
domain−name
domain−assignment−expression
domain−striding−expression
domain−slice−expression

domain−name:
identifier

domain−assignment−expression:
domain−name = *domain−expression*

domain−striding−expression:
domain−expression **by** *expression*

domain−slice−expression:
domain−slice−expression [*domain−expression*]
domain−slice−expression (*domain−expression*)

sparse−subdomain−type:
sparse $\textbf{subdomain}_{opt}$ (*domain−expression*)

mapped-domain-type:
  domain-type **dmapped** *dmap-value*

dmap-value:
  expression

array-type:
  [*domain-expression*] *type-specifier*

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

synchronization-type:
  sync-type
  single-type

sync-type:
  sync *type-specifier*

single-type:
  single *type-specifier*

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

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

cast-expression:
  expression : *type-specifier*

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

parenthesized-expression:
  (*expression*)

unary-expression:
  unary-operator expression

unary-operator: *one of*
  + − ˜ !

binary-expression:
  expression binary-operator expression

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

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

if−expression:
if *expression* **then** *expression* **else** *expression*
if *expression* **then** *expression*

for−expression:
for *index−var−declaration* **in** *iteratable−expression* **do** *expression*
for *iteratable−expression* **do** *expression*

forall−expression:
forall *index−var−declaration* **in** *iteratable−expression* **do** *expression*
forall *iteratable−expression* **do** *expression*
[*index−var−declaration* **in** *iteratable−expression*] *expression*
[*iteratable−expression*] *expression*

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

iteratable−expression:
expression

reduce−expression:
reduce−scan−operator **reduce** *expression*
class−type **reduce** *expression*

reduce−scan−operator: *one of*
+ ∗ && || & | ˆ **min max minloc maxloc**

scan−expression:
reduce−scan−operator **scan** *expression*
class−type **scan** *expression*

module−access−expression:
module−identifier−list . *identifier*

module−identifier−list:
module−identifier
module−identifier . *module−identifier−list*

tuple−expression:
(*expression* , *expression−list*)

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

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

locale−access−expression:
expression . **locale**

mapped−domain−expression:
domain−expression **dmapped** *dmap−value*

assignment-statement:
lvalue-expression assignment-operator expression

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

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

swap-operator:
<=>

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

else-part:
else *statement*

select-statement:
select *expression* { *when-statements* }

when-statements:
when-statement
when-statement when-statements

when-statement:
when *expression-list* **do** *statement*
when *expression-list block-statement*
otherwise *statement*

while-do-statement:
while *expression* **do** *statement*
while *expression block-statement*

do-while-statement:
do *statement* **while** *expression* ;

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

label-statement:
label *identifier statement*

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

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

param-for-statement:
for param *identifier* **in** *param-iteratable-expression* **do** *statement*
for param *identifier* **in** *param-iteratable-expression block-statement*

param-iteratable-expression:
range-literal
range-literal **by** *integer-literal*

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

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

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

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

type-when-statements:
type-when-statement
type-when-statement type-when-statements

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

empty-statement:
;

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

yield-statement:
yield *expression* ;

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

procedure-declaration-statement:
linkage-specifier$_{opt}$ **proc** *function-name argument-list*$_{opt}$ *return-intent*$_{opt}$ *return-type*$_{opt}$ *where-clause*$_{opt}$
function-body

linkage-specifier:
inline
export *identifier*$_{opt}$
extern

function-name:
identifier
operator-name

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

return−intent: *one of*
var const param type

return−type:
: *type−specifier*

where−clause:
where *expression*

function−body:
block−statement
return−statement

external−procedure−declaration−statement:
extern *external−name*$_{opt}$ **proc** *function−name argument−list return−intent*$_{opt}$ *return−type*$_{opt}$

exported−procedure−declaration−statement:
export *external−name*$_{opt}$ **proc** *function−name argument−list return−intent*$_{opt}$ *return−type*$_{opt}$
function−body

iterator−declaration−statement:
iter *iterator−name argument−list*$_{opt}$ *return−intent*$_{opt}$ *return−type*$_{opt}$ *where−clause*$_{opt}$
iterator−body

iterator−name:
identifier

iterator−body:
block−statement
yield−statement

method−declaration−statement:
linkage−specifier$_{opt}$ *proc−or−iter param−clause*$_{opt}$ *type−binding function−name argument−list*$_{opt}$
return−intent$_{opt}$ *return−type*$_{opt}$ *where−clause*$_{opt}$ *function−body*

proc−or−iter:
proc
iter

param−clause:
param

type−binding:
identifier .

type−declaration−statement:
enum−declaration−statement
class−declaration−statement
record−declaration−statement
union−declaration−statement
type−alias−declaration−statement

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

enum−constant−list:
enum−constant
enum−constant , *enum−constant−list*$_{opt}$

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

init−part:
= *expression*

class−declaration−statement:
external−class−declaration−statement
simple−class−declaration−statement

external−class−declaration−statement:
extern *simple−class−declaration−statement*

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

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

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

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

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

record−declaration−statement:
simple−record−declaration−statement
external−record−declaration−statement

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

record−inherit−list:
: *record−type−list*

record−type−list:
record−type
record−type , *record−type−list*

record−statement−list:
record−statement
record−statement record−statement−list

record−statement:
  type−declaration−statement
  method−declaration−statement
  variable−declaration−statement
  empty−statement

external−record−declaration−statement:
  extern *simple−record−declaration−statement*

union−declaration−statement:
  extern$_{opt}$ **union** *identifier* { *union−statement−list* }

union−statement−list:
  union−statement
  union−statement union−statement−list

union−statement:
  type−declaration−statement
  procedure−declaration−statement
  iterator−declaration−statement
  variable−declaration−statement
  empty−statement

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

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

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

external−type−alias−declaration−statement:
  extern type *type−alias−declaration−list* ;

variable−declaration−statement:
  config−or−extern$_{opt}$ *variable−kind variable−declaration−list* ;

config−or−extern: *one of*
  config extern

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
  array−alias−declaration

initialization−part:
= *expression*

type−part:
: *type−specifier*

array−alias−declaration:
identifier reindexing−expression$_{opt}$ => *array−expression* ;

reindexing−expression:
[*domain−expression*]

array−expression:
expression

remote−variable−declaration−statement:
on *expression variable−declaration−statement*

on−statement:
on *expression* **do** *statement*
on *expression block−statement*

cobegin−statement:
cobegin *block−statement*

coforall−statement:
coforall *index−var−declaration* **in** *iteratable−expression* **do** *statement*
coforall *index−var−declaration* **in** *iteratable−expression block−statement*
coforall *iteratable−expression* **do** *statement*
coforall *iteratable−expression block−statement*

begin−statement:
begin *statement*

sync−statement:
sync *statement*
sync *block−statement*

serial−statement:
serial *expression* **do** *statement*
serial *expression block−statement*

atomic−statement:
atomic *statement*

forall−statement:
forall *index−var−declaration* **in** *iteratable−expression* **do** *statement*
forall *index−var−declaration* **in** *iteratable−expression block−statement*
forall *iteratable−expression* **do** *statement*
forall *iteratable−expression block−statement*
[*index−var−declaration* **in** *iteratable−expression*] *statement*
[*iteratable−expression*] *statement*

delete−statement:
delete *expression* ;

Index

`#`, 68
`&`, 62
`&&`, 65
`&&=`, 73
`&=`, 73
`*`, 59
`*` tuples, 103
`**`, 62
`**=`, 73
`*=`, 73
`+`, 58, 68
`+` (unary), 57
`+=`, 73
`-`, 59
`-` (unary), 58
`-=`, 73
`...` tuple expansion, 109
`/`, 60
`/=`, 73
`<`, 66
`<<`, 64
`<<=`, 73
`<=`, 66
`=`, 73
`==`, 67
`=>`, 172
`>`, 66
`>=`, 66
`>>`, 64
`>>=`, 73
`?`, 53
`%`, 61
`%=`, 73
`~`, 62
`^`, 63
`^=`, 73

actual argument, 89, 90
argument passing
    arrays, 171
    domains, 160
arrays, 167
    arguments to functions, 171
    as formal arguments, 171
    assignment, 169
    association to domains, 173
    count operator, 171
    default values, 168
    domain maps, 168, 219
    indexing, 168
    initialization, 167
    iteration, 169
    literals, lack thereof, 167
    predefined functions, 174
    promotion, 172
    runtime representation, 168
    slicing, 170
    sparse, 173
    types, 167
    values, 167
assignment, 73
    tuples, 105
associative domains
    default values, 157
    initializations, 157
    literals, lack thereof, 157
    types, 156
    values, 157
`atomic`, 202
atomic statement, 202
atomic transactions, 202
automatic modules, 235
    Base, 240
    Math, 235

`begin`, 195
block, 72
Block distribution, 247
Block-Cyclic distribution, 250
`bool`, 32
`break`, 77
`by`, 68, 145
    on rectangular domains, 162

call site, 89
case sensitivity, 25
casts, 54
`class`, 113
class type, 114
class value, 114

classes, 113
  allocation, 133
  arguments, 134
  assignment, 122, 134
  class type, 114
  class value, 114
  comparison, 135
  constructors, 118
    compiler-generated, 119
    user-defined, 118
  declarations, 113
  default initialization, 118
  delete, 123
  derived class, 116
  field access, 120
  fields, 115
  generic, 184
  getter method, 120
  implicit conversion, 123
  indexing, 122
  inheritance, 116
    multiple, 117
  initialization, 118
    default, 118
  instances, 113
  iterating, 122
  method calls, 121
  methods, 115
  nested classes, 115
  new, 118
  nil, 117
  overriding, 134
  receiver, 120, 121
  shadowing, 134
`cobegin`, 199
`coforall`, 200
coforall loops, 200
comments, 25
compiler diagnostics
  user-defined, 188
compiler errors
  user-defined, 188
compiler warnings
  user-defined, 188
compiler-generated constructors, 119
compilerError, 188
compilerWarning, 188
complex
  casts from tuples, 54
`complex`, 33
conditional expression, 69
conditional statement, 74
  dangling else, 74
`config`, 44
`const`, 44
constants, 44
  compile-time, 43
  configuration, 44
  in classes or records, 185
  runtime, 44
constructors, 118
  compiler-generated, 119
    for generic classes or records, 187
  for derived classes, 116
  type constructors, 186
  user-defined, 118
    for generic classes or records, 187
`continue`, 77
conversions
  bool, 46
    in a statement, 47
  class, 49, 123
  enumerated types, 46
  enumeration, 48
  explicit, 47
    class, 49
    enumeration, 48
    numeric, 47
    record, 49
  implicit, 45
    allowed types, 45
    bool, 46, 47
    class, 123
    enumerated types, 46
    numeric, 46
    occurs at, 45
    parameter, 47
    record, 133
  numeric, 46, 47
    parameter, 47
  record, 49, 133
  source type, 45
  target type, 45
Cyclic distribution, 248

data parallelism, 205
  and evaluation order, 209
  forall, 205
  forall expressions, 206
  knobs for default data parallelism, 211
  reductions, 209
  scans, 209

default initialization
    classes, 118
    variables, 40
default values, 93
delete
    classes, 123
derived class, 116
distribution, 217
distributions
    standard, 247
        Block, 247
        Block-Cyclic, 250
        Cyclic, 248
        Replicated, 251
dmap value, 217
dmapped clause, 217
domain maps, 217
    distribution, 217
    dmap value, 217
    dmapped clause, 217
    domain assignment, 220
    for arrays, 219
    layout, 217
domains, 153
    adding indices, 163
    as arguments to functions, 160
    assignment, 161, 220
    association to arrays, 173
    associative, 156
        values, 157
    base, 154
    count operator, 163
    enumerated, 157
    index types, 160
    iteration, 160
    opaque, 157
        literals, 158
        values, 158
    predefined functions, 164
    promotion, 161
    rectangular
        default value, 156
        literals, 155
    removing indices, 163
    slicing, 162
    sparse, 159
        types, 159
        values, 159
    striding, 162
    values
        rectangular, 155
dynamic dispatch, 117

`else`, 69, 74
`enumerated`, 182
enumerated domains
    types, 157
enumerated types, 34
execution environment, 214
explicit return type, 97
exploratory programming, 87
expression
    as a statement, 72
expression statement, 72
expressions
    associativity, 54
    call, 52
    cast, 54
    conditional, 69
    enumeration constant, 52
    indexing, 53
    literal, 51
    lvalue, 54
    member access, 53
    operator, 57
    parenthesized, 52
    precedence, 54
    variable, 52

field access, 120, 130
fields, 115, 127
    generic, 184
    parameter, 185
    type alias, 184
    variable and constant, without types, 185
file type, 191
    methods, 191
    standard files stdin, stdout, stderr, 192
`for`, 69, 76, 77
for expressions
    and conditional expressions, 69
for loops, 76
    parameters, 77
`forall`, 205
forall expressions, 206
    and conditional expressions, 207
forall loops, 205
formal argument, 89, 92
formal arguments
    array types, 183
    defaults, 93
    generic types, 181

    naming, 92
    queried types, 180
    rectangular arrays, 171
    without types, 180
function calls, 52, 89
function overloading, 98
function resolution, 98
functions, 89
    actual argument, 89, 90
    argument intents, 93
    as lvalues, 95
    as parameters, 96
    as types, 96
    call site, 89
    candidates, 99
    default argument values, 93
    formal argument, 89, 92
    functions without parentheses, 92
    generic, 179
    iterators, 175
    most specific, 100
    named arguments, 92
    nested, 98
    operator, 89
    overloading, 98
    parameter function, 96
    procedure, 89
    procedure definition, 90
    resolution, 98
    `return` types, 97
    setter argument, 96
    variable functions, 95
    variable number of arguments, 94
    visible, 99

generics
    classes, 184
    constructors
        compiler-generated, 187
        user-defined, 187
    fields, 184
    function visibility, 183
    functions, 179
    instantiated type, 186
    methods, 186
    records, 184
    type constructor, 186
    types, 184
    `where`, 188
getter method, 120

`here`, 214

identifiers, 26
`if`, 69, 74
`imaginary`, 33
implicit modules, 84
implicit return type, 98
`in`, 93
indexing, 53
    arrays, 168
    rectangular arrays, 168
inheritance, 116
    records, 127, 134
initialization
    classes, 118
        default, 118
    records, 129
`inout`, 94
`int`, 32
`integral`, 182
intents, 93
    blank, 93
    `in`, 93
    `inout`, 94
    `out`, 94
    `param`, 180
    `type`, 179
interoperability, 227
    overview, 227
    sharing, 229
`isFull`, 198
iteration
    over arrays, 169
iterator calls, 176
iterator definitions, 175
iterators, 175
    and arrays, 176
    and generics, 176
    calls, 176
    definition, 175
    parallel, 177
    recursive, 177
    `yield`, 175

keywords, 26

`label`, 77
layout, 217
`let`, 68
lexical structure, 25
    braces, 29
    brackets, 29
    case sensitivity, 25

  - comment, 25
  - identifier, 26
  - keyword, 26
  - literal, 27
  - operator, 29
  - parentheses, 29
  - punctuation, 29
  - token, 25
- literal expressions, 51
- literals
  - primitive type, 27
- local, 213
- `locale`, 213, 215
- Locales
  - methods, 213
- `Locales`, 214
- locales, 213
- lvalue, 54

- `main`, 23, 87
- mapped
  - domain maps, 217
- member access, 53
- memory consistency model, 225
- memory management, 123
- method calls, 121, 131
- `module`, 83
- modules, 23, 83
  - and files, 83
  - explicitly named, 85
  - implicit, 84
  - initialization, 86
  - nested, 84
  - standard, 235
  - using, 78, 85
- multiple inheritance, 117

- named arguments, 92
- nested classes, 115
- nested function, 98
- new
  - classes, 118
- `new`, 118
- nil
  - not provided for records, 135
- `nil`, 117
- `numeric`, 182
- `numLocales`, 214

- objects, 113
- `on`, 216
- opaque domains
  - default values, 158
  - literals, lack thereof, 158
  - types, 157
  - values, 158
- operator
  - lexical structure, 29
  - procedure, 89
- operator overloading, 98
- operators
  - +, on strings, 68
  - arithmetic, 57
  - assignment, 73
  - associativity, 54
  - binary, 57
  - bitwise, 62
  - by, 68
  - compound assignment, 73
  - logical, 64
  - operator expressions, 57
  - overloading, 98
  - precedence, 54
  - relational, 65
  - unary, 57
- optional modules, 241
  - AdvancedIters, 241
  - BitOps, 242
  - Norm, 242
  - Random, 243
  - Search, 244
  - Sort, 244
  - Time, 245
  - UtilMath, 246
- `out`, 94

- parallel iterators, 177
- `param`, 43, 77
- parameter function, 96
- parameters, 43
  - configuration, 44
  - in classes or records, 185
- `proc`, 90
- procedure, 89
  - definition, 90
- promotion, 207
  - zipper iteration, 208

- range-based slicing, 163
- ranges, 139
  - align operator, 146
  - aligned high bound, 140
  - aligned integer, 140

  - aligned low bound, 140
  - alignment, 139
    - ambiguous, 139
  - ambiguously aligned, 140
  - arithmetic operators, 147
  - assignment, 143
  - bounded, 141
  - boundedHigh, 141
  - boundedLow, 141
  - boundedNone, 141
  - boundedType, 141
  - by operator, 145
  - count operator, 146
  - default values, 143
  - empty, 140
  - first index, 140
  - high bound, 139
  - idxType, 141
  - iterable, 140
  - last index, 140
  - literals, 142
  - low bound, 139
  - operators, 144
  - predefined functions, 149
  - represented sequence, 139
    - decreasing, 140
    - increasing, 140
  - sequence, 139
  - slicing, 148
  - stridable, 141
  - stride, 139
  - stride type, 141
  - strided, 145
  - types, 141
- rank-change slicing, 163
- read, 192
  - default methods, 194
  - on files, 193
- `read`, 192
- `readFE`, 198
- `readFF`, 198
- `readln`, 192
- `readXX`, 198
- `real`, 33
- receiver, 120, 121, 131
- `record`, 125
- records, 125
  - allocation, 129, 133
  - arguments, 130, 134
  - assignment, 131, 134
    - objects, 132
  - comparison, 135
  - declarations, 125
    - differences with classes, 133
  - differences with classes, 133
  - equality, 133
  - field access, 130
  - fields, 127
  - generic, 184
  - getters, 130
  - implicit conversion, 133
  - indexing, 131
  - inequality, 133
  - inheritance, 127, 134
  - initialization, 129
  - iterating, 131
  - method calls, 131
  - methods, 127
  - nested, 127
  - overriding, 134
  - receiver, 131
  - record types, 126
  - shadowing, 134
  - variable declarations, 128
- rectangular arrays
  - indexing, 168
- rectangular domains
  - default value, 156
  - literals, 155
  - types, 154
  - values, 155
- reduction expressions, 209
- reductions, 209
- remote, 213
- Replicated distribution, 251
- `reset`, 198
- `return`, 97
  - types, 97

- scan expressions, 210
- scans, 209
- `select`, 74
- `serial`, 201
- setter, 96
- `single`, 196
- slicing
  - arrays, 170
  - domains, 162
  - range-based, 163
  - rank-change, 163
- sparse domains
  - default value, 159

- initializations, 159
- literals, lack thereof, 159

standard distributions (see also distributions, standard), 247
standard modules, 235
- automatic, 235
- optional, 241

statement, 71
`stderr`, 192
`stdin`, 192
`stdout`, 192
`string`, 33
subdomains
- sparse, 159
  - types, 159
  - values, 159
- types
  - simple, 158

swap
- operator, 73
- statement, 73

`sync`, 196, 200
synchronization types
- formal arguments, 197

synchronization variables
- predefined methods on, 198
- `single`, 196
- `sync`, 196

`then`, 69, 74
`these`, 122
`this`, 121, 122, 131
`tuple`, 104
tuples, 103
- assignment, 105
- destructuring, 105
- expanding in place, 109
- formal arguments grouped as, 108
- homogeneous, 103
- indexing, 109
- indices grouped as, 107
- omitting components, 106–108
- operators, 110
- predefined functions, 111
- types, 103
- values, 104
- variable declarations grouped as, 106

type aliases, 36
- in classes or records, 184

type inference, 41
- of return types, 98

type select statements, 79
types
- associative domains, 156
- dataparallel, 36
- enumerated domains, 157
- generic, 184
- locale, 34
- opaque domains, 157
- primitive, 31
- rectangular domains, 154
- structured, 35
- subdomains
  - simple, 158
- synchronisation, 36

`uint`, 32
`union`, 137
unions, 137
- assignment, 138
- fields, 137
- `type select`, 138

`use`, 78
user-defined constructors, 118

variables
- configuration, 44
- declarations, 39
- default initialization, 40
- default values, 40
- global, 42
- in classes or records, 185
- local, 42

`void`, 32

`when`, 74
`where`, 188
- implicit, 181

`while`, 75
while loops, 75
white space, 25
whole array assignment, 208
write, 192
- default methods, 194
- on files, 193
- on strings, 193

`write`, 192
`writeEF`, 198
`writeFF`, 198
`writeln`, 192
`Writer`, 193
`writeXF`, 198

`yield`, 175

zipper iteration, 77