Chapel Language Specification
Version 0.988

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

September 19, 2019
# Contents

1 Scope .......................... 14  
2 Notation .......................... 15  
3 Organization ......................... 17  
4 Acknowledgments ...................... 19  
5 Language Overview ..................... 20  
   5.1 Guiding Principles ................. 20  
      5.1.1 General Parallel Programming .... 20  
      5.1.2 Locality-Aware Programming .... 21  
      5.1.3 Object-Oriented Programming ... 21  
      5.1.4 Generic Programming ......... 22  
   5.2 Getting Started ................... 22  
6 Lexical Structure ..................... 23  
   6.1 Comments ......................... 23  
   6.2 White Space ....................... 23  
   6.3 Case Sensitivity .................... 23  
   6.4 Tokens ............................ 23  
      6.4.1 Identifiers ..................... 24  
      6.4.2 Keywords ...................... 24  
      6.4.3 Literals ....................... 25  
      6.4.4 Operators and Punctuation ...... 29  
      6.4.5 Grouping Tokens ............... 30  
7 Types .............................. 31  
   7.1 Primitive Types .................... 31  
      7.1.1 The Void Type .................. 32  
      7.1.2 The Nothing Type .............. 32  
      7.1.3 The Bool Type ................. 33  
      7.1.4 Signed and Unsigned Integral Types .... 33  
      7.1.5 Real Types ................. 33  
      7.1.6 Imaginary Types ............... 33  
      7.1.7 Complex Types ................. 34  
      7.1.8 The String Type ............... 34  
      7.1.9 The Bytes Type ............... 34  
      7.2 Enumerated Types ............... 34  
   7.3 Structured Types ................. 36  
      7.3.1 Class Types ................. 36  
      7.3.2 Record Types ............... 36  
      7.3.3 Union Types ............... 36  
      7.3.4 Tuple Types ............... 37  
   7.4 Data Parallel Types .............. 38  
      7.4.1 Range Types ............... 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.13.2</td>
<td>Unary Minus Operators</td>
</tr>
<tr>
<td>10.13.3</td>
<td>Addition Operators</td>
</tr>
<tr>
<td>10.13.4</td>
<td>Subtraction Operators</td>
</tr>
<tr>
<td>10.13.5</td>
<td>Multiplication Operators</td>
</tr>
<tr>
<td>10.13.6</td>
<td>Division Operators</td>
</tr>
<tr>
<td>10.13.7</td>
<td>Modulus Operators</td>
</tr>
<tr>
<td>10.13.8</td>
<td>Exponentiation Operators</td>
</tr>
<tr>
<td>10.14</td>
<td>Bitwise Operators</td>
</tr>
<tr>
<td>10.14.1</td>
<td>Bitwise Complement Operators</td>
</tr>
<tr>
<td>10.14.2</td>
<td>Bitwise And Operators</td>
</tr>
<tr>
<td>10.14.3</td>
<td>Bitwise Or Operators</td>
</tr>
<tr>
<td>10.14.4</td>
<td>Bitwise Xor Operators</td>
</tr>
<tr>
<td>10.15</td>
<td>Shift Operators</td>
</tr>
<tr>
<td>10.16</td>
<td>Logical Operators</td>
</tr>
<tr>
<td>10.16.1</td>
<td>The Logical Negation Operator</td>
</tr>
<tr>
<td>10.16.2</td>
<td>The Logical And Operator</td>
</tr>
<tr>
<td>10.16.3</td>
<td>The Logical Or Operator</td>
</tr>
<tr>
<td>10.17</td>
<td>Relational Operators</td>
</tr>
<tr>
<td>10.17.1</td>
<td>Ordered Comparison Operators</td>
</tr>
<tr>
<td>10.17.2</td>
<td>Equality Comparison Operators</td>
</tr>
<tr>
<td>10.18</td>
<td>Class Operators</td>
</tr>
<tr>
<td>10.19</td>
<td>Miscellaneous Operators</td>
</tr>
<tr>
<td>10.19.1</td>
<td>The String Concatenation Operator</td>
</tr>
<tr>
<td>10.19.2</td>
<td>The By Operator</td>
</tr>
<tr>
<td>10.19.3</td>
<td>The Align Operator</td>
</tr>
<tr>
<td>10.19.4</td>
<td>The Range Count Operator</td>
</tr>
<tr>
<td>10.20</td>
<td>Let Expressions</td>
</tr>
<tr>
<td>10.21</td>
<td>Conditional Expressions</td>
</tr>
<tr>
<td>10.22.1</td>
<td>Filtering Predicates in For Expressions</td>
</tr>
</tbody>
</table>

### 11. Statements

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1</td>
<td>Blocks</td>
</tr>
<tr>
<td>11.2</td>
<td>Expression Statements</td>
</tr>
<tr>
<td>11.3</td>
<td>Assignment Statements</td>
</tr>
<tr>
<td>11.4</td>
<td>The Swap Statement</td>
</tr>
<tr>
<td>11.5</td>
<td>The I/O Statement</td>
</tr>
<tr>
<td>11.6</td>
<td>The Conditional Statement</td>
</tr>
<tr>
<td>11.7</td>
<td>The Select Statement</td>
</tr>
<tr>
<td>11.8</td>
<td>The While Do and Do While Loops</td>
</tr>
<tr>
<td>11.9</td>
<td>The For Loop</td>
</tr>
<tr>
<td>11.9.1</td>
<td>Zipper Iteration</td>
</tr>
<tr>
<td>11.9.2</td>
<td>Parameter For Loops</td>
</tr>
<tr>
<td>11.10</td>
<td>The Break, Continue and Label Statements</td>
</tr>
<tr>
<td>11.11</td>
<td>The Use Statement</td>
</tr>
<tr>
<td>11.12</td>
<td>The Defer Statement</td>
</tr>
<tr>
<td>11.13</td>
<td>The Empty Statement</td>
</tr>
</tbody>
</table>
12 Modules
12.1 Module Definitions
12.2 Prototype Modules
12.3 Files and Implicit Modules
12.4 Nested Modules
12.5 Access of Module Contents
12.5.1 Visibility Of A Module
12.5.2 Visibility Of A Module’s Symbols
12.5.3 Using Modules
12.5.4 Qualified Naming of Module Symbols
12.5.5 Module Initialization
12.5.6 Module Deinitialization
12.6 Program Execution
12.6.1 The main Function
12.6.2 Module Initialization Order
12.6.3 Module Deinitialization Order

13 Procedures
13.1 Function Calls
13.2 Procedure Definitions
13.3 Functions without Parentheses
13.4 Formal Arguments
13.4.1 Named Arguments
13.4.2 Default Values
13.5 Argument Intents
13.5.1 Concrete Intents
13.5.2 Abstract Intents
13.5.3 Abstract Intents Table
13.5.4 Abstract Intent for Arrays and Record ‘this’
13.5.5 Abstract Intent for ‘owned’ and ‘shared’
13.6 Variable Number of Arguments
13.7 Return Intents
13.7.1 The Ref Return Intent
13.7.2 The Const Ref Return Intent
13.7.3 Return Intent Overloads
13.7.4 The Param Return Intent
13.7.5 The Type Return Intent
13.8 The Return Statement
13.9 Return Types
13.9.1 Explicit Return Types
13.9.2 Implicit Return Types
13.10 Where Clauses
### Chapel Language Specification

**13.11 Nested Functions** ...................................................... 114
**13.12 Function and Operator Overloading** ............................. 114
**13.13 Function Resolution** ................................................. 115
**13.13.1 Determining Visible Functions** ................................. 115
**13.13.2 Determining Candidate Functions** ............................ 115
- **Valid Mapping** ............................................................. 115
- **Legal Argument Mapping** ............................................ 116
**13.13.3 Determining More Specific Functions** ....................... 116
**13.13.4 Determining Best Functions** .................................... 118
**13.13.5 Choosing Return Intent Overloads Based on Calling Context** .............................................................. 118

**14 Methods** ....................................................................... 120
**14.1 Method Calls** ............................................................. 120
**14.2 The Method Receiver and the *this* Argument** ............... 121
**14.3 The *this* Method** ...................................................... 123
**14.4 The *these* Method** .................................................... 123

**15 Error Handling** ............................................................ 125
**15.1 Throwing Errors** ......................................................... 125
**15.2 Handling Errors** ......................................................... 125
**15.2.1 Halting on error with try!** ........................................... 126
**15.2.2 Handling an error with catch** ...................................... 126
- **Catch clauses** ............................................................... 126
- **try! with catch** ............................................................. 127
- **Nested try** ................................................................. 127
**15.2.3 Propagating an error with throws** ............................... 127
- **After catch clauses** ....................................................... 128
- **catch-less try** .............................................................. 128
- **try expressions** ........................................................... 128
**15.2.4 Complete handling** .................................................. 129
**15.3 Defer statement** .......................................................... 129
**15.4 Methods** ................................................................. 130
**15.5 Multilocale** ............................................................. 130
**15.6 Parallelism** ............................................................... 130
**15.6.1 TaskErrors** ........................................................... 130
**15.6.2 begin** ................................................................. 131
**15.6.3 coforall and cobegin** ............................................... 131
**15.6.4 forall** ................................................................. 132
**15.7 Creating New Error Types** .......................................... 132
**15.8 Error Handling Modes** ................................................. 132
**15.8.1 Prototype Mode** ...................................................... 133
**15.8.2 Production Mode** .................................................... 133

**16 Tuples** ........................................................................... 135
**16.1 Tuple Types** ............................................................. 135
**16.2 Tuple Values** ............................................................. 136
**16.3 Tuple Indexing** .......................................................... 137
**16.4 Iteration over Tuples** .................................................. 137
**16.5 Tuple Assignment** ...................................................... 138
**16.6 Tuple Destructuring** .................................................... 138
**16.6.1 Splitting a Tuple with Assignment** .............................. 138
16.6.2 Splitting a Tuple in a Declaration ................................................................. 139
16.6.3 Splitting a Tuple into Multiple Indices of a Loop ........................................ 140
16.6.4 Splitting a Tuple into Multiple Formal Arguments in a Function Call ............ 140
16.6.5 Splitting a Tuple via Tuple Expansion ......................................................... 141
16.7 Tuple Operators .............................................................................................. 142
16.7.1 Unary Operators .......................................................................................... 142
16.7.2 Binary Operators ......................................................................................... 142
16.7.3 Relational Operators .................................................................................... 142
16.8 Predefined Functions and Methods on Tuples .................................................. 143
17 Classes ............................................................................................................... 144
17.1 Class Declarations .......................................................................................... 144
17.1.1 Class Lifetime and Borrows ........................................................................ 145
17.1.2 Class Types .................................................................................................. 146
17.1.3 Nilable Class Types ...................................................................................... 147
17.1.4 Class Values ................................................................................................ 148
17.1.5 The $nil$ Value ............................................................................................ 148
17.1.6 Class Fields ................................................................................................ 149
17.1.7 Class Methods ............................................................................................. 149
17.1.8 Nested Classes ............................................................................................. 150
17.2 Inheritance ...................................................................................................... 150
17.2.1 The $object$ Class ....................................................................................... 150
17.2.2 Accessing Base Class Fields ....................................................................... 151
17.2.3 Shadowing Base Class Fields ..................................................................... 151
17.2.4 Overriding Base Class Methods ................................................................... 151
17.3 Class New ........................................................................................................ 152
17.4 Class Initializers .............................................................................................. 152
17.4.1 User-Defined Initializers ............................................................................. 153
  Field Initialization Versus Assignment ................................................................. 153
  Omitting Field Initializations .............................................................................. 154
  Limitations on Instance Usage in Initializers ....................................................... 155
  Invoking Other Initializers ................................................................................. 157
  Initializing Fields in Conditional Statements ..................................................... 157
  Miscellaneous Field Initialization Rules .............................................................. 158
17.4.2 The Compiler-Generated Initializer .............................................................. 159
17.4.3 The postinit Method .................................................................................... 159
17.4.4 Initializing Inherited Classes ....................................................................... 160
  Calling Methods on Parent Classes .................................................................... 161
  The Compiler Generated Initializer for Inheriting Classes .................................. 162
  The postinit Method for Inheriting Classes ......................................................... 163
17.5 Field Accesses ................................................................................................ 163
17.5.1 Variable Getter Methods .......................................................................... 164
17.6 Class Method Calls ........................................................................................ 164
17.7 Common Operations ....................................................................................... 164
17.7.1 Class Assignment ....................................................................................... 164
17.8 Deleting Unmanaged Class Instances ............................................................ 165
17.8.1 Class Deinitializer ...................................................................................... 166
# 18 Records

18.1 Record Declarations ......................................................................................... 167
18.1.1 Record Types ............................................................................................... 168
18.1.2 Record Fields .............................................................................................. 168
18.1.3 Record Methods .......................................................................................... 168
18.1.4 Nested Record Types ................................................................................. 169
18.2 Record Variable Declarations ......................................................................... 169
18.2.1 Storage Allocation ...................................................................................... 169
18.2.2 Record Initialization .................................................................................... 169
18.2.3 Record Deinitializer ..................................................................................... 170
18.3 Record Arguments ......................................................................................... 171
18.4 Record Field Access ....................................................................................... 171
18.4.1 Field Getter Methods ................................................................................. 172
18.5 Record Method Calls ..................................................................................... 172
18.6 Common Operations ....................................................................................... 172
18.6.1 Copy Initialization of Records .................................................................. 172
18.6.2 Record Assignment .................................................................................... 173
18.6.3 Default Comparison Operators ................................................................. 173
18.7 Differences between Classes and Records .................................................... 173
18.7.1 Declarations ............................................................................................... 174
18.7.2 Storage Allocation ...................................................................................... 174
18.7.3 Assignment ................................................................................................ 174
18.7.4 Arguments ................................................................................................ 174
18.7.5 No nil Value .............................................................................................. 174
18.7.6 The delete operator .................................................................................... 175
18.7.7 Default Comparison Operators ................................................................. 175

# 19 Unions

19.1 Union Types .................................................................................................. 176
19.2 Union Declarations ....................................................................................... 176
19.2.1 Union Fields .............................................................................................. 176
19.3 Union Assignment .......................................................................................... 177

# 20 Ranges

20.1 Range Concepts ............................................................................................. 178
20.2 Range Types .................................................................................................. 180
20.3 Range Values ................................................................................................ 181
20.3.1 Range Literals ........................................................................................... 181
20.3.2 Default Values ........................................................................................... 182
20.4 Common Operations ...................................................................................... 182
20.4.1 Range Assignment ..................................................................................... 183
20.4.2 Range Comparisons .................................................................................. 183
20.4.3 Iterating over Ranges ................................................................................ 183
20.4.4 Iterating over Unbounded Ranges in Zippered Iterations ...................... 183
20.4.4 Range Promotion of Scalar Functions ...................................................... 184
20.5 Range Operators .......................................................................................... 184
20.5.1 By Operator ................................................................................................ 184
20.5.2 Align Operator ............................................................................................ 185
20.5.3 Count Operator ........................................................................................... 186
20.5.4 Arithmetic Operators ................................................................................ 187
20.5.5 Range Slicing ............................................................................................ 188
## 20.6 Predefined Functions on Ranges

- Predefined Functions on Ranges: 188
  - Range Type Parameters: 188
  - Range Properties: 189
  - Other Queries: 190
  - Range Transformations: 191

## 21 Domains

- Domain Overview: 193
- Base Domain Types and Values: 194
  - Rectangular Domains: 194
    - Rectangular Domain Types: 194
    - Rectangular Domain Values: 195
  - Associative Domains: 196
    - Associative Domain Types: 196
    - Associative Domain Values: 197
- Simple Subdomain Types and Values: 198
  - Simple Subdomain Types: 198
  - Simple Subdomain Values: 198
  - Sparse Subdomain Types and Values: 199
  - Sparse Subdomain Types: 199
  - Sparse Subdomain Values: 199
- Domain Index Types: 199
- Iteration Over Domains: 200
- Domains as Arguments: 200
  - Formal Arguments of Domain Type: 200
  - Domain Promotion of Scalar Functions: 201
- Domain Operations: 201
  - Domain Assignment: 201
  - Domain Striding: 202
  - Domain Alignment: 202
  - Domain Slicing: 202
    - Domain-based Slicing: 203
    - Range-based Slicing: 203
    - Rank-Change Slicing: 203
  - Count Operator: 203
- Methods on All Domain Types: 204
- Methods on Regular Domains: 206
- Methods on Irregular Domains: 207

## 22 Arrays

- Array Types: 208
- Array Values: 208
  - Rectangular Array Literals: 209
  - Associative Array Literals: 209
- Runtime Representation of Array Values: 210
- Array Indexing: 210
  - Rectangular Array Indexing: 211
  - Associative Array Indexing: 211
- Iteration over Arrays: 212
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>Task Parallelism and Synchronization</td>
<td>237</td>
</tr>
<tr>
<td>26.1</td>
<td>Tasks and Task Parallelism</td>
<td>237</td>
</tr>
<tr>
<td>26.2</td>
<td>The Begin Statement</td>
<td>238</td>
</tr>
<tr>
<td>26.3</td>
<td>Synchronization Variables</td>
<td>238</td>
</tr>
<tr>
<td>26.3.1</td>
<td>Predefined Single and Sync Methods</td>
<td>240</td>
</tr>
<tr>
<td>26.4</td>
<td>Atomic Variables</td>
<td>242</td>
</tr>
<tr>
<td>26.4.1</td>
<td>Predefined Atomic Methods</td>
<td>242</td>
</tr>
<tr>
<td>26.5</td>
<td>The Cobegin Statement</td>
<td>244</td>
</tr>
<tr>
<td>26.6</td>
<td>The Coforall Loop</td>
<td>244</td>
</tr>
<tr>
<td>26.7</td>
<td>Task Intents</td>
<td>245</td>
</tr>
<tr>
<td>26.8</td>
<td>The Sync Statement</td>
<td>247</td>
</tr>
<tr>
<td>26.9</td>
<td>The Serial Statement</td>
<td>248</td>
</tr>
<tr>
<td>26.10</td>
<td>Atomic Statements</td>
<td>249</td>
</tr>
<tr>
<td>27</td>
<td>Data Parallelism</td>
<td>250</td>
</tr>
<tr>
<td>27.1</td>
<td>The Forall Statement</td>
<td>250</td>
</tr>
<tr>
<td>27.1.1</td>
<td>Syntax</td>
<td>250</td>
</tr>
<tr>
<td>27.1.2</td>
<td>Execution and Serializability</td>
<td>251</td>
</tr>
<tr>
<td>27.1.3</td>
<td>Zipper Iteration</td>
<td>251</td>
</tr>
<tr>
<td>27.2</td>
<td>The Forall Expression</td>
<td>252</td>
</tr>
<tr>
<td>27.2.1</td>
<td>Syntax</td>
<td>252</td>
</tr>
<tr>
<td>27.2.2</td>
<td>Execution</td>
<td>252</td>
</tr>
<tr>
<td>27.2.3</td>
<td>Zipper Iteration</td>
<td>252</td>
</tr>
<tr>
<td>27.2.4</td>
<td>Filtering Predicates in Forall Expressions</td>
<td>253</td>
</tr>
<tr>
<td>27.3</td>
<td>Forall Intents</td>
<td>253</td>
</tr>
<tr>
<td>27.4</td>
<td>Task-Private Variables</td>
<td>254</td>
</tr>
<tr>
<td>27.5</td>
<td>Promotion</td>
<td>255</td>
</tr>
<tr>
<td>27.5.1</td>
<td>Default Arguments</td>
<td>256</td>
</tr>
<tr>
<td>27.5.2</td>
<td>Zipper Promotion</td>
<td>256</td>
</tr>
<tr>
<td>27.5.3</td>
<td>Whole Array Operations and Evaluation Order</td>
<td>257</td>
</tr>
<tr>
<td>27.6</td>
<td>Reductions and Scans</td>
<td>258</td>
</tr>
<tr>
<td>27.6.1</td>
<td>Reduction Expressions</td>
<td>258</td>
</tr>
<tr>
<td>27.6.2</td>
<td>Scan Expressions</td>
<td>259</td>
</tr>
<tr>
<td>27.7</td>
<td>Configuration Constants for Default Data Parallelism</td>
<td>259</td>
</tr>
<tr>
<td>28</td>
<td>Locales</td>
<td>261</td>
</tr>
<tr>
<td>28.1</td>
<td>Locales</td>
<td>261</td>
</tr>
<tr>
<td>28.1.1</td>
<td>Locale Types</td>
<td>261</td>
</tr>
<tr>
<td>28.1.2</td>
<td>Locale Methods</td>
<td>261</td>
</tr>
<tr>
<td>28.1.3</td>
<td>The Predefined Locales Array</td>
<td>262</td>
</tr>
<tr>
<td>28.1.4</td>
<td>The here Locale</td>
<td>263</td>
</tr>
<tr>
<td>28.1.5</td>
<td>Querying the Locale of an Expression</td>
<td>263</td>
</tr>
<tr>
<td>28.2</td>
<td>The On Statement</td>
<td>264</td>
</tr>
<tr>
<td>28.2.1</td>
<td>Remote Variable Declarations</td>
<td>264</td>
</tr>
<tr>
<td>29</td>
<td>Domain Maps</td>
<td>265</td>
</tr>
<tr>
<td>29.1</td>
<td>Domain Maps for Domain Types</td>
<td>265</td>
</tr>
<tr>
<td>29.2</td>
<td>Domain Maps for Domain Values</td>
<td>267</td>
</tr>
<tr>
<td>29.3</td>
<td>Domain Maps for Arrays</td>
<td>267</td>
</tr>
<tr>
<td>29.4</td>
<td>Domain Maps Are Not Retained upon Domain Assignment</td>
<td>268</td>
</tr>
<tr>
<td>30</td>
<td>User-Defined Reductions and Scans</td>
<td>269</td>
</tr>
</tbody>
</table>
31 Memory Consistency Model 270
31.1 Sequential Consistency for Data-Race-Free Programs 270
31.1.1 Program Order 272
31.1.2 Memory Order 272
31.2 Non-Sequentially Consistent Atomic Operations 273
31.2.1 Relaxed Atomic Operations 273
31.3 Unordered Memory Operations 274
31.3.1 Unordered Memory Operations Examples 274
31.4 Examples 275

32 Interoperability 277
32.1 Interoperability Overview 277
32.1.1 Calling External Functions 277
32.1.2 Calling Chapel Functions 278
32.2 Shared Language Elements 279
32.2.1 Shared Types 279
   Referring to Standard C Types 279
   Referring to External C Types 280
   Referring to External C Structs 280
   Opaque Types 281
32.2.2 Shared Data 282
32.2.3 Shared Procedures 282
   Calling External C Functions 282
32.2.4 Calling Chapel Procedures Externally 283
32.2.5 Argument Passing 283

A Collected Lexical and Syntax Productions 285
A.1 Alphabetical Lexical Productions 285
A.2 Alphabetical Syntax Productions 288
A.3 Depth-First Lexical Productions 304
A.4 Depth-First Syntax Productions 307

Index 324
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.

```chapel
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

```chapel
classic: one of
    true
    false
```

defines `classic` 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

```chapel
unary-operator: one of
    +  -  ~  !
```

is equivalent to

```chapel
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, Acknowledgements, 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 namespace management.
• Chapter 13, Functions, describes functions and function resolution in Chapel.
• Chapter 16, Tuples, describes tuples in Chapel.
• Chapter 17, Classes, describes reference classes in Chapel.
• Chapter 18, Records, describes records or value classes in Chapel.
• Chapter 19, Unions, describes unions in Chapel.
• Chapter 20, Ranges, describes ranges in Chapel.
• Chapter 21, 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 22, Arrays, describes arrays in Chapel. Chapel arrays are more general than in most languages including support for multidimensional, sparse, associative, and unstructured arrays.
• Chapter 23, Iterators, describes iterator functions.
• Chapter 24, Generics, describes Chapel’s support for generic functions and types.
• Chapter 25, Input and Output, describes support for input and output in Chapel, including file input and output.
• Chapter 26, Task Parallelism and Synchronization, describes task-parallel expressions and statements in Chapel as well as synchronization constructs, atomic variables, and the atomic statement.
• Chapter 27, Data Parallelism, describes data-parallel expressions and statements in Chapel including reductions and scans, whole array assignment, and promotion.

• Chapter 28, Locales, describes constructs for managing locality and executing Chapel programs on distributed-memory systems.

• Chapter 29, 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 30, User-Defined Reductions and Scans, describes how Chapel programmers can define their own reduction and scan operators.

• Chapter 31, Memory Consistency Model, describes Chapel’s rules for ordering the reads and writes performed by a program’s tasks.

• Chapter 32 describes Chapel’s interoperability features for combining Chapel programs with code written in different languages.

• 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 been actively involved in the recent evolution of the Chapel language and its specification: Kyle Brady, Bradford Chamberlain, Sung-Eun Choi, Lydia Duncan, Michael Ferguson, Ben Harshbarger, Tom Hildebrandt, David Iten, Vassily Litvinov, Tom MacDonald, Michael Noakes, Elliot Ronaghan, Greg Titus, Thomas Van Doren, and Tim Zakian

The following people have contributed to previous versions of the language and its specification: Robert Bocchino, David Callahan, Steven Deitz, Roxana Diaconescu, James Dinan, Samuel Figueroa, Shannon Hoffswell, Mary Beth Hribar, Mark James, Mackale Joyner, Jacob Nelson, John Plevyak, Lee Prokowich, Albert Sidelnik, Andy Stone, Wayne Wong, and Hans Zima.

We are also grateful to our many enthusiastic and vocal users for helping us continually improve the quality of the Chapel language and compiler.

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 an emerging parallel programming language designed for productive scalable computing. Chapel’s primary goal is to make parallel programming far more productive, from multicore desktops and laptops to commodity clusters and the cloud to high-end supercomputers. Chapel's design and development are being led by Cray Inc. in collaboration with academia, computing centers, and industry.

Chapel is being developed in an open-source manner at GitHub under the Apache v2.0 license and also makes use of other third-party open-source packages under their own licenses. Chapel emerged from Cray’s entry in the DARPA-led High Productivity Computing Systems program (HPCS). It is currently being hardened from that initial prototype to more of a product-grade implementation.

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 that supports global-view data structures, as Chapel does, 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.
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:

```chapel
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:

```bash
> chpl hello.chpl
> ./hello
```

The following output will be printed to the console:

```text
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. A comment delimited by /* and */ can be nested in another comment delimited by /* and */

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 bytes or string literal, they do not start a comment but rather are part of the bytes or string literal.

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

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

6.2 White Space

White-space characters are spaces, tabs, line feeds, form 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-or-underscore legal-identifier-chars opt

legal-identifier-chars:
  legal-identifier-char legal-identifier-chars opt

legal-identifier-char:
  letter-or-underscore digit
  $

letter-or-underscore:
  letter
  _

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:
The following identifiers are keywords reserved for future use:

```
lambda
pragma
primitive
```

### 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
  0o octal-digits
  0O octal-digits
  0b binary-digits
  0B binary-digits

digits:
  digit
  digit separator--digits

separator--digits:
  digit

  digit separator--digits
  _ separator--digits
```
hexadecimal-digits:
  hexadecimal-digit
  hexadecimal-digit separator hexadecimal-digits

separator-hexadecimal-digits:
  hexadecimal-digit
  __ hexadecimal-digit separator hexadecimal-digits
  __ separator-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

octal-digits:
  octal-digit
  octal-digit separator octal-digits

separator-octal-digits:
  octal-digit
  __ octal-digit separator octal-digits
  __ separator-octal-digits

octal-digit: one of 0 1 2 3 4 5 6 7

binary-digits:
  binary-digit
  binary-digit separator binary-digits

separator-binary-digits:
  binary-digit
  __ binary-digit separator binary-digits
  __ separator-binary-digits

binary-digit: one of 0 1

Integer literals in the range 0 to max(int), §7.1.4 have type int and the remaining literals have 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.

Rationale. Underscores can be used to group the digits of numbers for legibility. For example:

```
var i = 1_234_567_890;
var x = 0xFF_FF_12_34;
```

Real literals are designated by the following syntax:
real-literal:
  digits\(\text{opt}\) \(\cdot\) digits exponent-part-opt
  digits opt exponent-part
  0x hexadecimal-digits\(\text{opt}\) \(\cdot\) hexadecimal-digits p-exponent-part-opt
  0X hexadecimal-digits\(\text{opt}\) \(\cdot\) hexadecimal-digits p-exponent-part-opt
  0x hexadecimal-digits\(\text{opt}\) p-exponent-part
  0X hexadecimal-digits\(\text{opt}\) p-exponent-part

exponent-part:
  e sign opt digits
  E sign opt digits

p-exponent-part:
  p sign opt digits
  P sign opt digits

sign: one of
  + -

Rationale. Why can’t a real literal end with "."? There is a lexical ambiguity 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.

Hexadecimal real literals are supported with a hexadecimal integer and fractional part. Because ‘e’ could be a hexadecimal character, the exponent for these literals is instead marked with ‘p’ or ‘P’. The exponent value follows and is written in decimal.

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:

\[\text{imaginary-literal:}\]
  \text{real-literal} \text{i}
  \text{integer-literal} \text{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: \(1.0 + 2.0\text{i}\) and \((1.0, 2.0)\):complex.
Interpreted string literals are designated by the following syntax:

```
interpreted-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
```

Uninterpreted string literals are designated by the following syntax:

```
uninterpreted-string-literal:
  """ uninterpreted-double-quote-delimited-characters """
  """ uninterpreted-single-quote-delimited-characters """

uninterpreted-double-quote-delimited-characters:
  uninterpreted-double-quote-string-character uninterpreted-double-quote-delimited-characters_opt

uninterpreted-single-quote-delimited-characters:
  uninterpreted-single-quote-string-character uninterpreted-single-quote-delimited-characters_opt

uninterpreted-double-quote-string-character:
  any character except three double quotes in a row

uninterpreted-single-quote-string-character:
  any character except three single quotes in a row
```

Uninterpreted string literals do not interpret their contents, so for example """\n""" is not a newline, but rather two characters ‘\’ and ‘n’. Uninterpreted string literals may span multiple lines and the literal newline characters will be included in the string.

A string literal can be either interpreted or uninterpreted.

```
string-literal:
  interpreted-string-literal
  uninterpreted-string-literal
```
Lexical Structure

Interpreted bytes literals are designated by the following syntax:

interpreted-bytes-literal:
   b" double-quote-delimited-characters_{opt}"
   b' single-quote-delimited-characters_{opt}'

Uninterpreted bytes literals are designated by the following syntax:

uninterpreted-bytes-literal:
   b"=" uninterpreted-double-quote-delimited-characters"
   b'=' uninterpreted-single-quote-delimited-characters

Uninterpreted bytes literals do not interpret their contents, so for example b""n"" is not a newline, but rather two characters ‘\’ and ‘n’. Uninterpreted bytes literals may span multiple lines and the literal newline characters will be included in the bytes.

A bytes literal can be either interpreted or uninterpreted.

bytes-literal:
   interpreted-bytes-literal
   uninterpreted-bytes-literal

6.4.4 Operators and Punctuation

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

<table>
<thead>
<tr>
<th>symbols</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>assignment</td>
</tr>
<tr>
<td>+=</td>
<td>compound assignment</td>
</tr>
<tr>
<td>-=</td>
<td>swap</td>
</tr>
<tr>
<td>*=</td>
<td>I/O</td>
</tr>
<tr>
<td>/=</td>
<td>range specifier</td>
</tr>
<tr>
<td>**=</td>
<td>range/domain stride specifier</td>
</tr>
<tr>
<td>%=</td>
<td>range count operator</td>
</tr>
<tr>
<td>&amp;=</td>
<td>variable argument lists</td>
</tr>
<tr>
<td>^=</td>
<td>logical/bitwise operators</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>type specifier</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&gt;&lt;&gt;</td>
</tr>
<tr>
<td></td>
<td>type query</td>
</tr>
<tr>
<td>...</td>
<td>string delimiters</td>
</tr>
</tbody>
</table>
6.4.5 Grouping Tokens

The following braces are part of the Chapel language:

<table>
<thead>
<tr>
<th>braces</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>parenthesization, function calls, and tuples</td>
</tr>
<tr>
<td>[ ]</td>
<td>array literals, array types, forall expressions, and function calls</td>
</tr>
<tr>
<td>{ }</td>
<td>domain literals, block statements</td>
</tr>
</tbody>
</table>
7 Types

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

The syntax of a type is as follows:

- **type-expression:**
  - **primitive-type**
  - **enum-type**
  - **structured-type**
  - **dataparallel-type**
  - **synchronization-type**
  - **lvalue-expression**
  - **if-expression**
  - **unary-expression**
  - **binary-expression**

Many expressions are syntactically allowed as a type; however not all expressions produce a type. For example, a call to a function is syntactically allowed as the type of a variable. However it would be an error for that call to result in a value (rather than a type) in that context.

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, §17.1, §18.1, §19.2, and §7.6, respectively.

7.1 Primitive Types

The concrete primitive types are: void, nothing, bool, int, uint, real, imag, complex, string and bytes. They are defined in this section.

In addition, there are several generic primitive types that are described in §24.3.1.

The primitive types are summarized by the following syntax:
Types

primitive-type:
  void
  nothing
  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
  bytes
  enum
  record
  class
  owned
  shared
  unmanaged
  borrowed

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. It is an error to assign the result of a function that returns void to a variable.

7.1.2 The Nothing Type

The nothing type is used to indicate a variable or field that should be removed by the compiler. The value none is the only value of type nothing.

The value none can only be assigned to a variable of type nothing, or to a generic variable that will take on the type nothing. The variable will be removed from the program and have no representation at run-time.

Rationale. The nothing type can be used to conditionally remove a variable or field from the code based on a param conditional expression.
7.1.3 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 (§9.1.5).

7.1.4 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` correspond to `int(64)` and `uint(64)` respectively.

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Minimum Value</th>
<th>Maximum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int(8)</code></td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td><code>uint(8)</code></td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td><code>int(16)</code></td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td><code>uint(16)</code></td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td><code>int(32)</code></td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td><code>uint(32)</code></td>
<td>0</td>
<td>4294967295</td>
</tr>
<tr>
<td><code>int(64).int</code></td>
<td>-9223372036854775808</td>
<td>9223372036854775807</td>
</tr>
<tr>
<td><code>uint(64).uint</code></td>
<td>0</td>
<td>18446744073709551615</td>
</tr>
</tbody>
</table>

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.

7.1.5 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.

7.1.6 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.
7.1.7 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. The standard Math module provides some functions on complex types. See [https://chapel-lang.org/docs/modules/standard/Math.html](https://chapel-lang.org/docs/modules/standard/Math.html)

**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.

7.1.8 The String Type

Strings are a primitive type designated by the symbol string comprised of Unicode characters in UTF-8 encoding. Their length is unbounded.

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

7.1.9 The Bytes Type

Bytes is a primitive type designated by the symbol bytes comprised of arbitrary bytes. Bytes are immutable in-place and their length is unbounded.

*Open issue.* There is an expectation of future support for mutable bytes.

7.2 Enumerated Types

Enumerated types are declared with the following syntax:

\[
\begin{align*}
\text{enum-declaration-statement:} & \\
& \text{enum identifier } \{ \text{enum-constant-list} \} \\
\end{align*}
\]

\[
\begin{align*}
\text{enum-constant-list:} & \\
& \text{enum-constant} \\
& \text{enum-constant, enum-constant-list_opt} \\
\end{align*}
\]

\[
\begin{align*}
\text{enum-constant:} & \\
& \text{identifier init-part_opt} \\
\end{align*}
\]

\[
\begin{align*}
\text{init-part:} & \\
& = \text{expression} \\
\end{align*}
\]
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. Each enumerated type is a distinct type.

If the `init-part` is omitted for all of the named constants in an enumerated type, the enumerated values are `abstract` and do not have associated integer values. Any constant that has an `init-part` will be associated with that integer value. Such constants must be parameter values of integral type. Any constant that does not have an `init-part`, yet which follows one that does, will be associated with an integer value one greater than its predecessor. An enumerated type whose first constant has an `init-part` is called `concrete`, since all constants in the enum will have an associated integer value, whether explicit or implicit. An enumerated type that specifies an `init-part` for some constants, but not the first, is called `semi-concrete`. Numeric conversions are automatically supported for enumerated types which are concrete or semi-concrete (see §9.2.3).

**Example (enum.chpl).** The code

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

defines an abstract enumerated type with four constants. The function

```chpl
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 name and a dot unless a use statement is employed (see §11.11).

It is possible to iterate over an enumerated type. The loop body will be invoked on each named constant in the enum. The following method is also available:

```chpl
proc enum.size: param int
```

The number of constants in the given enumerated type.
7.3 Structured Types

The structured types are summarized by the following syntax:

\[
\text{structured-type:}
\begin{align*}
\text{class-type} \\
\text{record-type} \\
\text{union-type} \\
\text{tuple-type}
\end{align*}
\]

Classes are discussed in §17. Records are discussed in §18. Unions are discussed in §19. Tuples are discussed in §16.

7.3.1 Class Types

A class can contain variables, constants, and methods.

Classes are defined in §17. The class type can also contain type aliases and parameters. Such a class is generic and is defined in §24.3.

A class type C has several variants:

- C and C?
- owned C and owned C?
- shared C and shared C?
- borrowed C and borrowed C?
- unmanaged C and unmanaged C?

The variants with a question mark, such as owned C?, can store nil (see §17.1.3). Variants without a question mark cannot store nil. The keywords owned, shared, borrowed, and unmanaged indicate the memory management strategy used for the class. When none is specified, as with C or C?, the class is considered to have generic memory management strategy. See §17.1.2.

7.3.2 Record Types

Records can contain variables, constants, and methods. Unlike class types, records are values rather than references. Records are defined in §18.

7.3.3 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 §19.
7.3.4 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 §16.
7.4 Data Parallel Types

The data parallel types are summarized by the following syntax:

\[
\text{dataparallel-type:} \\
\quad \text{range-type} \\
\quad \text{domain-type} \\
\quad \text{mapped-domain-type} \\
\quad \text{array-type} \\
\quad \text{index-type}
\]

Ranges and their index types are discussed in §20. Domains and their index types are discussed in §21. Arrays are discussed in §22.

7.4.1 Range Types

A range defines an integral sequence of some integral type. Ranges are defined in §20.

7.4.2 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 indices can be of any type. Domains, arrays, and their index types are defined in §21 and §22.

7.5 Synchronization Types

The synchronization types are summarized by the following syntax:

\[
\text{synchronization-type:} \\
\quad \text{sync-type} \\
\quad \text{single-type} \\
\quad \text{atomic-type}
\]

Sync and single types are discussed in §26.3. The atomic type is discussed in §26.4.
7.6 Type Aliases

Type aliases are declared with the following syntax:

```
  type-alias-declaration-statement:
    privacy-specifier, config, 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-expression
    identifier
```

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

Type aliases defined at the module level are public by default. The optional `privacy-specifier` keywords are provided to specify or change this behavior. For more details on the visibility of symbols, see §12.5.2.

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-expression` 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 (§32) for more information on external types.

The `type-expression` 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 (§24.3.3).

*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:
  privacy-specifier<opt> config-or-extern<opt> variable-kind variable-declaration-list ;

config-or-extern: one of
  config extern

variable-kind:
  param
  const
  var
  ref
  const ref

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

variable-declaration:
  identifier-list type-part<opt> initialization-part
  identifier-list type-part no-initialization-part<opt>

type-part:
  : type-expression

initialization-part:
  = expression

no-initialization-part:
  = noinit

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 module level; otherwise they are local. Module level variables are discussed in §8.2. Local variables are discussed in §8.3.

The optional privacy-specifier keywords indicate the visibility of module level variables to outside modules. By default, variables are publicly visible. More details on visibility can be found in §12.3.2.

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. See §32.2.2 for further details.

The variable-kind specifies whether the variables are parameters (param), constants (const), ref variables (ref), or regular variables (var). Parameters are compile-time constants whereas constants are runtime constants. Both levels of constants are discussed in §8.4. Ref variables are discussed in §8.6.

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 §8.1.3. If the type-part refers to a generic type, then an initialization-part is required and will be used to determine the type of the variable. In this event, the compiler will fail with an error if the initialization-part is not coercible to an instantiation of the generic type.

The initialization-part of a variable declaration specifies an initial expression to assign to the variable. If the initialization-part is omitted, the type-part must be present, and the variable is initialized to the default value of its type as described in §8.1.1.

If the no-initialization-part is present, the variable declaration does not initialize the variable to any value, as described in §8.1.2. The result of any read of an uninitialized variable is undefined until that variable is written.

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 §8.1.4.

Multiple variables can be grouped together using a tuple notation as described in §16.6.2.

### 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:
### 8.1.2 Deferred Initialization

For performance purposes, a variable’s declaration can specify that the variable should not be default initialized by using the `noinit` keyword in place of an initialization expression. Since this variable should be written at a later point in order to be read properly, it must be a regular variable (`var`). It is incompatible with declarations that require the variable to remain unchanged throughout the program’s lifetime, such as `const` or `param`. Additionally, its type must be specified at declaration time.

The result of any read of this variable before it is written is undefined; it exists and therefore can be accessed, but no guarantees are made as to its contents.

### 8.1.3 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

```go
var v = e;
```

is equivalent to

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

for an arbitrary expression `e`. If `e` is of sync or single type, the type of `v` is the base type of `e`.

### 8.1.4 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

```chapel
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 \texttt{real} with value \( 1.0 \). Variables \( c \) and \( d \) are of type \texttt{int} and are initialized to the default value of \( 0 \). Variables \( e \) and \( f \) are of type \texttt{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 \texttt{identifier-list} are declared with a type, but without an initialization expression as in
  ```chapel
  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
  ```chapel
  var v1: t; var v2: v1.type; var v3: v1.type;
  ```

- If the variables in the \texttt{identifier-list} are declared without a type, but with an initialization expression as in
  ```chapel
  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
  ```chapel
  var v1 = e; var v2 = v1; var v3 = v1;
  ```

- If the variables in the \texttt{identifier-list} are declared with both a type and an initialization expression as in
  ```chapel
  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 \texttt{readXX} on the first variable as in
  ```chapel
  var v1: t = e; var v2: v1.type = readXX(v1); var v3: v1.type = readXX(v1);
  ```

where the function \texttt{readXX} is defined as follows:

```chapel
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 \texttt{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 \texttt{sync} and \texttt{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 \texttt{identifier-list}. However, both \texttt{sync} and \texttt{single} variables require careful handling to avoid unintentional changes to their \texttt{full/empty} state.
8.2 Module Level Variables

Variables declared in statements that are in a module but not in a function or block within that module are module level variables. Module level variables can be accessed anywhere within that module after the declaration of that variable. If they are public, they can also be accessed in other modules that use that module.

8.3 Local Variables

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. Initializers of class types create storage that is not associated with any scope. Such storage can be reclaimed as described in §17.8.

8.4 Constants

Constants are divided into two categories: parameters, specified with the keyword \texttt{param}, are compile-time constants and constants, specified with the keyword \texttt{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 \texttt{~} 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 +, −, *, /, \texttt{&}, \texttt{&&}, \texttt{|}, \texttt{&}, ^, \texttt{<<}, \texttt{>>, =\,!, =\,<, >\,=\,=\,<, >\,<!, and > on operands that are bool or integral parameter expressions.
- Applications of the binary operators +, −, *, /, \texttt{**}, \texttt{==}, \texttt{!=}, \texttt{<=}, \texttt{>=}, <, and > on operands that are real, imaginary or complex parameter expressions.
Variables

- Applications of the string concatenation operator +, string comparison operators ==, !=, <=, >=, <, >, and the string length and byte methods 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 §13.7.4

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. Whether initialized explicitly or via its type’s default value, a constant stores the same value throughout its 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 at the module level.

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 an integer parameter `rank` to 2. At compile-time, this default value of `rank` can be overridden with another parameter expression, such as `3` or `2*n`, provided `n` itself is a parameter. The `rank` configuration variable can be used to write rank-independent code.

8.6 Ref Variables

A ref variable is a variable declared using the `ref` keyword. A ref variable serves as an alias to another variable, field or array element. The declaration of a ref variable must contain `initialization-part`, which specifies what is to be aliased and can be a variable or any lvalue expression.

Access or update to a ref variable is equivalent to access or update to the variable being aliased. For example, an update to a ref variable is visible via the original variable, and visa versa.
If the expression being aliased is a runtime constant variable, a formal argument with a `const ref` concrete intent (§13.5.1), or a call to a function with a `const ref` return intent (§13.7.2), the corresponding ref variable must be declared as `const ref`. Parameter constants and expressions cannot be aliased.

**Open issue.** The behavior of a `const ref` alias to a non-`const` variable is an open issue. The options include disallowing such an alias, disallowing changes to the variable while it can be accessed via a `const ref` alias, making changes visible through the alias, and making the behavior undefined.

**Example (refVariables.chpl).** For example, the following code:

```chapel
var myInt = 51;
ref refInt = myInt; // alias of a local or global variable
myInt = 62;
writeln("refInt = ", refInt);
refInt = 73;
writeln("myInt = ", myInt);

var myArr: [1..3] int = 51;
proc arrayElement(i) ref return myArr[i];
ref refToExpr = arrayElement(3); // alias to lvalue returned by a function
writeln("refToExpr = ", refToExpr);
refToExpr = 73;

const constArr: [1..3] int = 51..53;
const ref myConstRef = constArr[2]; // would be an error without ‘const’
writeln("myConstRef = ", myConstRef);
```

prints out:

```plaintext
refInt = 62
myInt = 73
refToExpr = 62
myArr[3] = 73
myConstRef = 52
```
9 Conversions

A conversion converts an expression of one type to another type, possibly changing its value. In certain cases noted below the source expression can be a type expression. 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.
- In a variable or field declaration, the initializing expression is converted to the type of the variable or field. The initializing expression is the r.h.s. of the = in the declaration, if present, or in the field initialization statement in an initializer.
- 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 param, in, const in, or an abstract intent (§13.5.2) with the semantics of in or const in.
- The actual type argument of a function call or an operator is converted to the corresponding formal argument of the type intent or the this formal of a type method. See §9.1.4.
- If the formal argument’s intent is out, the formal argument is converted to the type of the corresponding actual argument upon function return.
- The return or yield expression within a function without a ref return intent 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. See §9.1.5.

Implicit conversions are not applied for actual arguments passed to ref or const ref formal arguments.

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

- numeric and boolean types (§9.1.1).
Conversions

• numeric types in the special case when the expression’s value is a compile-time constant (§9.1.2), and
• class types (§9.1.3),
• class and generic types in certain cases (§9.1.4)
• from an integral or class type to bool in certain cases (§9.1.5).
• generic target types (§9.1.6)

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 $T_1$ to $T_2$ and from $T_2$ to $T_3$, that by itself does not allow an implicit conversion from $T_1$ to $T_3$.

9.1.1 Implicit Numeric and Bool 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 unintended 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.

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

### Target Type

<table>
<thead>
<tr>
<th>Source Type</th>
<th>bool(t)</th>
<th>uint(t)</th>
<th>int(t)</th>
<th>real(t)</th>
<th>imag(t)</th>
<th>complex(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool(s)</td>
<td>all s, t</td>
<td>all s, t</td>
<td>all s</td>
<td>2 ≤ t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uint(s)</td>
<td>s ≤ t</td>
<td>s &lt; t</td>
<td>s ≤ mant(t)</td>
<td>s ≤ mant(t/2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uint(64)</td>
<td></td>
<td></td>
<td>real(64)</td>
<td></td>
<td></td>
<td>complex(128)</td>
</tr>
<tr>
<td>int(s)</td>
<td></td>
<td></td>
<td>s ≤ t</td>
<td>s ≤ mant(t) + 1</td>
<td>s ≤ mant(t/2) + 1</td>
<td></td>
</tr>
<tr>
<td>int(64)</td>
<td></td>
<td></td>
<td>real(64)</td>
<td></td>
<td></td>
<td>complex(128)</td>
</tr>
<tr>
<td>real(s)</td>
<td></td>
<td></td>
<td>s ≤ t</td>
<td>s ≤ t/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>imag(s)</td>
<td></td>
<td></td>
<td></td>
<td>s ≤ t</td>
<td>s ≤ t/2</td>
<td></td>
</tr>
<tr>
<td>complex(s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s ≤ t</td>
<td></td>
</tr>
</tbody>
</table>

Here, \(\text{mant}(i)\) 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 (\(\text{uint}\), \(\text{complex}\), etc.) are the same as for their explicitly-sized counterparts.

#### 9.1.2 Implicit Compile-Time Constant Conversions

A parameter of numeric type can be implicitly converted to any other numeric type if the value of the parameter can be represented exactly by the target type. This rule does not allow conversions from \(\text{real}\) to \(\text{imag}\), or from \(\text{complex}\) to a non-complex type. It does allow conversions from \(\text{real}\) or \(\text{imag}\) to \(\text{complex}\).

#### 9.1.3 Implicit Class Conversions

An expression of class type can be implicitly converted to the borrow type; to a nilable type; or to a parent class type. The value \(\text{nil}\) can be implicitly converted to any nilable class type.

First, class types can be converted to the corresponding borrowed type. For example, \(\text{owned C}\) can be implicitly converted to \(\text{borrowed C}\), and \(\text{shared C}\) can be implicitly converted to \(\text{borrowed C}\). This coercion is equivalent to calling the .\(\text{borrow()}\) method. See §17.1.1 For example:

```chapel
Example (implicit-conversion-to-borrow.chpl).

class C { }
var c:owned C = new owned C();

proc f(arg: borrowed C) { }
f(c); // equivalent to f(c.borrow())
```

Second, an expression of non-nilable class type can be implicitly converted to the nilable class type. Continuing the above example:

\(^1\)For the IEEE 754 format, \(\text{mant}(32) = 24\) and \(\text{mant}(64) = 53\).
Example (implicit-conversion-to-nilable.chpl).

```chpl
var b: borrowed C = c.borrow();

proc g(arg: borrowed C?) { }

// equivalent to g(b:borrowed C?)
```

Third, an implicit conversion from class type \( \mathcal{D} \) to another class type \( \mathcal{C} \) is allowed when \( \mathcal{D} \) is a subclass of \( \mathcal{C} \).

Any combination of these three conversions is allowed.

### 9.1.4 Implicit Type Argument Conversions

An implicit type argument conversion applies only when a type actual is passed to a formal with the `type` intent. This includes the `this` formal of a type method. In this case, a subset of Implicit Class Conversions (§9.1.3) applies, in addition to Implicit Conversions To Generic Types (§9.1.6).

Future. The details are forthcoming.

### 9.1.5 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.

### 9.1.6 Implicit Conversions To Generic Types

When the target type \( \mathcal{T} \) is generic (§24.3), an implicit conversion is allowed when there is an instantiation of this type such that an implicit conversion is allowed between the source type and that instantiation by another rule in this section.

That instantiation is taken to be the instantiated type of the variable, field, formal argument, or the return type whose declared type is the generic type \( \mathcal{T} \).

The conversions in this subsection apply when the source is either an expression or a type expression.

### 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 not between all types.

The explicit conversions are a superset of the implicit conversions. In addition to the following definitions, an explicit conversion from a type to the same type is allowed for any type. Such conversion does not change the value of the expression.
9.2.1 Explicit Numeric Conversions

Explicit conversions are allowed from any numeric type or boolean to bytes or string, and vice-versa.

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 a int, uint, or real is converted to a bool, the result is false if the number was equal to 0 and true otherwise.

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(\(\ell\)) such that \(\ell > 2k\), the conversion is performed as if the original value is first converted to real(\(\ell/2\)) and then to \(\ell\).

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.

---

[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.
9.2.2 Explicit Tuple to Complex Conversion

A two-tuple of numerical values may be converted to a complex value. If the destination type is \texttt{complex(128)}, each member of the two-tuple must be convertible to \texttt{real(64)}. If the destination type is \texttt{complex(64)}, each member of the two-tuple must be convertible to \texttt{real(32)}. The first member of the tuple becomes the real part of the resulting complex value; the second member of the tuple becomes the imaginary part of the resulting complex value.

9.2.3 Explicit Enumeration Conversions

Explicit conversions are allowed from any enumerated type to \texttt{bytes} or \texttt{string} and vice-versa, and include \texttt{param} conversions. For enumerated types that are either concrete or semi-concrete (§7.2), conversions are supported between the enum’s constants and any numeric type or \texttt{bool}, including \texttt{param} conversions. For a semi-concrete enumerated type, if a conversion is attempted involving a constant with no underlying integer value, it will generate a compile-time error for a \texttt{param} conversion or an execution-time error otherwise.

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

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

When the target type is \texttt{bytes} or \texttt{string}, the value becomes the name of the enumerator.

When the source type is \texttt{bool}, enumerators corresponding to the values 0 and 1 in the underlying integer type are selected, corresponding to input values of \texttt{false} and \texttt{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 \texttt{complex} and \texttt{imag} types to an enumerated type is not permitted.

When the source type is \texttt{bytes} or \texttt{string}, the enumerator whose name matches value of the input is selected. If no such enumerator exists, an \texttt{IllegalArgumentException} is thrown.

9.2.4 Explicit Class Conversions

An expression of static class type \texttt{C} can be explicitly converted to a class type \texttt{D} provided that \texttt{C} is derived from \texttt{D} or \texttt{D} is derived from \texttt{C}.

When at run time the source expression refers to an instance of \texttt{D} or it subclass, its value is not changed. Otherwise, the cast fails and the result depends on whether or not the destination type is nilable. If the cast fails and the destination type is not nilable, the cast expression will throw a \texttt{classCastException}. If the cast fails and the destination type is nilable, as with \texttt{D?}, then the result will be \texttt{nil}.

An expression of class type can also be converted to a different nilability with a cast. For conversions from a nilable class type to a non-nilable class type, the cast will throw a \texttt{NilClassError} if the value was actually \texttt{nil}. 

In some cases a new variant of a class type needs to be computed that has different nilability or memory management strategy. Supposing that $T$ represents a class type, then these casts may compute a new type:

- $T$:owned - new management is owned, nilability from $T$
- $T$:shared - new management shared, nilability from $T$
- $T$:borrowed - new management borrowed, nilability from $T$
- $T$:unmanaged - new management unmanaged, nilability from $T$
- $T$:class - non-nilable type with specific concrete or generic management from $T$
- $T$:class? - nilable type with specific concrete or generic management from $T$
- $T$:owned class - non-nilable type with owned management
- $T$:owned class? - nilable type with owned management
- $T$:shared class - non-nilable type with shared management
- $T$:shared class? - nilable type with shared management
- $T$:borrowed class - non-nilable type with borrowed management
- $T$:borrowed class? - nilable type with borrowed management
- $T$:unmanaged class - non-nilable type with unmanaged management
- $T$:unmanaged class? - nilable type with unmanaged management

The conversions in this subsection apply when the source is either an expression or a type expression.

### 9.2.5 Explicit Range Conversions

An expression of stridable range type can be explicitly converted to an unstridable range type, changing the stride to 1 in the process.

### 9.2.6 Explicit Domain Conversions

An expression of stridable domain type can be explicitly converted to an unstridable domain type, changing all strides to 1 in the process.

### 9.2.7 Explicit String to Bytes Conversions

An expression of string type can be explicitly converted to a bytes. However, the reverse is not possible as a bytes can contain arbitrary bytes. Instead, bytes.decode() method should be used to produce a string from a bytes.
9.2.8 Explicit Type to String Conversions

A type expression can be explicitly converted to a string. The resultant string is the name of the type.

Example (explicit-type-to-string.chpl). For example:

```chapel
var x: real(64) = 10.0;
writeln(x.type:string);
```

This program will print out the string "real(64)".
10 Expressions

Chapel provides the following expressions:

expression:
  literal-expression
  nil-expression
  variable-expression
  enum-constant-expression
  call-expression
  type-expression
  iterable-call-expression
  member-access-expression
  new-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 §27
- module access §12.5.4
- tuple and tuple expand §16
- locale access §28.1.5
- mapped domain §29
- initializer calls §17.3
- nil §17.1.5
10.1 Literal Expressions

A literal value for any of the predefined types is a literal expression.

Literal expressions are given by the following syntax:

\[
\text{literal-expression:} \\
\quad \text{bool-literal} \\
\quad \text{integer-literal} \\
\quad \text{real-literal} \\
\quad \text{imaginary-literal} \\
\quad \text{string-literal} \\
\quad \text{bytes-literal} \\
\quad \text{range-literal} \\
\quad \text{domain-literal} \\
\quad \text{array-literal}
\]

Literal values for primitive types are described in §6.4.3. Literal range values are described in §20.3.1. Literal tuple values are described in §16.2. Literal values for domains are described in §21.2.1 and §21.2.2. Literal values for arrays are described in §22.2.1 and §22.2.2.

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:

\[
\text{variable-expression:} \\
\quad \text{identifier}
\]

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:

\[
\text{enum-constant-expression:} \\
\quad \text{enum-type . identifier}
\]

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

10.4 Parenthesized Expressions

A \textit{parenthesized-expression} is an expression that is delimited by parentheses as given by:

\[
\text{parenthesized-expression:} \\
\quad ( \text{expression} )
\]

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 §17.5 and §17.6 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 §24.1.5.

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:

```chpl
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:

```plaintext
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:

```plaintext
cast-expression:
expression : type-expression
```

The expression is converted to the specified type. A cast expression invokes the corresponding explicit conversion (§9.2). A resolution error occurs if no such conversion exists.

### 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, inout or ref intent, or that can be returned by a function with a ref return intent (§13.7.1). Valid lvalue expressions include the following:

- Variable expressions.
- Member access expressions.
- Call expressions of functions with a ref return intent.
- Indexing expressions.

LValue expressions are given by the following syntax:

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

The syntax is less restrictive than the definition above. For example, not all call-expressions are lvalues.
10.11 Precedence and Associativity

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>left</td>
<td>member access</td>
</tr>
<tr>
<td>()</td>
<td></td>
<td>function call or access</td>
</tr>
<tr>
<td>[ ]</td>
<td></td>
<td>function call or access</td>
</tr>
<tr>
<td>new</td>
<td>right</td>
<td>initializer call</td>
</tr>
<tr>
<td>owned</td>
<td>right</td>
<td>apply management strategy to a class</td>
</tr>
<tr>
<td>shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>borrowed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unmanaged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>postfix ?</td>
<td>left</td>
<td>compute a nilable class type</td>
</tr>
<tr>
<td>postfix !</td>
<td>left</td>
<td>assert non-nilable and borrow</td>
</tr>
<tr>
<td>:</td>
<td>left</td>
<td>cast</td>
</tr>
<tr>
<td>**</td>
<td>right</td>
<td>exponentiation</td>
</tr>
<tr>
<td>reduce</td>
<td>left</td>
<td>reduction</td>
</tr>
<tr>
<td>scan</td>
<td></td>
<td>scan</td>
</tr>
<tr>
<td>dmapped</td>
<td></td>
<td>domain map application</td>
</tr>
<tr>
<td>prefix !</td>
<td>right</td>
<td>logical negation</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>bitwise negation</td>
</tr>
<tr>
<td>*</td>
<td>left</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td></td>
<td>division</td>
</tr>
<tr>
<td>%</td>
<td></td>
<td>modulus</td>
</tr>
<tr>
<td>unary +</td>
<td>right</td>
<td>positive identity</td>
</tr>
<tr>
<td>unary -</td>
<td></td>
<td>negation</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left</td>
<td>left shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>left</td>
<td>right shift</td>
</tr>
<tr>
<td>&amp;</td>
<td>left</td>
<td>bitwise/logical and</td>
</tr>
<tr>
<td>^</td>
<td>left</td>
<td>bitwise/logical xor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>left</td>
</tr>
<tr>
<td>+</td>
<td>left</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>left</td>
<td>subtraction</td>
</tr>
<tr>
<td>..</td>
<td>left</td>
<td>range initialization</td>
</tr>
<tr>
<td>&lt;=</td>
<td>left</td>
<td>less-than-or-equal-to comparison</td>
</tr>
<tr>
<td>&gt;=</td>
<td>left</td>
<td>greater-than-or-equal-to comparison</td>
</tr>
<tr>
<td>&lt;</td>
<td>left</td>
<td>less-than comparison</td>
</tr>
<tr>
<td>&gt;</td>
<td>left</td>
<td>greater-than comparison</td>
</tr>
<tr>
<td>==</td>
<td>left</td>
<td>equal-to comparison</td>
</tr>
<tr>
<td>!=</td>
<td>left</td>
<td>not-equal-to comparison</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left</td>
<td>short-circuiting logical and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>by</td>
<td>left</td>
<td>range/domain stride application</td>
</tr>
<tr>
<td>#</td>
<td>left</td>
<td>range count application</td>
</tr>
<tr>
<td>align</td>
<td>left</td>
<td>range alignment</td>
</tr>
<tr>
<td>in</td>
<td>left</td>
<td>forall expression</td>
</tr>
<tr>
<td>if then else</td>
<td>left</td>
<td>conditional expression</td>
</tr>
<tr>
<td>forall do</td>
<td>left</td>
<td>forall expression</td>
</tr>
<tr>
<td>[ ]</td>
<td></td>
<td>forall expression</td>
</tr>
<tr>
<td>for do</td>
<td></td>
<td>for expression</td>
</tr>
<tr>
<td>sync single atomic</td>
<td>left</td>
<td>sync, single and atomic type</td>
</tr>
<tr>
<td>,</td>
<td>left</td>
<td>comma separated expressions</td>
</tr>
</tbody>
</table>
The above table summarizes operator and expression precedence and associativity. Operators and expressions listed earlier have higher precedence than those listed later.

**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:

```c
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.

In contrast to C, we give bitwise operations a higher precedence than binary addition/subtraction and comparison operators. This enables using the shift operators as shorthand for multiplication/division by powers of 2, and also makes it easier to extract and test a bitmapped field:

- $(x \&\& MASK) == MASK$ as $x \&\& MASK == MASK$
- $a + b * pow(2,y)$ as $a * b << y$

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:

- $\text{max reduce } A - \text{min reduce } A$ wants $(\text{max reduce } A) - (\text{min reduce } A)$
- $\text{max reduce } A * B$ wants $\text{max reduce } (A * B)$
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:

\[
\text{max reduce } A \times B \quad \text{requires} \quad \text{max reduce} \ (A \times B)
\]

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 parentheses 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:

- \(\text{unary-expression:}\)
  - \(\text{unary-operator expression}\)

- \(\text{unary-operator: one of}\)
  - \(+\)
  - \(-\)
  - \(~\)
  - \(!\)

The syntax of a binary expression is given by:

- \(\text{binary-expression:}\)
  - \(\text{expression binary-operator expression}\)

- \(\text{binary-operator: one of}\)
  - \(+\)
  - \(-\)
  - \(\times\)
  - \(%\)
  - \(\ast\)
  - \(^\)\)
  - \(<\)
  - \(\geq\)
  - \(==\)
  - \(<\lessgtr\)
  - \(\text{by #}\)

The operators are defined in subsequent sections.

### 10.13 Arithmetic Operators

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

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 explicit cast is required.
10.13.1 Unary Plus Operators

The unary plus operators are predefined as follows:

```plaintext
proc +(a: int(8)) : int(8)
proc +(a: int(16)) : int(16)
proc +(a: int(32)) : int(32)
proc +(a: int(64)) : int(64)
proc +(a: uint(8)) : uint(8)
proc +(a: uint(16)) : uint(16)
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:

```plaintext
proc -(a: int(8)) : int(8)
proc -(a: int(16)) : int(16)
proc -(a: int(32)) : int(32)
proc -(a: int(64)) : int(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:

\[
\begin{align*}
\textproc{+(a: \text{int}(8), b: \text{int}(8)): \text{int}(8)} \\
\textproc{+(a: \text{int}(16), b: \text{int}(16)): \text{int}(16)} \\
\textproc{+(a: \text{int}(32), b: \text{int}(32)): \text{int}(32)} \\
\textproc{+(a: \text{int}(64), b: \text{int}(64)): \text{int}(64)} \\
\textproc{+(a: \text{uint}(8), b: \text{uint}(8)): \text{uint}(8)} \\
\textproc{+(a: \text{uint}(16), b: \text{uint}(16)): \text{uint}(16)} \\
\textproc{+(a: \text{uint}(32), b: \text{uint}(32)): \text{uint}(32)} \\
\textproc{+(a: \text{uint}(64), b: \text{uint}(64)): \text{uint}(64)} \\
\textproc{+(a: \text{real}(32), b: \text{real}(32)): \text{real}(32)} \\
\textproc{+(a: \text{real}(64), b: \text{real}(64)): \text{real}(64)} \\
\textproc{+(a: \text{imag}(32), b: \text{imag}(32)): \text{imag}(32)} \\
\textproc{+(a: \text{imag}(64), b: \text{imag}(64)): \text{imag}(64)} \\
\textproc{+(a: \text{complex}(64), b: \text{complex}(64)): \text{complex}(64)} \\
\textproc{+(a: \text{complex}(128), b: \text{complex}(128)): \text{complex}(128)} \\
\textproc{+(a: \text{real}(32), b: \text{imag}(32)): \text{complex}(64)} \\
\textproc{+(a: \text{imag}(32), b: \text{real}(32)): \text{complex}(64)} \\
\textproc{+(a: \text{real}(64), b: \text{imag}(64)): \text{complex}(128)} \\
\textproc{+(a: \text{imag}(64), b: \text{real}(64)): \text{complex}(128)} \\
\textproc{+(a: \text{complex}(64), b: \text{imag}(32)): \text{complex}(64)} \\
\textproc{+(a: \text{complex}(64), b: \text{real}(32)): \text{complex}(64)} \\
\textproc{+(a: \text{complex}(128), b: \text{imag}(64)): \text{complex}(128)} \\
\textproc{+(a: \text{complex}(128), b: \text{real}(64)): \text{complex}(128)}
\end{align*}
\]

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 \text{uint}(64) and a value of type \text{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:

\[
\begin{align*}
&\text{proc } -(a:\text{ int}(8), b:\text{ int}(8)): \text{ int}(8) \\
&\text{proc } -(a:\text{ int}(16), b:\text{ int}(16)): \text{ int}(16) \\
&\text{proc } -(a:\text{ int}(32), b:\text{ int}(32)): \text{ int}(32) \\
&\text{proc } -(a:\text{ int}(64), b:\text{ int}(64)): \text{ int}(64) \\
&\text{proc } -(a:\text{ uint}(8), b:\text{ uint}(8)): \text{ uint}(8) \\
&\text{proc } -(a:\text{ uint}(16), b:\text{ uint}(16)): \text{ uint}(16) \\
&\text{proc } -(a:\text{ uint}(32), b:\text{ uint}(32)): \text{ uint}(32) \\
&\text{proc } -(a:\text{ uint}(64), b:\text{ uint}(64)): \text{ uint}(64) \\
&\text{proc } -(a:\text{ real}(32), b:\text{ real}(32)): \text{ real}(32) \\
&\text{proc } -(a:\text{ real}(64), b:\text{ real}(64)): \text{ real}(64) \\
&\text{proc } -(a:\text{ imag}(32), b:\text{ imag}(32)): \text{ imag}(32) \\
&\text{proc } -(a:\text{ imag}(64), b:\text{ imag}(64)): \text{ imag}(64) \\
&\text{proc } -(a:\text{ complex}(64), b:\text{ complex}(64)): \text{ complex}(64) \\
&\text{proc } -(a:\text{ complex}(128), b:\text{ complex}(128)): \text{ complex}(128) \\
&\text{proc } -(a:\text{ real}(32), b:\text{ imag}(32)): \text{ complex}(64) \\
&\text{proc } -(a:\text{ imag}(32), b:\text{ real}(32)): \text{ complex}(64) \\
&\text{proc } -(a:\text{ real}(64), b:\text{ imag}(64)): \text{ complex}(128) \\
&\text{proc } -(a:\text{ imag}(64), b:\text{ real}(64)): \text{ complex}(128) \\
&\text{proc } -(a:\text{ real}(32), b:\text{ complex}(64)): \text{ complex}(64) \\
&\text{proc } -(a:\text{ complex}(64), b:\text{ real}(32)): \text{ complex}(64) \\
&\text{proc } -(a:\text{ real}(64), b:\text{ complex}(128)): \text{ complex}(128) \\
&\text{proc } -(a:\text{ complex}(128), b:\text{ real}(64)): \text{ complex}(128) \\
&\text{proc } -(a:\text{ imag}(32), b:\text{ complex}(64)): \text{ complex}(64) \\
&\text{proc } -(a:\text{ complex}(64), b:\text{ imag}(32)): \text{ complex}(64) \\
&\text{proc } -(a:\text{ imag}(64), b:\text{ complex}(128)): \text{ complex}(128) \\
&\text{proc } -(a:\text{ complex}(128), b:\text{ imag}(64)): \text{ complex}(128)
\end{align*}
\]

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 \text{ uint}(64) from a value of type \text{ 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:

\[
\begin{align*}
\text{proc } & \ast (a: \text{int}(8), b: \text{int}(8)): \text{int}(8) \\
\text{proc } & \ast (a: \text{int}(16), b: \text{int}(16)): \text{int}(16) \\
\text{proc } & \ast (a: \text{int}(32), b: \text{int}(32)): \text{int}(32) \\
\text{proc } & \ast (a: \text{int}(64), b: \text{int}(64)): \text{int}(64) \\
\text{proc } & \ast (a: \text{uint}(8), b: \text{uint}(8)): \text{uint}(8) \\
\text{proc } & \ast (a: \text{uint}(16), b: \text{uint}(16)): \text{uint}(16) \\
\text{proc } & \ast (a: \text{uint}(32), b: \text{uint}(32)): \text{uint}(32) \\
\text{proc } & \ast (a: \text{uint}(64), b: \text{uint}(64)): \text{uint}(64) \\
\text{proc } & \ast (a: \text{real}(32), b: \text{real}(32)): \text{real}(32) \\
\text{proc } & \ast (a: \text{real}(64), b: \text{real}(64)): \text{real}(64) \\
\text{proc } & \ast (a: \text{imag}(32), b: \text{imag}(32)): \text{real}(32) \\
\text{proc } & \ast (a: \text{imag}(64), b: \text{imag}(64)): \text{real}(64) \\
\text{proc } & \ast (a: \text{complex}(64), b: \text{complex}(64)): \text{complex}(64) \\
\text{proc } & \ast (a: \text{complex}(128), b: \text{complex}(128)): \text{complex}(128) \\
\text{proc } & \ast (a: \text{real}(32), b: \text{imag}(32)): \text{imag}(32) \\
\text{proc } & \ast (a: \text{imag}(32), b: \text{real}(32)): \text{imag}(32) \\
\text{proc } & \ast (a: \text{real}(64), b: \text{imag}(64)): \text{imag}(64) \\
\text{proc } & \ast (a: \text{imag}(64), b: \text{real}(64)): \text{imag}(64) \\
\text{proc } & \ast (a: \text{real}(32), b: \text{complex}(64)): \text{complex}(64) \\
\text{proc } & \ast (a: \text{complex}(64), b: \text{real}(32)): \text{complex}(64) \\
\text{proc } & \ast (a: \text{real}(64), b: \text{complex}(128)): \text{complex}(128) \\
\text{proc } & \ast (a: \text{complex}(128), b: \text{real}(64)): \text{complex}(128) \\
\text{proc } & \ast (a: \text{imag}(32), b: \text{complex}(64)): \text{complex}(64) \\
\text{proc } & \ast (a: \text{complex}(64), b: \text{imag}(32)): \text{complex}(64) \\
\text{proc } & \ast (a: \text{imag}(64), b: \text{complex}(128)): \text{complex}(128) \\
\text{proc } & \ast (a: \text{complex}(128), b: \text{imag}(64)): \text{complex}(128)
\end{align*}
\]

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 \text{uint}(64) and a value of type \text{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:

```plaintext
proc /\(a:\text{int}(8), b:\text{int}(8)\): \text{int}(8)
proc /\(a:\text{int}(16), b:\text{int}(16)\): \text{int}(16)
proc /\(a:\text{int}(32), b:\text{int}(32)\): \text{int}(32)
proc /\(a:\text{int}(64), b:\text{int}(64)\): \text{int}(64)
proc /\(a:\text{uint}(8), b:\text{uint}(8)\): \text{uint}(8)
proc /\(a:\text{uint}(16), b:\text{uint}(16)\): \text{uint}(16)
proc /\(a:\text{uint}(32), b:\text{uint}(32)\): \text{uint}(32)
proc /\(a:\text{uint}(64), b:\text{uint}(64)\): \text{uint}(64)
proc /\(a:\text{real}(32), b:\text{real}(32)\): \text{real}(32)
proc /\(a:\text{real}(64), b:\text{real}(64)\): \text{real}(64)
proc /\(a:\text{imag}(32), b:\text{imag}(32)\): \text{real}(32)
proc /\(a:\text{imag}(64), b:\text{imag}(64)\): \text{real}(64)
proc /\(a:\text{complex}(64), b:\text{complex}(64)\): \text{complex}(64)
proc /\(a:\text{complex}(128), b:\text{complex}(128)\): \text{complex}(128)
proc /\(a:\text{real}(32), b:\text{imag}(32)\): \text{imag}(32)
proc /\(a:\text{imag}(32), b:\text{real}(32)\): \text{imag}(32)
proc /\(a:\text{real}(64), b:\text{imag}(64)\): \text{imag}(64)
proc /\(a:\text{imag}(64), b:\text{real}(64)\): \text{imag}(64)
proc /\(a:\text{real}(32), b:\text{complex}(64)\): \text{complex}(64)
proc /\(a:\text{complex}(64), b:\text{real}(32)\): \text{complex}(64)
proc /\(a:\text{real}(64), b:\text{complex}(128)\): \text{complex}(128)
proc /\(a:\text{complex}(128), b:\text{real}(64)\): \text{complex}(128)
proc /\(a:\text{imag}(32), b:\text{complex}(64)\): \text{complex}(64)
proc /\(a:\text{complex}(64), b:\text{imag}(32)\): \text{complex}(64)
proc /\(a:\text{imag}(64), b:\text{complex}(128)\): \text{complex}(128)
proc /\(a:\text{complex}(128), b:\text{imag}(64)\): \text{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 \text{uint}(64) by a value of type \text{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 \(q_1\) and \(q_2\) such that \(b \cdot q_1\) and \(b \cdot q_2\) 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:

```plaintext
proc % (a: int(8), b: int(8)): int(8)
proc % (a: int(16), b: int(16)): int(16)
proc % (a: int(32), b: int(32)): int(32)
proc % (a: int(64), b: int(64)): int(64)
proc % (a: uint(8), b: uint(8)): uint(8)
proc % (a: uint(16), b: uint(16)): uint(16)
proc % (a: uint(32), b: uint(32)): uint(32)
proc % (a: uint(64), b: uint(64)): 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:

```plaintext
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 \(\text{uint}(64)\) and a value of type \(\text{int}(64)\), 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:

```plaintext
proc ** (a: int(8), b: int(8)): int(8)
proc ** (a: int(16), b: int(16)): int(16)
proc ** (a: int(32), b: int(32)): int(32)
proc ** (a: int(64), b: int(64)): int(64)
proc ** (a: uint(8), b: uint(8)): uint(8)
proc ** (a: uint(16), b: uint(16)): uint(16)
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)
```

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 \(\text{uint}(64)\) by a value of type \(\text{int}(64)\), and vice versa.

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

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

10.14.1 Bitwise Complement Operators

The bitwise complement operators are predefined as follows:

\[
\begin{align*}
\text{proc} &\sim (a:\text{int}(8)):\text{int}(8) \\
\text{proc} &\sim (a:\text{int}(16)):\text{int}(16) \\
\text{proc} &\sim (a:\text{int}(32)):\text{int}(32) \\
\text{proc} &\sim (a:\text{int}(64)):\text{int}(64) \\
\text{proc} &\sim (a:\text{uint}(8)):\text{uint}(8) \\
\text{proc} &\sim (a:\text{uint}(16)):\text{uint}(16) \\
\text{proc} &\sim (a:\text{uint}(32)):\text{uint}(32) \\
\text{proc} &\sim (a:\text{uint}(64)):\text{uint}(64)
\end{align*}
\]

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:

\[
\begin{align*}
\text{proc} &\& (a:\text{bool}, b:\text{bool}):\text{bool} \\
\text{proc} &\& (a:\text{int}(?w), b:\text{int}(w)):\text{int}(w) \\
\text{proc} &\& (a:\text{uint}(?w), b:\text{uint}(w)):\text{uint}(w) \\
\text{proc} &\& (a:\text{int}(?w), b:\text{uint}(w)):\text{uint}(w) \\
\text{proc} &\& (a:\text{uint}(?w), b:\text{int}(w)):\text{uint}(w)
\end{align*}
\]

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.

\textit{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:

\[
\begin{align*}
&\text{proc } |(a: \text{ bool}, b: \text{ bool}): \text{ bool} \\
&\text{proc } |(a: \text{ int}(?w), b: \text{ int}(w)): \text{ int}(w) \\
&\text{proc } |(a: \text{ uint}(?w), b: \text{ uint}(w)): \text{ uint}(w) \\
&\text{proc } |(a: \text{ int}(?w), b: \text{ uint}(w)): \text{ uint}(w) \\
&\text{proc } |(a: \text{ uint}(?w), b: \text{ int}(w)): \text{ uint}(w)
\end{align*}
\]

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:

\[
\begin{align*}
&\text{proc } ^(a: \text{ bool}, b: \text{ bool}): \text{ bool} \\
&\text{proc } ^(a: \text{ int}(?w), b: \text{ int}(w)): \text{ int}(w) \\
&\text{proc } ^(a: \text{ uint}(?w), b: \text{ uint}(w)): \text{ uint}(w) \\
&\text{proc } ^(a: \text{ int}(?w), b: \text{ uint}(w)): \text{ uint}(w) \\
&\text{proc } ^(a: \text{ uint}(?w), b: \text{ int}(w)): \text{ uint}(w)
\end{align*}
\]

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 (§13.12).

The shift operators are predefined as follows:

```plaintext
proc <<(a: int(8), b): int(8)
proc <<(a: int(16), b): int(16)
proc <<(a: int(32), b): int(32)
proc <<(a: int(64), b): int(64)
proc <<(a: uint(8), b): uint(8)
proc <<(a: uint(16), b): uint(16)
proc <<(a: uint(32), b): uint(32)
proc <<(a: uint(64), b): uint(64)
proc >>>(a: int(8), b): int(8)
proc >>>(a: int(16), b): int(16)
proc >>>(a: int(32), b): int(32)
proc >>>(a: int(64), b): int(64)
proc >>>(a: uint(8), b): uint(8)
proc >>>(a: uint(16), b): uint(16)
proc >>>(a: uint(32), b): uint(32)
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 (§13.12).

10.16.1 The Logical Negation Operator

The logical negation operator is predefined for booleans and integers as follows:

```plaintext
proc !(a: bool): bool
proc !(a: int(?w)): bool
proc !(a: uint(?w)): bool
```

For the boolean form, the result is the logical negation of the operand. For the integer forms, the result is true if the operand is zero and false otherwise.
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

\[
\text{\texttt{\texttt{a && b}}}
\]

is evaluated as the expression

\[
\text{\texttt{if isTrue(a) then isTrue(b) else false}}
\]

The function \( \text{isTrue} \) is predefined over bool type as follows:

\[
\text{proc isTrue(a:bool) return a;}
\]

Overloading the logical and operator over other types is accomplished by overloading the \( \text{isTrue} \) 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

\[
\text{\texttt{a || b}}
\]

is evaluated as the expression

\[
\text{\texttt{if isTrue(a) then true else isTrue(b)}}
\]

The function \( \text{isTrue} \) is predefined over bool type as described in §10.16.2. Overloading the logical or operator over other types is accomplished by overloading the \( \text{isTrue} \) function over other types.
10.17  Relational Operators

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

10.17.1  Ordered Comparison Operators

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

```plaintext
proc <(a: int(8), b: int(8)): bool
proc <(a: int(16), b: int(16)): bool
proc <(a: int(32), b: int(32)): bool
proc <(a: int(64), b: int(64)): bool
proc <(a: uint(8), b: uint(8)): bool
proc <(a: uint(16), b: uint(16)): bool
proc <(a: uint(32), b: uint(32)): bool
proc <(a: uint(64), b: uint(64)): bool
proc <(a: int(64), b: uint(64)): bool
proc <(a: uint(64), b: int(64)): bool
proc <(a: real(32), b: real(32)): bool
proc <(a: real(64), b: real(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:

```plaintext
proc >(a: int(8), b: int(8)): bool
proc >(a: int(16), b: int(16)): bool
proc >(a: int(32), b: int(32)): bool
proc >(a: int(64), b: int(64)): bool
proc >(a: uint(8), b: uint(8)): bool
proc >(a: uint(16), b: uint(16)): bool
proc >(a: uint(32), b: uint(32)): bool
proc >(a: uint(64), b: uint(64)): bool
proc >(a: int(64), b: uint(64)): bool
proc >(a: uint(64), b: int(64)): bool
proc >(a: real(32), b: real(32)): bool
proc >(a: real(64), b: real(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:

```plaintext
proc <=(a: int(8), b: int(8)): bool
proc <=(a: int(16), b: int(16)): bool
proc <=(a: int(32), b: int(32)): bool
proc <=(a: int(64), b: int(64)): bool
proc <=(a: uint(8), b: uint(8)): bool
proc <=(a: uint(16), b: uint(16)): bool
proc <=(a: uint(32), b: uint(32)): bool
```
Expressions

proc <=(a: uint(64), b: uint(64)) : bool
proc <=(a: int(64), b: uint(64)) : bool
proc <=(a: uint(64), b: int(64)) : bool
proc <=(a: real(32), b: real(32)) : bool
proc <=(a: real(64), b: real(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(8), b: int(8)) : bool
proc >=(a: int(16), b: int(16)) : bool
proc >=(a: int(32), b: int(32)) : bool
proc >=(a: int(64), b: int(64)) : bool
proc >=(a: uint(8), b: uint(8)) : bool
proc >=(a: uint(16), b: uint(16)) : 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

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 == and != are predefined over bool and the numeric types as follows:

proc ==(a: int(8), b: int(8)) : bool
proc ==(a: int(16), b: int(16)) : bool
proc ==(a: int(32), b: int(32)) : bool
proc ==(a: int(64), b: int(64)) : bool
proc ==(a: uint(8), b: uint(8)) : bool
proc ==(a: uint(16), b: uint(16)) : 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
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:

- \( \text{proc} \) ==\( (a: \text{object}, b: \text{object}) : \text{bool} \)
- \( \text{proc} \) !=\( (a: \text{object}, b: \text{object}) : \text{bool} \)

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 §18.6.3.

The equality comparison operators are predefined over strings as follows:

- \( \text{proc} \) ==\( (a: \text{string}, b: \text{string}) : \text{bool} \)
- \( \text{proc} \) !=\( (a: \text{string}, b: \text{string}) : \text{bool} \)

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 Class Operators

The keywords owned, shared, borrowed, and unmanaged act as a prefix unary operator when specifying the management strategy for a class type. See §17.1.2.
The unary postfix operator ? results in the nilable variant of a class type. See §17.1.3.

The unary postfix operator ! asserts that the receiver is not storing nil and borrows from it. See §17.1.3.

### 10.19 Miscellaneous Operators

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

#### 10.19.1 The String Concatenation Operator

The string concatenation operator + is predefined for string arguments and returns a new string that is the concatenation of its arguments:

```
proc +(s0: string, s1: string): string
```

**Example (string-concat.chpl).** The code:

```chapel
var x: string = "hi";
var y: string = "there";
var z = x + y;
```

will cause z to be a new string containing the value "hi there".

#### 10.19.2 The By Operator

The operator by is predefined on ranges and rectangular domains. It is described in §20.5.1 for ranges and §21.8.2 for domains.

#### 10.19.3 The Align Operator

The operator align is predefined on ranges and rectangular domains. It is described in §20.5.2 for ranges and §21.8.3 for domains.

#### 10.19.4 The Range Count Operator

The operator # is predefined on ranges. It is described in §20.5.3.
10.20 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:

\[
\text{let-expression:} \\
\quad \text{let variable-declaration-list in expression}
\]

The scope of the variables is the let-expression.

Example (let.chpl). Let expressions are useful for defining variables in the context of an expression. In the code

\[
\text{let } x: \text{real - } a*b, \ y - x*x \ \text{in } 1/y
\]

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.

10.21 Conditional Expressions

A conditional expression is given by the following syntax:

\[
\text{if-expression:} \\
\quad \text{if expression then expression else expression} \\
\quad \text{if expression then expression}
\]

The conditional expression is evaluated in two steps. First, the expression following the \textit{if} keyword is evaluated. Then, if the expression evaluated to true, the expression following the \textit{then} keyword is evaluated and taken to be the value of this expression. Otherwise, the expression following the \textit{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 §10.22.1 and §27.2.4 respectively.

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

\[
\text{writehalf}(8); \\
\text{writehalf}(21); \\
\text{writehalf}(1000); \\
\text{proc writehalf}(i: \text{int}) \{ \\
\quad \text{var half = if (i \ % \ 2) then i/2 +1 else i/2;} \\
\quad \text{writeln("Half of ",i," is ",half);} \\
\}
\]

produces the output

\begin{verbatim}
Half of 8 is 4  
Half of 21 is 11  
Half of 1000 is 500
\end{verbatim}
10.22 For Expressions

A for expression is given by the following syntax:

\[
\text{for-expression:} \\
\text{for index-var declaration in iterable-expression do expression} \\
\text{for iterable-expression do expression}
\]

A for expression is an iterator that executes a for loop (§11.9), evaluates the body expression on each iteration of the loop, and yields each resulting value.

When a for expression is used to initialize a variable, such as

\[
\text{var X = for iterableExpression() do computeValue();}
\]

the variable will be inferred to have an array type. The array’s domain is defined by the iterable-expression following the same rules as for promotion, both in the regular case §27.5 and in the zipper case §27.5.2.

10.22.1 Filtering Predicates in For Expressions

A conditional expression that is immediately enclosed in a for expression and does not require an else clause 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.

When a for expression with a filtering predicate is captured into a variable, the resulting array has a 1-based one-dimensional domain.

Example (yieldPredicates.chpl). The code

\[
\text{var A = for i in 1..10 do if i % 3 != 0 then i;}
\]

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`
- `io-statement`
- `conditional-statement`
- `select-statement`
- `while-do-statement`
- `do-while-statement`
- `for-statement`
- `label-statement`
- `break-statement`
- `continue-statement`
- `param-for-statement`
- `use-statement`
- `defer-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 §13.8
- yield §23.2
- module declaration §12
• procedure declaration §13.2
• external procedure declaration §32.1.1
• exporting procedure declaration §32.1.2
• iterator declaration §23.1
• method declaration §17.1.7
• type declaration §7
• variable declaration §8.1
• remote variable declaration §28.2.1
• on statement §28.2
• cobegin, coforall, begin, sync, serial and atomic statements §26
• forall §27
• delete §17.8

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

\[
\text{block}\phantom{\text{statement}}: \\
\{ \text{statements}_{\text{opt}} \}
\]

\[
\text{statements}: \\
\text{statement} \\
\text{statement statements}
\]

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 (§26.5).

11.2 Expression Statements

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

\[
\text{expression}\phantom{\text{statement}}: \\
\text{variable}\phantom{\text{expression}}; \\
\text{member}\phantom{-}\text{access}\phantom{-}\text{expression} ; \\
\text{call}\phantom{-}\text{expression} ; \\
\text{new}\phantom{-}\text{expression} ; \\
\text{let}\phantom{-}\text{expression} ;
\]
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:

\[
\text{assignment-statement:} \quad \text{lvalue-expression assignment-operator expression}
\]

\[
\text{assignment-operator:} \quad \text{one of}
\]

\[
= \; += \; -= \; *= \; /= \; %= \; &= \; |= \; ^= \; &&= \; ||= \; <<= \; >>=
\]

The assignment operators that contain a binary operator symbol as a prefix are compound assignment operators. The remaining assignment operator = is called simple assignment.

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 left-hand side is of a numerical type, there is an implicit conversion (§9.1) of the right-hand side expression to the type of the left-hand side expression. Additionally, for simple assignment, if the left-hand side is of Boolean type, the right-hand side is implicitly converted to the type of the left-hand side (i.e. a bool(?w) with the same width w).

For simple assignment, the validity and semantics of assigning between classes (§17.7.1), records (§18.6.2), unions (§19.3), tuples (§16.5), ranges (§20.4.1), domains (§21.8.1), and arrays (§22.5) are discussed in these later sections.

A compound assignment is shorthand for applying the binary operator to the left- and right-hand side expressions and then assigning the result to the left-hand side expression. For numerical types, 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.

For all other compound assignments, Chapel provides a completely generic catch-all implementation defined in the obvious way. For example:

\[
\text{inline proc +=(ref lhs, rhs) \{}
\quad \text{lhs = lhs} + \text{rhs;}
\quad \}
\]

Thus, compound assignment can be used with operands of arbitrary types, provided that the following provisions are met: If the type of the left-hand argument of a compound assignment operator \(\text{op}\) is \(L\) and that of the right-hand argument is \(R\), then a definition for the corresponding binary operator \(\text{op}\) exists, such that \(L\) is coercible to the type of its left-hand formal and \(R\) is coercible to the type of its right-hand formal. Further, the result of \(\text{op}\) must be coercible to \(L\), and there must exist a definition for simple assignment between objects of type \(L\).

Both simple and compound assignment operators can be overloaded for different types using operator overloading (§13.12). In such an overload, the left-hand side expression should have ref intent and be modified within the body of the function. The return type of the function should be void.
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).

The swap operator can be overloaded for different types using operator overloading (§13.12).

\[
\text{swap–statement:} \\
\text{\hspace{1em} lvalue–expression swap–operator lvalue–expression}
\]

\[
\text{swap–operator:} \\
\text{\hspace{1em} <=>}
\]

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

*Example.* When resolved to the default swap operator, the following swap statement

\[
\text{var } a, b: \text{real}; \\
a \text{ <=> } b;
\]

is semantically equivalent to:

\[
\text{const } t = b; \\
b = a; \\
a = t;
\]

11.5 The I/O Statement

The I/O operator indicates writing to the left-hand-side the value in the right-hand-side; or reading from the left-hand-side and storing the result in the variable on the right-hand-side. This operator can be chained with other I/O operator calls.

The I/O operator can be overloaded for different types using operator overloading (§13.12).

\[
\text{io–statement:} \\
\text{\hspace{1em} io–expression io–operator expression}
\]

\[
\text{io–expression:} \\
\text{\hspace{1em} expression} \\
\text{\hspace{1em} io–expression io–operator expression}
\]

\[
\text{io–operator:} \\
\text{\hspace{1em} <~>}
\]

See the module documentation on I/O for details on how to use the I/O statement.

*Example.* In the example below,

\[
\text{var } w = \text{opentmp().writer();} \text{ \textbar a channel} \\
\text{var } a: \text{real}; \\
\text{var } b: \text{int}; \\
w <~> a <~> b;
\]

the I/O operator is left-associative and indicates writing \(a\) and then \(b\) to \(w\) in this case.
11.6 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
  if expression block-statement else-part

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.

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

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. The statement in the else-clause can be a conditional statement, too.

Example (conditionals.chpl). The following function prints `two` when `x` is 2 and `B`, `four` when `x` is 4.

```chpl
proc condttest(x: int) {
  if x > 3 then
    if x > 5 then
      write("A,");
    else
      write("B,");
  if x == 2 then
    writeln("two");
  else if x == 4 then
    writeln("four");
  else
    writeln("other");
}
```
11.7 The Select Statement

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

```start
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
  otherwise do statement
expression-list:
  expression
  expression , expression-list
```

The expression that follows the keyword `select`, the select expression, is evaluated once and its value is then compared with the list of case expressions following each `when` keyword. These values are compared 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` or the `otherwise-statement` has its own scope whether or not an explicit block surrounds it.

11.8 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:

```start
while-do-statement:
  while expression do statement
  while expression block-statement
```

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

```start
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 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. The code

```chapel
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);
}
```

produces the output

```
Scope of do while loop:
12
Scope of while loop:
```
Chapel do-while loops differ from those found in most other languages in one important regard. If the body of a do-while statement is a block statement and new variables are defined within that block statement, then the scope of those variables extends to cover the loop’s termination expression.

**Example (do-while.chpl)**. The following example demonstrates that the scope of the variable \( t \) includes the loop termination expression.

```chpl
var i = 0;
do {
    var t = i;
i += 1;
    writeln(t);
} while (t != 5);
```

produces the output

```
0
1
2
3
4
5
```

### 11.9 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 iterable-expression do statement
    for index-var-declaration in iterable-expression block-statement
    for iterable-expression do statement
    for iterable-expression block-statement

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

iterable-expression:
    expression
    zip ( expression-list )
```

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 destruct the values returned by the iterator expression, as described in §16.6.3

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

If the iterable-expression begins with the keyword `zip` followed by a parenthesized expression-list, the listed expressions must support zipper iteration.
11.9.1 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

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

is

```
1 4 2 5 3 6
```

11.9.2 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-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
```

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.
11.10 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:

\[
\text{break-statement:} \quad \text{break identifier}_{\text{opt}}; \\
\text{continue-statement:} \quad \text{continue identifier}_{\text{opt}}; \\
\text{label-statement:} \quad \text{label identifier statement}
\]

A break statement cannot be used to exit a parallel loop §27.1

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.

\[
\text{label} \quad \text{outer} \quad \text{for } i \text{ in } 1..n \{ \\
\text{for } j \text{ in } 1..n \{ \\
\text{if } A[i, j] == \text{findVal} \{ \\
\text{writeln("index: ", (i, j), " matches."); } \\
\text{continue} \quad \text{outer; } \\
\}
\}
\]
11.11 The Use Statement

The use statement provides access to the constants in an enumerated type or to the public symbols of a module without
the need to use a fully qualified name. When using a module, the statement also ensures that the module symbol itself
is visible within the current scope (top-level modules are not otherwise visible without a use).

The syntax of the use statement is given by:

```
use statement:
  privacy-specifier_opt, use module-or-enum-name-list;

module-or-enum-name-list:
  module-or-enum-name limitation-clause_opt
  module-or-enum-name, module-or-enum-name-list

module-or-enum-name:
  identifier
  identifier.module-or-enum-name

limitation-clause:
  except exclude-list
  only rename-list_opt

exclude-list:
  identifier-list
  *

rename-list:
  rename-base
  rename-base.rename-list

rename-base:
  identifier as identifier
  identifier
```

For example, the program

```
Example (use1.chpl).

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

module M2 {
  use M1;
  proc main() {
    writeln("In M2’s main.");
    M1.foo();
  }
}
```

prints out

```
In M2’s main.
In M1’s foo.
```
This program is equivalent to:

**Example (use2.chpl).**

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

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

which also prints out

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

The names that are imported by a use statement are inserted into a new scope that immediately encloses the scope within which the statement appears. This implies that the position of the use statement within a scope has no effect on its behavior. If a scope includes multiple use statements then the imported names are inserted into a common enclosing scope.

An error is signaled if multiple enumeration constants or public module-level symbols would be inserted into this enclosing scope with the same name, and that name is referenced by other statements in the same scope as the use.

Use statements are transitive by default: if a module A uses a module B, and module B contains a use of a module or enumerated type C, then C’s public symbols may also be visible within A. The exception to this occurs when B has public symbols that shadow symbols with the same name in C, or when the use of C has been declared explicitly private. If a use statement is declared to be private, then the symbols it makes visible will only be visible to the scope containing the use.

This notion of transitivity extends to the case in which a scope imports symbols from multiple modules or constants from multiple enumeration types. For example if a module A uses modules B1, B2, B3 and modules B1, B2, B3 use modules C1, C2, C3 respectively, then all of the public symbols in B1, B2, B3 have the potential to shadow the public symbols of C1, C2, and C3. However an error is signaled if C1, C2, C3 have public module level definitions of the same symbol.

An optional limitation-clause may be provided to limit the symbols made available by a given use statement. If an except list is provided, then all the visible but unlisted symbols in the module or enumerated type will be made available without prefix. If an only list is provided, then just the listed visible symbols in the module or enumerated type will be made available without prefix. All visible symbols not provided via these limited use statements are still accessible by prefixing the access with the name of the module or enumerated type. It is an error to provide a name in a limitation-clause that does not exist or is not visible in the respective module or enumerated type.

If a type is specified in the limitation-clause, then the type’s fields and methods are treated similarly to the type name. These fields and methods cannot be specified in a limitation-clause on their own.

If an only list is left empty or except is followed by * then no symbols are made available to the scope without prefix. However, any methods or fields defined within a module used in this way will still be accessible on instances of the type. For example:
Example (limited-access.chpl).

```chapel
module M1 {
    record A {
        var x = 1;
        proc foo() {
            writeln("In A.foo()");
        }
    }
}
module M2 {
    proc main() {
        use M1 only;
        var a = new M1.A(3); // Only accessible via the module prefix
        writeln(a.x); // Accessible because we have a record instance
        a.foo(); // Ditto
    }
}
```

will print out

```
3
In A.foo()
```

Within an `only` list, a visible symbol from that module may optionally be given a new name using the `as` keyword. This new name will be usable from the scope of the use in place of the old name unless the old name is additionally specified in the `only` list. If a use which renames a symbol is present at module scope, uses of that module will also be able to reference that symbol using the new name instead of the old name. Renaming does not affect accesses to that symbol via the source module’s or enumerated type’s prefix, nor does it affect uses of that module or enumerated type from other contexts. It is an error to attempt to rename a symbol that does not exist or is not visible in the respective module or enumerated type, or to rename a symbol to a name that is already present in the same `only` list. It is, however, perfectly acceptable to rename a symbol to a name present in the respective module or enumerated type which was not specified via that `only` list.

If a use statement mentions multiple modules or enumerated types or a mix of these symbols, only the last module or enumerated type can have a `limitation-clause`. Limitation clauses are applied transitively as well - in the first example, if module A’s use of module B contains an `except` or `only` list, that list will also limit which of C’s symbols are visible to A.

For more information on enumerated types, please see §7.2. For use statement rules which are only applicable to modules, please see §12.5.3. For more information on modules in general, please see §11.12

**11.12 The Defer Statement**

A `defer` statement declares a clean-up action to be run when exiting a block. `defer` is useful because the clean-up action will be run no matter how the block is exited.

The syntax is:

```
defer-statement:
    defer statement
```
At a given place where control flow exits a block, the in-scope `defer` statements and the local variables will be handled in reverse declaration order. Handling a `defer` statement consists of executing the contained clean-up action. Handling a local variable consists of running its deinitializer if it is of record type.

When an iterator contains a `defer` statement at the top level, the associated clean-up action will be executed when the loop running the iterator exits. `defer` actions inside a loop body are executed when that iteration completes.

The following program demonstrates a simple use of `defer` to create an action to be executed when returning from a function:

**Example (defer1.chpl).**

```chapel
class Integer {
  var x:int;
}
proc deferInFunction() {
  var c = new unmanaged Integer(1);
  writeln("created ", c);
  defer {
    writeln("defer action: deleting ", c);
    delete c;
  }
  // ... (function body, possibly including return statements)
  // The defer action is executed no matter how this function returns.
}
deferInFunction();
```

produces the output

```
created {x = 1}
defer action: deleting {x = 1}
```

The following example uses a nested block to demonstrate that `defer` is handled when exiting the block in which it is contained:

**Example (defer2.chpl).**

```chapel
class Integer {
  var x:int;
}
proc deferInNestedBlock() {
  var i = 1;
  writeln("before inner block");
  {
    var c = new unmanaged Integer(i);
    writeln("created ", c);
    defer {
      writeln("defer action: deleting ", c);
      delete c;
    }
    writeln("in inner block");
    // note, defer action is executed no matter how this block is exited
  }
  writeln("after inner block");
}
deferInNestedBlock();
```

produces the output
Lastly, this example shows that when `defer` is used in a loop, the action will be executed for every loop iteration, whether or not loop body is exited early.

**Example (defer3.chpl).**

```chapel
class Integer {
    var x:int;
}
proc deferInLoop() {
    for i in 1..10 {
        var c = new unmanaged Integer(i);
        writeln("created ", c);
        defer {
            writeln("defer action: deleting ", c);
            delete c;
        }
        writeln(c);
        if i == 2 then break;
    }
}
deferInLoop();
```

produces the output

```
created {x = 1}
{x = 1}
defer action: deleting {x = 1}
created {x = 2}
{x = 2}
defer action: deleting {x = 2}
```

### 11.13 The Empty Statement

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

```chapel
empty-statement:
    ;
```
12 Modules

Chapel supports modules to manage namespaces. 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 §12.3 Nested modules are described in §12.4 The visibility of a module’s symbols by users of the module is described in §12.5.2. The execution of a program and module initialization/deinitialization are described in §12.6.

12.1 Module Definitions

A module is declared with the following syntax:

```plaintext
module-declaration-statement:
  privacy-specifier, prototype-specifier, module module-identifier block-statement
```

```plaintext
privacy-specifier:
  private
  public
```

```plaintext
prototype-specifier:
  prototype
```

```plaintext
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 module-scope symbols. The visibility of a module is defined by its privacy-specifier (§12.5.1).

Module declaration statements are only legal as file-scope or module-scope statements. For example, module declaration statements may not occur within block statements, functions, classes, or records.

Any module declaration that is not contained within another module creates a top-level module. Module declarations within other modules create nested modules (§12.4).

12.2 Prototype Modules

Modules that are declared with the `prototype` keyword use relaxed rules for error handling and `nil` checking. These relaxed rules are appropriate for programs in the early stages of development but are not appropriate for libraries. In particular, within a prototype module:

- errors that are not handled will terminate the program (see §15.8.1)
- methods on nilable class instances can be called - these will implicitly convert to non-nilable class instances (see §17.1.3)
12.3 Files and Implicit Modules

Multiple modules can be defined within 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, MX and MY.

```chapel
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 module-scope symbols x and printX, while MY defines module-scope symbols y and printY.

For any file that contains file-scope statements other than module declarations, the file itself is treated as a module declaration. In this case, the module is implicit. Implicit modules are always prototype modules. An implicit module 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 implicit.chpl, defines an implicitly named module called implicit.

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

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

Module implicit defines the module-scope symbols x, y, printX, and printY.

12.4 Nested Modules

A nested module (or sub-module) is a module that is defined within another module, known as the outer, or parent, module. A nested module automatically has access to all of the symbols in its outer module. However, an outer module needs to explicitly name or use a nested module in order to access 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 file-scope statements result in nested modules.

Example (nested.chpl). The following file, named nested.chpl, defines an implicitly named module called nested, 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);
}

12.5 Access of Module Contents

A module’s contents can be accessed by code outside of that module depending on the visibility of the module itself (§12.5.1) and the visibility of each individual symbol (§12.5.2). This can be done via the use statement (§12.5.3) or qualified naming (§12.5.4).

12.5.1 Visibility Of A Module

A top-level module is available for use (§12.5.3) anywhere. The visibility of a nested module is subject to the rules of §12.5.2 where the nested module is considered a "module-scope symbol" of its outer module.

12.5.2 Visibility Of A Module’s Symbols

A symbol defined at module scope is visible from outside the module when the privacy-specifier of its definition is public or is omitted (i.e. by default). When a module-scope symbol is declared private, it is not visible outside of that module. A symbol’s visibility inside its module is controlled by normal lexical scoping and is not affected by its privacy-specifier. When a module’s symbol is visible (§12.5.1), the visible symbols it contains are accessible via the use statement (§12.5.3) or qualified naming (§12.5.4).
12.5.3 Using Modules

The `use` statement provides the primary means of accessing a module’s symbols from outside of the module. Use statements make both the module’s name and its public symbols available for reference within a given scope. For top-level modules, a use statement is required before referring to the module’s name or the symbols it contains within a given lexical scope.

Use statements can also restrict or rename the set of module symbols that are available within the scope. For further information about use statements, see §11.11.

12.5.4 Qualified Naming of Module Symbols

When a module’s symbol is visible—via a use statement, or lexically for nested modules—its public symbols can be referred to via qualified naming with the following syntax:

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

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

This allows two symbols that have the same name to be distinguished based on the name of their module. Using qualified 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. Qualified naming can be used to disambiguate the symbols in this case.

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

```chapel
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. The call could be qualified via M1.printY() or M3.printY() to resolve this ambiguity.

12.5.5 Module Initialization

Module initialization occurs at program start-up. All module-scope statements within a module other than function and type declarations are executed during module initialization. Modules that are not referred to, including both top-level modules and sub-modules, will not be initialized.

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 §12.6.2

12.5.6 Module Deinitialization

Module deinitialization occurs at program tear-down. During module deinitialization:

- If the module contains a deinitializer, which is a module-scope function named deinit(), it is executed first.
- If the module declares global variables, they are deinitialized in the reverse declaration order.

Module deinitialization order is discussed in §12.6.3

12.6 Program Execution

Chapel programs start by initializing all modules and then executing the main function (§12.6.1).
12.6.1 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.

Example (main-module.chpl). Because it defines two main functions, the following code will yield an error unless a main module is specified on the command line.

```chapel
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, a default main function is created if the program does not contain a user-defined main function. The default main function is equivalent to

```chapel
proc main() {}
```

Example (no-main.chpl). The code

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

is a legal and complete Chapel program. The startup code for a Chapel program first calls the module initialization code for the main module and then calls main(). This program’s initialization function is the file-scope writeln() statement. The module declaration is taken to be the entire file, as described in §12.3.
12.6.2 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 statement 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, the parent module and its uses are initialized before the nested module and its uses.

*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.

12.6.3 Module Deinitialization Order

Module deinitialization is performed in the reverse order of module initialization, as specified in §12.6.2.
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 §23
- methods (when bound to a class) §17.1.7
- function calls §13.1
- various aspects of defining a procedure §13.2 §13.11
- calling external functions from Chapel §32.1.1
- calling Chapel functions from external functions §32.1.2
- determining the function to invoke for a given call site: function and operator overloading §13.12 function resolution §13.13

13.1 Function Calls

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

\[
\text{call-expression:}
\]
\[
\text{lvalexpression ( named-expression-list )}
\]
\[
\text{lvalexpression [ named-expression-list ]}
\]
\[
\text{parenthesesless-function-identifier}
\]

\[
\text{named-expression-list:}
\]
\[
\text{named-expression}
\]
\[
\text{named-expression , named-expression-list}
\]
Procedures

A named-expression is resolved to a particular function according to the algorithm for function resolution described in §13.13.

Functions can be called using either parentheses or brackets.

Rationale. This provides an opportunity to blur the distinction between an array access and a function call and thereby exploit a possible space/time tradeoff.

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 §17.6.

13.2 Procedure Definitions

Procedures are defined with the following syntax:

```
procedure-declaration-statement:
    privacy-specifier_opt procedure-kind_opt proc function-name argument-list_opt return-intent_opt return-type_opt where-clause_opt function-body
```

```
procedure-kind:
    inline
    export
    extern
    override
```

```
function-name:
    identifier
    operator-name
```

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

```
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-expression
  : ? identifier_opt

default-expression:
  = expression

variable-argument-expression:
  ... expression
  ... ? identifier_opt
  ...

formal-intent:
  const
  const in
  const ref
  in
  out
  inout
  ref
  param
  type

return-intent:
  const
  const ref
  ref
  param
  type

return-type:
  : type-expression

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 §16.6.4.

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 `const`, `const in`, `const ref`, `in`, `out`, `inout` and `ref` are discussed in §13.5. The intents `param` and `type` make a function generic and are discussed in §24.1. If the formal argument’s type is omitted, generic, or prefixed with a question mark, the function is also generic and is discussed in §24.1.

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

The `return–intent` can be used to indicate how the value is returned from a function. `return–intent` is described further in §13.7.

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 §13.9. A type function may not specify a return type.

The `where–clause` is optional and is discussed in §13.10.

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

The optional `privacy–specifier` keywords indicate the visibility of module level procedures to outside modules. By default, procedures are publicly visible. More details on visibility can be found in §12.5.2.

The linkage specifier `inline` indicates that the function body must be inlined at every call site.

Rationale. A Chapel compiler is permitted to inline any function if it determines there is likely to be a performance benefit to do so. Hence an error must be reported if the compiler is unable to inline a procedure with this specifier. One example of a preventable inlining error is to define a sequence of inlined calls that includes a cycle back to an inlined procedure.

See the chapter on interoperability (§32) 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

```chapel
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

A formal argument’s intent (§13.5) specifies how the actual argument is passed to the function. If no intent is specified, the default intent (§13.5.2) is applied, resulting in type-dependent behavior.

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).* Running the code

```chapel
proc foo(x: int, y: int) { writeln(x); writeln(y); }
foo(x=2, y=3);
foo(y=3, x=2);
```

will produce the output

```
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. Note though that the default value is evaluated in the same scope as the called function. Default value expressions can refer to previous formal arguments or to variables that are visible to the scope of the function definition.

*Example (default-values.chpl).* The code

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

writes out
Default values are specified for the formal arguments \( x \) and \( y \). The three calls to \texttt{foo} are equivalent to the following three calls where the actual arguments are explicit: \texttt{foo(5, 7)}, \texttt{foo(7, 7)}, and \texttt{foo(5, 5)}. The example \texttt{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 Argument Intents

Argument intents specify how an actual argument is passed to a function where it is represented by the corresponding formal argument.

Argument intents are categorized as being either \textit{concrete} or \textit{abstract}. Concrete intents are those in which the semantics of the intent keyword are independent of the argument’s type. Abstract intents are those in which the keyword (or lack thereof) expresses a general intention that will ultimately be implemented via one of the concrete intents. The specific choice of concrete intent depends on the argument’s type and may be implementation-defined. Abstract intents are provided to support productivity and code reuse.

#### 13.5.1 Concrete Intents

The concrete intents are \texttt{in}, \texttt{out}, \texttt{inout}, \texttt{ref}, \texttt{const in}, and \texttt{const ref}.

**The In Intent**

When \texttt{in} is specified as the intent, the formal argument represents a variable that is copy-initialized with the value of the actual argument. For example, for integer arguments, the formal argument will store a copy of the actual argument. An implicit conversion occurs from the actual argument to the type of the formal. The formal can be modified within the function, but such changes are local to the function and not reflected back to the call site.

**The Out Intent**

When \texttt{out} is specified as the intent, the actual argument is ignored when the call is made, but when the function returns, the formal argument is copied back to the actual argument. An implicit conversion occurs from the type of the formal to the type of the actual. The actual argument must be a valid lvalue. The formal argument is initialized to its default value if one is supplied, or to its type’s default value otherwise. The formal argument can be modified within the function.
Procedures

The Inout Intent

When `inout` is specified as the intent, the actual argument is copied into the formal argument as with the `in` intent and then copied back out as with the `out` intent. The actual argument must be a valid lvalue. The formal argument can be modified within the function. The type of the actual argument must be the same as the type of the formal.

The Ref Intent

When `ref` is specified as the intent, the actual argument is passed by reference. Any reads of, or modifications to, the formal argument are performed directly on the corresponding actual argument at the call site. The actual argument must be a valid lvalue. The type of the actual argument must be the same as the type of the formal.

The ref intent differs from the inout intent in that the inout intent requires copying from/to the actual argument on the way in/out of the function, while ref allows direct access to the actual argument through the formal argument without copies. Note that concurrent modifications to the ref actual argument by other tasks may be visible within the function, subject to the memory consistency model.

The Const In Intent

The const in intent is identical to the in intent, except that modifications to the formal argument are prohibited within the function.

The Const Ref Intent

The const ref intent is identical to the ref intent, except that modifications to the formal argument are prohibited within the dynamic scope of the function. Note that concurrent tasks may modify the actual argument while the function is executing and that these modifications may be visible to reads of the formal argument within the function’s dynamic scope (subject to the memory consistency model).

Summary of Concrete Intents

The following table summarizes the differences between the concrete intents:

<table>
<thead>
<tr>
<th></th>
<th>in</th>
<th>out</th>
<th>inout</th>
<th>ref</th>
<th>const in</th>
<th>const ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>copied in on function call?</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>copied out on function return?</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>refers to actual argument?</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>formal can be read?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>formal can be modified?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>local changes affect the actual?</td>
<td>no</td>
<td>on return</td>
<td>on return</td>
<td>immediately</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
13.5.2 Abstract Intents

The abstract intents are `const` and the *default intent* (when no intent is specified).

Abstract Intents Table

The following table summarizes what these abstract intents mean for each type:

<table>
<thead>
<tr>
<th>type</th>
<th>meaning of <code>const</code> intent</th>
<th>meaning of default intent</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>const in</td>
<td>const in</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>const in</td>
<td>const in</td>
<td></td>
</tr>
<tr>
<td>uint</td>
<td>const in</td>
<td>const in</td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>const in</td>
<td>const in</td>
<td></td>
</tr>
<tr>
<td>imag</td>
<td>const in</td>
<td>const in</td>
<td></td>
</tr>
<tr>
<td>complex</td>
<td>const in</td>
<td>const in</td>
<td></td>
</tr>
<tr>
<td>range</td>
<td>const in</td>
<td>const in</td>
<td></td>
</tr>
<tr>
<td>owned class</td>
<td>const ref</td>
<td>const ref</td>
<td></td>
</tr>
<tr>
<td>shared class</td>
<td>const ref</td>
<td>const ref</td>
<td></td>
</tr>
<tr>
<td>borrowed class</td>
<td>const in</td>
<td>const in</td>
<td></td>
</tr>
<tr>
<td>unmanaged class</td>
<td>const in</td>
<td>const in</td>
<td></td>
</tr>
<tr>
<td>atomic</td>
<td>const ref</td>
<td>ref</td>
<td></td>
</tr>
<tr>
<td>single</td>
<td>const ref</td>
<td>ref</td>
<td></td>
</tr>
<tr>
<td>sync</td>
<td>const ref</td>
<td>ref</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>const ref</td>
<td>const ref</td>
<td></td>
</tr>
<tr>
<td>bytes</td>
<td>const ref</td>
<td>const ref</td>
<td></td>
</tr>
<tr>
<td>record</td>
<td>const ref</td>
<td>const ref</td>
<td></td>
</tr>
<tr>
<td>union</td>
<td>const ref</td>
<td>const ref</td>
<td></td>
</tr>
<tr>
<td>dmap</td>
<td>const ref</td>
<td>const ref</td>
<td></td>
</tr>
<tr>
<td>domain</td>
<td>const ref</td>
<td>const ref</td>
<td></td>
</tr>
<tr>
<td>array</td>
<td>const ref</td>
<td>ref/const ref</td>
<td></td>
</tr>
</tbody>
</table>

The Const Intent

The `const` intent specifies the intention that the function will not and cannot modify the formal argument within its dynamic scope. Whether the actual argument will be passed by `const in` or `const ref` intent depends on its type. In general, small values, such as scalar types, will be passed by `const in`; while larger values, such as domains and arrays, will be passed by `const ref` intent. The table earlier in this sub-section lists the meaning of the const intent for each type.

The Default Intent

When no intent is specified for a formal argument, the *default intent* is applied. It is designed to take the most natural/least surprising action for the argument, based on its type. The table earlier in this sub-section lists the meaning of the default intent for each type.
Default argument passing for tuples generally matches the default argument passing strategy that would be applied if each tuple element was passed as a separate argument.

*Open issue.* How tuples should be handled under default intents is an open issue; particularly for heterogeneous tuples whose components would fall into separate categories in the table above. One proposed approach is to apply the default intent to each component of the tuple independently.

**Default Intent for Arrays and Record 'this'**

The default intent for arrays and for a `this` argument of record type (see §14.2) is `ref` or `const ref`. It is `ref` if the formal argument is modified inside the function, otherwise it is `const ref`. Note that neither of these cause an array or record to be copied by default. The choice between `ref` and `const ref` is similar to and interacts with return intent overloads (see §13.7.3).

**Default Intent for 'owned' and 'shared'**

The default intent for owned and shared arguments is `const ref`. Arguments can use the `in` or `const in` intents to transfer or share ownership if those arguments apply to owned or shared types.

```chapl
Example (owned-any-intent.chpl).
proc defaultGeneric(arg) {
    writeln(arg.type: string);
}
class SomeClass { }
var own = new owned SomeClass();
defaultGeneric(own);
writeln(own != nil);
```

### 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.

```chapl
Example (varargs.chpl). The code
```
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 (§11.9.2) 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 written as

```chapel
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

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

defines a generic procedure that is equivalent to building a tuple. Therefore the expressions `tuple(1, 2)` and `(1,2)` are equivalent, as are the expressions `tuple(1)` and `(1,)`.

### 13.7 Return Intents

The `return-intent` specifies how the value is returned from a function, and in what contexts that function is allowed to be used. By default, or if the `return-intent` is `const`, the function returns a value that cannot be used as an lvalue.

#### 13.7.1 The Ref Return Intent

When using a `ref` return intent, the function call is an lvalue (specifically, a call expression for a procedure and an iterator variable for an iterator).

The `ref` return intent is specified by following the argument list with the `ref` keyword. The function must return or yield an lvalue.

**Example (ref-return-intent.chpl).** The following code defines a procedure that can be interpreted as a simple two-element array where the elements are actually module level variables:

```chapel
var x, y = 0;

proc A(i: int) ref {
  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

\[
\text{writeln}(A(0) + A(1));
\]

This code outputs the number 3.

13.7.2 The Const Ref Return Intent

The const ref return intent is also available. It is a restricted form of the ref return intent. Calls to functions marked with the const ref return intent are not lvalue expressions.

13.7.3 Return Intent Overloads

In some situations, it is useful to choose the function called based upon how the returned value is used. In particular, suppose that there are two functions that have the same formal arguments and differ only in their return intent. One might expect such a situation to result in an error indicating that it is ambiguous which function is called. However, the Chapel language includes a special rule for determining which function to call when the candidate functions are otherwise ambiguous except for their return intent. This rule enables data structures such as sparse arrays.

See 13.13.5 for a detailed description of how return intent overloads are chosen based upon calling context.

Example (ref-return-intent-pair.chpl).

Return intent overload 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. The following is an example:

```chapel
var x, y = 0;
proc doA(param setter, i: int) ref { 
  if i < 0 || i > 1 then 
    halt("array access out of bounds");
  if setter && i == 1 && x <= 0 then 
    halt("cannot assign value to A(1) if A(0) <= 0");
  if i == 0 then 
    return x;
  else 
    return y;
}
proc A(i: int) ref { 
  return doA(true, i);
}
proc A(i: int) { 
  return doA(false, i);
}
A(0) = 0;
A(1) = 1;
```
13.7.4 The Param Return Intent

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

```chapel
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 §11.9.2 and conditionals with a conditional expressions that is not a parameter.

13.7.5 The Type Return Intent

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 §11.9.2 and conditionals with a conditional expression that is not a parameter.

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

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

`myType` is a type procedure that takes a single argument `x` and returns `int(32)` 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 standard Types module.
13.8 The Return Statement

The return statement can only appear in a function. It causes control to exit that function, returning it to the point at which that function was called.

A procedure can return a value by executing a return statement that includes an expression. If it does, that expression’s value becomes the value of the invoking call expression.

A return statement in a procedure of a non-void return type (§13.9) must include an expression. A return statement in a procedure of a void return type or in an iterator must not include an expression. A return statement of a variable procedure must contain an lvalue expression.

The syntax of the return statement is given by

\[
\text{return} \quad \text{statement} \\
= \quad \text{return} \quad \text{expression}_{\text{opt}} ;
\]

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

\[
\text{proc} \quad \text{sum}(i1: \text{int}, i2: \text{int}, i3: \text{int}) \\
\text{return} \quad i1 + i2 + i3;
\]

13.9 Return Types

Every procedure has a return type. The return type is either specified explicitly via return-type in the procedure declaration, or is inferred implicitly.

13.9.1 Explicit Return Types

If a return type is specified and is not void, each return statement of the procedure must include an expression. For a non-ref return intent, an implicit conversion occurs from each return expression to the specified return type. For a ref return intent (§13.7.1), 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).

13.9.2 Implicit Return Types

If a return type is not specified, it is inferred from the return statements. It is illegal for a procedure to have a return statement with an expression and a return statement without an expression. For procedures without any return statements, or when none of the return statements include an expression, the return type is void.

Otherwise, the types of the expressions in all of the procedure’s return statements are considered. If a function has a ref return intent (§13.7.1), they all must be the same exact type, which becomes the inferred return type. Otherwise, there must exist exactly one type such that an implicit conversion is allowed between every other type and that type, and that type becomes the inferred return type. If the above requirements are not satisfied, it is an error.
13.10 Where Clauses

The list of function candidates can be constrained by where clauses. A where clause is specified in the definition of a function (§13.2). The expression in the where clause must be a boolean parameter expression that evaluates to either true or false. If it evaluates to false, the function is rejected and thus is not a possible candidate for function resolution.

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 the where clause on the second function evaluates to false.

13.11 Nested Functions

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

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

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

13.12 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 §13.13.

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:

<table>
<thead>
<tr>
<th>arity</th>
<th>operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary</td>
<td>+ - ! ~</td>
</tr>
<tr>
<td>binary</td>
<td>+ - * / % ** == &lt;= &gt;= &lt; &gt; &lt;&lt; &gt;&gt; &amp;</td>
</tr>
<tr>
<td></td>
<td>= += -= *= /= %== *== &gt;=</td>
</tr>
</tbody>
</table>

The arity and precedence of the operator must be maintained when it is overloaded. Operator resolution follows the same algorithm as function resolution.
13.13 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 §13.13.1. 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 §13.13.2. 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, determine the set of most specific functions. In most cases, there is one most specific function, but there can be several if they differ only in return intent. The set of most specific functions is the set of functions that are not more specific than each other but that are more specific than every other candidate function. The more specific relationship is defined in §13.13.3.
- From the set of most specific functions, the compiler determines a best function for each return intent as described in §13.13.4. If there is more than one best function for a given return intent, the compiler will issue an error stating that the call is ambiguous. Otherwise, it will choose which function to call based on the calling context as described in §13.13.5.

13.13.1 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 publicly declared 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 §24.2.

13.13.2 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 than 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 $T_F$.

13.13.3 Determining More Specific Functions

Given two functions $F_1$ and $F_2$, the more specific function is determined by the first of the following steps that applies:

- 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 at least one of the legal argument mappings to $F_1$ is weak preferred and none of the legal argument mappings to $F_2$ are weak preferred, then $F_1$ is more specific.
- If at least one of the legal argument mappings to $F_2$ is weak preferred and none of the legal argument mappings to $F_1$ are weak preferred, then $F_2$ is more specific.
- If at least one of the legal argument mappings to $F_1$ is weaker preferred and none of the legal argument mappings to $F_2$ are weaker preferred, then $F_1$ is more specific.
- If at least one of the legal argument mappings to $F_2$ is weaker preferred and none of the legal argument mappings to $F_1$ are weaker preferred, then $F_2$ is more specific.
- If at least one of the legal argument mappings to $F_1$ is weakest preferred and none of the legal argument mappings to $F_2$ are weakest preferred, then $F_1$ is more specific.
- If at least one of the legal argument mappings to $F_2$ is weakest preferred and none of the legal argument mappings to $F_1$ are weakest preferred, 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, \( F_1 \), of type \( T_{F_1} \) and an argument mapping, \( M_2 \), from the same actual argument to a formal argument, \( F_2 \), of type \( T_{F_2} \), the level of preference for one of these argument mappings is determined by the first of the following steps that applies:

- If \( T_{F_1} \) and \( T_{F_2} \) are the same type, \( F_1 \) is an instantiated parameter, and \( F_2 \) is not an instantiated parameter, \( M_1 \) is more specific.
- If \( T_{F_1} \) and \( T_{F_2} \) are the same type, \( F_2 \) is an instantiated parameter, and \( F_1 \) 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 \( T_{F_1} \) and \( T_{F_2} \) are the same type, \( F_1 \) is generic, and \( F_2 \) is not generic, \( M_1 \) is more specific.
- If \( T_{F_1} \) and \( T_{F_2} \) are the same type, \( F_2 \) is generic, and \( F_1 \) is not generic, \( M_2 \) is more specific.
- If \( F_1 \) is not generic over all types and \( F_2 \) is generic over all types, \( M_1 \) is more specific.
- If \( F_1 \) is generic over all types and \( F_2 \) is not generic over all types, \( M_2 \) is more specific.
- If \( F_1 \) and \( F_2 \) are both generic, and \( F_1 \) is partially concrete but \( F_2 \) is not, then \( M_1 \) is more specific.
- If \( F_1 \) and \( F_2 \) are both generic, and \( F_2 \) is partially concrete but \( F_1 \) is not, then \( M_2 \) is more specific.
- If \( F_1 \) is a \texttt{param} argument but \( F_2 \) is not, then \( M_1 \) is weak preferred.
- If \( F_2 \) is a \texttt{param} argument but \( F_1 \) is not, then \( M_2 \) is weak preferred.
- If \( A \) is not a \texttt{param} argument with a default size and \( F_2 \) requires a narrowing conversion but \( F_1 \) does not, then \( M_1 \) is weak preferred.
- If \( A \) is not a \texttt{param} argument with a default size and \( F_1 \) requires a narrowing conversion but \( F_2 \) does not, then \( M_2 \) is weak preferred.
- If \( T_A \) and \( T_{F_1} \) are the same type and \( T_A \) and \( T_{F_2} \) are not the same type, \( M_1 \) is more specific.
- If \( T_A \) and \( T_{F_1} \) are not the same type and \( T_A \) and \( T_{F_2} \) are the same type, \( M_2 \) is more specific.
- If \( A \) uses a scalar promotion type equal to \( T_{F_1} \) but different from \( T_{F_2} \), then \( M_1 \) will be preferred as follows:
  - if \( A \) is a \texttt{param} argument with a default size, then \( M_1 \) is weakest preferred
  - if \( A \) is a \texttt{param} argument with non-default size, then \( M_1 \) is weaker preferred
  - otherwise, \( M_1 \) is more specific
- If \( A \) uses a scalar promotion type equal to \( T_{F_2} \) but different from \( T_{F_1} \), then \( M_2 \) will be preferred as follows:
  - if \( A \) is a \texttt{param} argument with a default size, then \( M_2 \) is weakest preferred
  - if \( A \) is a \texttt{param} argument with non-default size, then \( M_2 \) is weaker preferred
  - otherwise, \( M_2 \) is more specific
- If \( T_A \) or its scalar promotion type prefers conversion to \( T_{F_1} \) over conversion to \( T_{F_2} \), then \( M_1 \) is preferred. If \( A \) is a \texttt{param} argument with a default size, then \( M_1 \) is weakest preferred. Otherwise, \( M_1 \) is weaker preferred.

Type conversion preferences are as follows:
- Prefer converting a numeric argument to a numeric argument of a different width but the same category over converting to another type. Categories are
  * bool
  * enum
  * int or uint
  * real
  * imag
  * complex
- Prefer an enum or bool cast to int over uint
- Prefer an enum or bool cast to a default-sized int or uint over another size of int or uint
- Prefer an enum, bool, int, or uint cast to a default-sized real over another size of real or complex
- Prefer an enum, bool, int, or uint cast to a default-sized complex over another size of complex
- Prefer real/imag cast to the complex with that component size (ie total width of twice the real/imag) over another size of complex

• If \( T_A \) or its scalar promotion type prefers conversion to \( T_{F2} \) over conversion to \( T_{F1} \), then \( M_2 \) is preferred. If \( A \) is a param argument with a default size, then \( M_2 \) is weakest preferred. Otherwise, \( M_2 \) is weaker preferred.
• 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.

13.13.4 Determining Best Functions

Given the set of most specific functions for a given return intent, only the following function(s) are selected as best functions:

• all functions, if none of them contain a where clause;
• only those functions that have a where clause, otherwise.

13.13.5 Choosing Return Intent Overloads Based on Calling Context

See also [13.7.3](#).

The compiler can choose between overloads differing in return intent when:
• there are zero or one best functions for each of ref, const ref, const, or the default (blank) return intent
• at least two of the above return intents have a best function.

In that case, the compiler is able to choose between ref return, const ref return, and value return functions based upon the context of the call. The compiler chooses between these return intent overloads as follows:

If present, a ref return version will be chosen when:

• the call appears on the left-hand side of a variable initialization or assignment statement
• the call is passed to another function as a formal argument with out, inout, or ref intent
• the call is captured into a ref variable
• the call is returned from a function with ref return intent

Otherwise, the const ref return or value return version will be chosen. If only one of these is in the set of most specific functions, it will be chosen. If both are present in the set, the choice will be made as follows:

The const ref version will be chosen when:

• the call is passed to another function as a formal argument with const ref intent
• the call is captured into a const ref variable
• the call is returned from a function with const ref return intent

Otherwise, the value version will be chosen.
14 Methods

A method is a procedure or iterator that is associated with an expression known as the receiver.

Methods are declared with the following syntax:

\[
\text{method-declaration-statement:}
\begin{align*}
\text{procedure-kind}_{\text{opt}} & \quad \text{proc} \quad \text{or} \quad \text{iter} \quad \text{this-intent}_{\text{opt}} \quad \text{type-binding}_{\text{opt}} \quad \text{function-name} \quad \text{argument-list}_{\text{opt}} \\
\text{return-intent}_{\text{opt}} & \quad \text{return-type}_{\text{opt}} \quad \text{where-clause}_{\text{opt}} \quad \text{function-body}
\end{align*}
\]

\[
\text{proc-or-iter:}
\begin{align*}
\text{proc} \\
\text{iter}
\end{align*}
\]

\[
\text{this-intent:}
\begin{align*}
\text{param} \\
\text{type} \\
\text{ref} \\
\text{const-ref} \\
\text{const}
\end{align*}
\]

\[
\text{type-binding:}
\begin{align*}
\text{identifier} \: . \\
\text{( expr )} \\
\end{align*}
\]

Methods defined within the lexical scope of a class, record, or union are referred to as primary methods. For such methods, the type-binding is omitted and is taken to be the innermost class, record, or union in which the method is defined.

Methods defined outside of such scopes are known as secondary methods and must have a type-binding (otherwise, they would simply be standalone functions rather than methods). Note that secondary methods can be defined not only for classes, records, and unions, but also for any other type (e.g., integers, reals, strings).

Secondary methods can be declared with a type expression instead of a type identifier. In particular, if the type-binding is a parenthesized expression, the compiler will evaluate that expression to find the receiver type for the method. In that case, the method applies only to receivers of that type. See also §24.5

Method calls are described in §14.1

The use of this-intent is described in §14.2

14.1 Method Calls

A method is invoked with a method call, which is similar to a non-method call expression, but it can include a receiver clause. The receiver clause syntactically identifies a single argument by putting it before the method name. That argument is the method receiver. When calling a method from another method, or from within a class or record declaration, the receiver clause can be omitted.
The receiver-clause (or its absence) specifies the method’s receiver §14.2.

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

```chapl
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 §14.2.

### 14.2 The Method Receiver and the this Argument

A method’s receiver is an implicit formal argument named this representing the expression on which the method is invoked. The receiver’s actual argument is specified by the receiver-clause of a method-call-expression as specified in §14.1.

Example (implicitThis.chpl). Let class C, method foo, and function bar be defined as

```chapl
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. Within primary method `C.foo()`, the (implicit) receiver formal has static type `C` (otherwise known as borrowed `C`) and is referred to as `this`.

Methods whose receivers are objects are called *instance methods*. Methods may also be defined to have *type* receivers—these are known as *type methods*.

Note that within a method for a class C, the type of this is generally *borrowed C*. Within a type method on a class C, this refers to the class type C with management and nilability matching the type of the receiver. Please see §17.1.7 for more details.

The optional *this-intent* is used to specify type methods, to constrain a receiver argument to be a param, or to specify how the receiver argument should be passed to the method.

When no *this-intent* is used, a default this intent applies. For methods on classes and other primitive types, the default this intent is the same as the default intent for that type. For record methods, the intent for the receiver formal
Methods

argument is \texttt{ref} or \texttt{const ref}, depending on whether the formal argument is modified inside of the method. Programmers wishing to be explicit about whether or not record methods modify the receiver can explicitly use the \texttt{ref} or \texttt{const ref} \texttt{this-intent}.

A method whose \texttt{this-intent} is \texttt{type} defines a \textit{type method}. It can only be called on the type itself rather than on an instance of the type. When \texttt{this-intent} is \texttt{param}, it specifies that the function can only be applied to \texttt{param} objects of the given type binding.

\textit{Example (paramTypeThisIntent.chpl)}. In the following code, the \texttt{isOdd} method is defined with a \texttt{this-intent} of \texttt{param}, permitting it to be called on \texttt{param}s only. The \texttt{size} method is defined with a \texttt{this-intent} of \texttt{type}, requiring it to be called on the \texttt{int} type itself, not on integer values.

\begin{alltt}
proc param int.isOdd() param { 
    return this & 0x1 == 0x1;
}

proc type int.size() param { 
    return 64;
}

param three = 3;
var seven = 7;

writeln(42.isOdd()); // prints false
writeln(three.isOdd()); // prints true
// writeln((42+three).isOdd()); // prints true
// writeln(seven.isOdd()); // illegal since ‘seven’ is not a param

writeln(int.size()); // prints 64
// writeln(42.size()); // illegal since ‘size()’ is a type method
\end{alltt}

Type methods can also be iterators.

\textit{Example (typeMethodIter.chpl)}. In the following code, the \texttt{C} class defines a type method iterator which can be invoked on the type itself:

\begin{alltt}
class C { 
    var x: int;
    var y: string;

    iter type myIter() { 
        yield 3;
        yield 5;
        yield 7;
        yield 11;
    }
}

for i in C.myIter() do 
    writeln(i);
\end{alltt}

When \texttt{this-intent} is \texttt{ref}, the receiver argument will be passed by reference, allowing modifications to \texttt{this}. If \texttt{this-intent} is \texttt{const ref}, the receiver argument is passed by reference but it cannot be modified inside the method. The \texttt{this-intent} can also describe an abstract intent as follows. If it is \texttt{const}, the receiver argument will be passed with \texttt{const} intent. If it is left out entirely, the receiver will be passed with a default intent. For records, that default intent is \texttt{ref} if \texttt{this} is modified within the function and \texttt{const ref} otherwise. For other types, the default \texttt{this-intent} matches the default argument intent described in \S13.5.2.
Example (refThisIntent.chpl). In the following code, the doubleMe function is defined with a this-intent of ref, allowing variables of type int to double themselves.

```chpl
proc ref int.doubleMe() { this *= 2; }
```

Given a variable \( x = 2 \), a call to \( x.doubleMe() \) will set \( x \) to 4.

### 14.3 The this Method

A procedure method declared with the name this allows the receiver to be “indexed” similarly to how an array is indexed. Indexing (as with \( A[1] \)) 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.

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

### 14.4 The these Method

An iterator method declared with the name these allows the receiver to be “iterated over” similarly to how a domain or array supports iteration. When a value supporting a these method is used as the the iterable-expression of a loop, the loop proceeds in a manner controlled by the these iterator.

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.

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

An iterator type method with the name these supports iteration over the class type itself.
Methods

Example (typeMethodIterThese.chpl). In the following code, the class C defines a type method iterator named these, supporting direct iteration over the type:

class C {
  var x: int;
  var y: string;

  iter type these() {
    yield 1;
    yield 2;
    yield 4;
    yield 8;
  }
}

for i in C do
  writeln(i);
15 Error Handling

The Chapel language supports throw, try, try!, catch, and throws which are described below. Chapel supports several error handling modes, including a mode suitable for prototype development and a less-permissive mode intended for production code.

Cray’s Chapel Implementation. Additional information about the Cray Chapel implementation of error handling and the strict error handling mode, which is not defined here, is available online in the technical note: https://chapel-lang.org/docs/technotes/errorHandling.html

15.1 Throwing Errors

Errors may be thrown from a function to its callee with a throw statement. For a function to throw an error, its signature must include a throws declaration. The declaration is put after the return type and before any where clauses.

Only owned instances of a type inheriting from Error can be thrown.

Example (throwing.chpl).

```chapel
proc canThrow(i: int): int throws {
  if i < 0 then
    throw new owned Error();
  return i + 1;
}
proc alwaysThrows():int throws {
  throw new owned Error();
  // never reached
  return 1;
}
```

15.2 Handling Errors

There are three ways to handle an error:

- Halt with try!.
- Handle the error with catch blocks.
- Propagate the error out of the current function with throws.
Error Handling

15.2.1 Halting on error with try!

If an error is thrown by a call within the lexical scope of a try! block or a try! expression prefix, the program halts.

Example (try-bang.chpl).

```chapel
proc haltsOnError():int {
  // the try! next to the throwing call
  // halts the program if an error occurs.
  return try! canThrow(0);
}

proc haltsOnErrorBlock() {
  try! {
    canThrow(1);
    canThrow(0);
  }
}
```

15.2.2 Handling an error with catch

When an error is raised by a call in a try or try! block, the rest of the block is abandoned and control flow is passed to its catch clause(s), if any.

Catch clauses

A try or try! block can have one or more catch clauses.

A catch clause can specify the variable that refers to the caught error within the catch block. If the variable is given a type, for example catch e:SomeError, it is a type filter. The corresponding catch clause matches the errors that are of the class SomeError or its subclass. If no type filter is present on a catch clause, or if no variable is present at all, then it is a catchall clause, which matches all errors.

The type filters are evaluated in the order that the catch clauses appear in the program. If a catch clause’s type filter matches, then its block is executed to the exclusion of the others. Hence there is no notion of best match, only a first match.

If the catch block itself throws an error, it is handled in the same manner as if that error were thrown by a statement adjacent to the try-catch blocks. Otherwise, after the execution of the catch block completes, the program execution proceeds to the next statement after the try-catch blocks.

Example (catching-errors.chpl).

```chapel
proc catchingErrors() throws {
  try {
    alwaysThrows(0);
  } catch {
    writeln("caught an error, unnamed catchall clause");
  }
  try |
```
Error Handling

```chapel
var x = alwaysThrows(-1);
writeln("never reached");
} catch e:FileNotFoundException {
    writeln("caught an error, FileNotFoundError type filter matched");
} catch e {
    writeln("caught an error in a named catchall clause");
}
```

**try! with catch**

If an error is thrown within a `try!` block and none of its `catch` clauses, if any, match that error, the program halts.

*Example (catching-errors-halt.chpl).*

```chapel
proc catchingErrorsHalt() {
    try! {
        var x = alwaysThrows(-1);
        writeln("never reached");
    } catch e:FileNotFoundException {
        writeln("caught a file not found error");
    }
    // errors other than FileNotFoundError cause a halt
}
```

**Nested try**

If an error is thrown within a `try` block and none of its `catch` clauses, if any, match that error, the error is directed to the enclosing `try` block, when present.

*Example (nested-try.chpl).*

```chapel
class DemoError : Error { }
proc nestedTry() {
    try {
        try {
            alwaysThrows(0);
        } catch e: DemoError {
            writeln("caught a DemoError");
        }
        writeln("never reached");
    } catch {
        writeln("caught an Error from inner try");
    }
}
```

15.2.3 Propagating an error with throws

A function marked `throws` can pass along an error thrown by a function called within it. This can be done in several ways.
After catch clauses

Propagation can occur when no matching catch clause is found for an error raised in a try block.

Example (catching-errors-propagate.chpl).

```chpl
proc catchingErrorsPropagate() throws {
try {
    var x = alwaysThrows(-1);
    writeln("never reached");
} catch e:FileNotFoundException {
    writeln("caught a file not found error");
}
// errors other than FileNotFoundError propagate
}
```

catch-less try

A logical extension of the above is the case where no catch blocks are attached to the try. Here the try keyword marks throwing calls to clarify control flow.

Example (propagates-error.chpl).

```chpl
proc propagatesError() throws {
    // control flow changes if an error was thrown;
    // could be indicated more clearly with try
    canThrow(0);
    try canThrow(0);
    try {
        canThrow(0);
    }
    var x = try canThrow(1);
    writeln(x);
    return try canThrow(0);
}
```

try expressions

try and try! are available as expressions to clarify control flow at expression granularity. The expression form may not be used with catch clauses.

Example (expression-try.chpl).

```chpl
proc expressionTry(): int throws {
    var x = try canThrow(1);
    writeln(x);
    return try canThrow(0);
}
```
15.2.4 Complete handling

For a function to handle errors from its calls without itself throwing, its try/catch must be complete. This may be accomplished in two ways:

- A catchall clause on try. This prevents try from propagating the error out of the function as described above.

  Example (warns-on-error.chpl).

  ```chpl
  proc warnsOnError(i: int): int {
  try {
    alwaysThrows(i);
  } catch e {
    writeln("Warning: caught a error ", e);
  } }
  ```

- try! instead of try. This will halt the program if no matching catch clause is found, instead of propagating.

  Example (halts-on-error.chpl).

  ```chpl
  class DemoError : Error { }
  proc haltsOnError(i: int): int {
  try! {
    canThrow(i);
  } catch e: DemoError {
    writeln("caught a DemoError");
  } }
  ```

15.3 Defer statement

When an error is thrown, it is sometimes necessary to clean up state and allocated memory. defer statements facilitate that by running when a scope is exited, regardless of how it is exited.

Example (defer.chpl).

```chpl
proc deferredDelete(i: int) {
  try {
    var huge = allocateLargeObject();
    defer {
      delete huge;
      writeln("huge has been deleted");
    }
    canThrow(i);
    processObject(huge);
  } catch {
    writeln("no memory leaks");
  }
}
``` 

It is not possible to throw errors out of a defer statement because the atomicity of all defer statements must be guaranteed, and the handling context would be unclear.

Errors also cannot be thrown by deinit() for similar reasons.
15.4 Methods

Errors can be thrown by methods, just as with any other function. An overriding method must throw if the overridden method throws, or not throw if the overridden method does not throw.

*Example (throwing-methods.chpl).*

```chapel
class ThrowingObject {
    proc f() throws {
        throw new owned Error();
    }
}

class SubThrowingObject : ThrowingObject {
    // must be marked throws even though it doesn’t throw
    proc f() throws {
        writeln("this version doesn’t throw");
    }
}
```

15.5 Multilocus

Errors can be thrown within `on` statements. In that event, the error will be propagated out of the `on` statement.

*Example (handle-from-on.chpl).*

```chapel
proc handleFromOn() {
    try {
        on Locales[0] {
            canThrow(1);
        }
    } catch {
        writeln("caught from Locale 0");
    }
}
```

15.6 Parallelism

15.6.1 TaskErrors

*TaskErrors* class helps coordinate errors among groups of tasks by collecting them for centralized handling. It can be iterated on and filtered for different kinds of errors. See also [https://chapel-lang.org/docs/builtins/ChapelError.html#ChapelError.TaskErrors](https://chapel-lang.org/docs/builtins/ChapelError.html#ChapelError.TaskErrors).

Nested `coforall` statements do not produce nested TaskErrors. Instead, the nested errors are flattened into the TaskErrors error thrown by the outer loop.
15.6.2 begin

Errors can be thrown within a begin statement. In that event, the error will be propagated to the sync statement that waits for that task.

*Example (handle-from-begin.chpl).*

```chapl
proc handleFromBegin() {  
  try! {  
    sync {  
      begin canThrow(0);  
      begin canThrow(1);  
    }  
    catch e: TaskErrors {  
      writeln("caught from Locale 0");  
    }  
  }  
}
```

15.6.3 coforall and cobegin

Errors can be thrown from coforall and cobegin statements and handled as TaskErrors. The nested coforall loops will emit a flattened TaskErrors error.

*Example (handle-from-coforall.chpl).*

```chapl
proc handleFromCoforall() {  
  try! {  
    writeln("before coforall block");  
    coforall i in 1..2 {  
      coforall j in 1..2 {  
        throw new owned DemoError();  
      }  
    }  
    writeln("after coforall block");  
    catch errors: TaskErrors { // not nested  
      // all of e will be of runtime type DemoError in this example  
      for e in errors {  
        writeln("Caught task error e ", e.message());  
      }  
    }  
  }  
}
```

*Example (handle-from-cobegin.chpl).*

```chapl
proc handleFromCobegin() {  
  try! {  
    writeln("before cobegin block");  
    cobegin {  
      throw new owned DemoError();  
      throw new owned DemoError();  
    }  
    writeln("after cobegin block");  
    catch errors: TaskErrors {  
      for e in errors {  
        writeln("Caught task error e ", e.message());  
      }  
    }  
  }  
}
15.6.4 forall

Errors can be thrown from forall loops, too. Although the forall may execute serially within a single task, it will always throw a TaskErrors error if error(s) are thrown in the loop body.

Example (handle-from-forall.chpl).

```chapel
proc handleFromForall() {
  try {
    writeln("before forall block");
    forall i in 1..2 {
      throw new owned DemoError();
    }
    writeln("after forall block");
  } catch errors: TaskErrors {
    for e in errors {
      writeln("Caught task error e ", e.message());
    }
  }
}
```

15.7 Creating New Error Types

Errors in Chapel are implemented as classes, with a base class Error defined in the standard modules. Error may be used directly, and new subclass hierarchies may be created from it. See also [https://chapel-lang.org/docs/builtins/ChapelError.html](https://chapel-lang.org/docs/builtins/ChapelError.html).

A hierarchy for system errors is included in the SysError module, accessed with a use statement. See also [https://chapel-lang.org/docs/modules/standard/SysError.html](https://chapel-lang.org/docs/modules/standard/SysError.html).

Example (defining-errors.chpl).

```chapel
use SysError;

class DemoError : Error { }
class DemoSysError : SystemError { }
```

15.8 Error Handling Modes

Certain error handling details depend on the error handling mode:

- Code in prototype modules (§12.2), including implicit modules (§12.3), is handled in the prototype mode.
- Otherwise, code is handled in the production mode.

Code that is legal in the production mode is always legal in the prototype mode.
15.8.1 Prototype Mode

In the prototype mode, it is not necessary to explicitly handle errors from a function that throws. If an error is thrown and the calling function throws, the error will be propagated out of the function. However, if an error is thrown and the calling function does not include a `throws` declaration, the program will halt.

In the following example, the code is in an implicit module. It is legal in the prototype mode:

```chapel
Example (fatal-mode.chpl).

canThrow(1); // handling can be omitted; halts if an error occurs

proc throwsErrorsOn() throws {
    // error propagates out of this function
    canThrow(-1);
}

proc doesNotThrowErrorsOn() {
    // causes a halt if called
    alwaysThrows();
}
```

The following module is explicitly marked as a prototype module, so the prototype mode applies here, too.

```chapel
Example (PrototypeModule.chpl).

prototype module PrototypeModule {

canThrow(1); // handling can be omitted; halts if an error occurs

proc throwsErrorsOn() throws {
    // error propagates out of this function
    alwaysThrows();
}

proc doesNotThrowErrorsOn() {
    // causes a halt if called
    alwaysThrows();
}

proc canThrow(i: int): int throws {
    if i < 0 then
        throw new owned Error();
    return i + 1;
}
```

15.8.2 Production Mode

In the production mode, it is necessary to handle errors if the calling function does not throw. If the calling function does throw, then the error will be propagated out, as with the prototype mode.
Example (ProductionModule.chpl),

```chapel
module ProductionModule {
  // This would cause a compilation error since the error is not handled:
  // canThrow();

  proc throwsErrorsOn() throws {
    // any error thrown by alwaysThrows will propagate out
    alwaysThrows();
  }

  // this function does not compile because the error is not handled
  // proc doesNotThrowErrorsOn() {
  //   alwaysThrows();
  // }
}
```
16 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.

16.1 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-expression , type-list )  
( type-expression , )  

type-list:  
type-expression  
type-expression , 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 (§16.3). 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.

**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 (homogeneous.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 \(3 \times (3 \times \text{int})\) specifies a 3-tuple of 3-tuples of integers rather than a 3-tuple of 1-tuples of 3-tuples of integers. The type \(3 \times 3 \times \text{int}\), on the other hand, specifies a 9-tuple of integers.

### 16.2 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. A trailing comma is allowed.

The syntax of a tuple expression is given by:

\[
\text{tuple-expression}:
\begin{align*}
\text{( tuple-component,)} \\
\text{( tuple-component, tuple-component-list) } \\
\text{( tuple-component, tuple-component-list,)} \\
\text{tuple-component:} \\
\text{expression} \\
\text{tuple-component-list:} \\
\text{ tuple-component} \\
\text{ tuple-component, tuple-component-list }
\end{align*}
\]

An underscore can be used to omit components when splitting a tuple (see [16.6.1]).

**Example (values.chpl).** The statement

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

defines three tuple 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.

Note that if a single expression is delimited by parentheses, the parentheses only impact precedence. Thus \((1)\) is equivalent to 1. To specify a 1-tuple, use the form with the trailing comma \((1,)\).

**Example (onetuple.chpl).** The statement

```chapel
var x: 1*int = (7,);
```

creates a 1-tuple of integers storing the value 7.

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.
16.3 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

```chpl
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

```chpl
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.

16.4 Iteration over Tuples

Only homogeneous tuples support iteration via standard for, forall and coforall loops. These loops iterate over all of the tuple’s elements. A loop of the form:

```chpl
[for|forall|coforall] e in t do
  ...e...
```

where t is a homogeneous tuple of size n, is semantically equivalent to:

```chpl
[for|forall|coforall] i in 1..n do
  ...t(i)...
```

The iterator variable for an tuple iteration is a either a const value or a reference to the tuple element type, following default intent semantics.

16.5 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.).
16.6 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.

16.6.1 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

```chapel
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

```chapel
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 underscore token can be used to omit storing some of the components. In this case, the full expression on the right-hand side of the assignment operator is evaluated, but the omitted values will not be assigned to anything.
Example (omit-component.chpl). The code

```chpl
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 \( 1 \).

### 16.6.2 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

```chpl
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 (onetuple-destruct.chpl). The code

```chpl
var (a) = (1, );
```

initialize the new variable \( a \) to 1.

When splitting a tuple into multiple variable declarations, the underscore token may be used to omit components of the tuple rather than declaring a new variable for them. In this case, no variables are defined for the omitted components.

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

```chpl
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 \( 1 \).
16.6.3 Splitting a Tuple into Multiple Indices of a Loop

When multiple indices in a loop are grouped using tuple notation, the tuple returned by the iterator (§23) 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

```chapel
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.

16.6.4 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

```chapel
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.
Tuples

Example (implicit-where.chpl). The function $f$ defined as

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

is equivalent to the function $g$ defined as

```chpl
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

```chpl
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.

### 16.6.5 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

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

the following statement

```chpl
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:

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

16.7.1 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.

16.7.2 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. These operators are also defined for homogeneous tuples and scalar values of matching type.

The size of the result tuple is the same as the argument tuple(s). 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

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

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

16.7.3 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.

### 16.8 Predefined Functions and Methods on Tuples

**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.

**proc Tuple.size param**

Returns the size of the tuple.
17 Classes

Classes are data structures with associated state and functions. A variable of class type either refers to a class instance, or contains a special \texttt{nil} value. Note that \texttt{object} is another name for a class instance. Storage for a class instance is not necessarily tied to the scope of the variable(s) referring to that class instance. It is possible for multiple variables to refer to the same class instance.

The \texttt{new-expression} can be used to create an instance of a class (§17.3). Depending on the memory management strategy, a class instance is either deleted automatically or can be deleted using the \texttt{delete-statement} (§17.8).

A class declaration (§17.1) generates a class type (§17.1.2). 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. A class can also be generic if it inherits from a generic class. Generic classes and fields are discussed in §24.3.

17.1 Class Declarations

A class is defined with the following syntax:

\begin{verbatim}
class-declaration-statement:
    class identifier class-inherits \{ class-statement-list \}

class-inherits:
    : basic-class-type

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

class-statement:
    variable-declaration-statement
    method-declaration-statement
    type-declaration-statement
    empty-statement
\end{verbatim}

A \texttt{class-declaration-statement} defines a new class type symbol specified by the identifier. It inherits from the class specified in the \texttt{class-inherits} clause, when provided (§17.2).

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 §24.3.

Future. Privacy controls for classes and records are currently not specified, as discussion is needed regarding its impact on inheritance, for instance.
17.1.1 Class Lifetime and Borrows

The lifetime of a class instance is the time between its creation and its deletion. It is legal to access the class fields or methods only during its lifetime.

Each allocation of a class instance specifies a memory management strategy. Four memory management strategies are available: owned, shared, borrowed, and unmanaged.

Owned and shared class instances always have their lifetime managed by the compiler. In other words, the compiler automatically calls delete on these instances to reclaim their memory. For these instances, = and copy initialization can result in the transfer or sharing of ownership. See https://chapel-lang.org/docs/builtins/OwnedObject.html and https://chapel-lang.org/docs/builtins/SharedObject.html

When borrowed is used as a memory management strategy in a new-expression, it also creates an instance that has its lifetime managed by the compiler (§17.3).

Class instances that are unmanaged have their lifetime managed explicitly and delete must be used to reclaim their memory.

No matter the memory management strategy used, class types support borrowing. A borrowed class instance refers to the same class instance as another variable but has no impact on the lifetime of that instance. The process of getting such a reference to an instance is called borrowing.

There are several ways to borrow an instance. To borrow explicitly the instance managed by another variable, call the .borrow() method. Additionally, coercions are available that are equivalent to calling the .borrow() method. For example:

```
Example (borrowing.chpl).

class C { }
proc test() {
  var own = new owned C();  // 'own' manages the memory of the instance
  var b = own.borrow();    // 'b' refers to the same instance but has no
                           // impact on the lifetime.
  var bc: borrowed C = own; // 'bc' stores the result of own.borrow()
                           // due to coercion from owned C to C
  var c: C = own;           // same as above
  // since 'C' is equivalent to 'borrowed C'
  // Note that these coercions can also apply
  // in the context of procedure calls.
  // the instance referred to by 'own' is
  // deleted here, at the end of the containing
  // block.
  .}
```
The `.borrow()` method is available on all class types (including `unmanaged` and `borrowed`) in order to support generic programming. For nilable class types, it returns the borrowed nilable class type.

Errors due to accessing an instance after the end of its lifetime are particularly difficult to debug. For this reason, the compiler includes a component called the lifetime checker. It identifies some cases where a borrowing variable can be accessed beyond the lifetime of an instance it refers to.

*Future.* The details of lifetime checking are not yet finalized or specified. Additional syntax to specify the lifetimes of function returns will probably be needed.

### 17.1.2 Class Types

A class type is formed by the combination of a basic class type and a memory management strategy.

```
class–type:
  basic–class–type
  owned basic–class–type
  shared basic–class–type
  borrowed basic–class–type
  unmanaged basic–class–type
```

A basic 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 24.3.6.

```
  basic–class–type:
    identifier
    identifier( named–expression–list )
```

A basic class type, including a generic class type that is not fully specified, may appear in the inheritance lists of other class declarations.

If no memory management strategy is indicated, the class will be considered to have generic management.

Variables of class type cannot store `nil` unless the class type is nilable (§17.1.3).

The memory management strategies have the following meaning:

- **owned** the instance will be deleted automatically when the `owned` variable goes out of scope, but only one `owned` variable can refer to the instance at a time. See [https://chapel-lang.org/docs/builtins/OwnedObject.html](https://chapel-lang.org/docs/builtins/OwnedObject.html)
- **shared** will be deleted when all of the `shared` variables referring to the instance go out of scope. See [https://chapel-lang.org/docs/builtins/SharedObject.html](https://chapel-lang.org/docs/builtins/SharedObject.html)
- **borrowed** refers to a class instance that has a lifetime managed by another variable.
- **unmanaged** the instance must have `delete` called on it explicitly to reclaim its memory.
It is an error to apply more than one memory management strategy to a class type. However, in some cases, generic code needs to compute a variant of the class type using a different memory management strategy. Casts from the class type to a different memory management strategy are available for this purpose (see §9.2.4).

Example (duplicate-management.chpl).

```chapel
class C { }
var x: borrowed unmanaged C;
```

Example (changing-management.chpl).

```chapel
class C { }
type borrowedC = borrowed C;
type ownedC = (borrowedC:owned);
```

### 17.1.3 Nilable Class Types

Variables of class type cannot store `nil` unless the class type is nilable. To declare a nilable class type, use the `?` operator to create a nilable type. For example, if `C` is a class type, then `C?` indicates the nilable class type with generic management. The `?` operator can be combined with memory management specifiers as well. For example, `borrowed C?` indicates a nilable class using the borrowed memory management strategy. Note that the `?` operator applies only to types.

The postfix `!` operator applies to a type or a value. When applied to a `borrowed` or `unmanaged` type, it returns the non-nilable version of that type. When applied to an `owned` or `shared` type, it returns the non-nilable borrowed type. When applied to a value, it asserts that the value is not `nil` and returns that value as a non-nilable type. If the value was in fact `nil`, it halts. It returns the borrowed type for `owned` or `shared`.

An alternative to `!` is to use a cast to a non-nilable type. Such a cast will throw `NilClassError` if the value was in fact `nil`. See §9.2.4.

Non-nilable class types are implicitly convertible to nilable class types. See §9.1.3.

Class methods generally expect a receiver of type `borrowed C` (see §17.1.7). Since such a class method call might involve dynamic dispatch, it is a program error to call a class method on a class receiver storing `nil`. The language helps to identify this error in two different ways, depending on whether or not the module is a prototype module (§12.2).

For modules that are not prototype modules, the compiler will not resolve calls to class methods if the receiver has nilable type. If the programmer knows that the receiver cannot store `nil` at that moment, they can use `!` to assert that the receiver is not `nil` and to convert it to the non-nilable borrowed type. For example:

Example (nilable-classes-bang.chpl).

```chapel
class C { }
proc method() { }
}
var c: owned C? = new C();
// The following call is allowed only in prototype modules,
// in which case it is equivalent to c!.method()
The `borrow()` method is an exception. Suppose it is invoked on an expression of a class type `C`. It will return `borrowed C` for any non-nilable `C` type (e.g. `owned C`). It will return `borrowed C?` for any nilable `C` type (e.g. `C?`). Within a prototype module, the compiler will implicitly convert a nilable method receiver to the non-nilable type by adding a `!` call.

### 17.1.4 Class Values

A class value is either a reference to an instance of a class or `nil` (§17.1.5). Class instances can be created using a new expression (§17.3).

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 non-nilable class type.

The default value of a concrete nilable class type is `nil`. Generic class types and non-nilable class types do not have a default value.

**Example (declaration.chpl).**

```chapl
class C { }
var c : owned C?; // c has class type owned C?, meaning
                   // the instance can be nil and is deleted automatically
                   // when it is not.
c = new C();       // Now c refers to an initialized instance of type C.
var c2 = c.borrow(); // The type of c2 is borrowed C?.
                   // c2 refers to the same object as c.
class D : C {}     // Class D is derived from C.
c = new D();       // Now c refers to an object of type D.
                   // Since c is owned, the previous is deleted.
                   // the C and D instances allocated above will be reclaimed
                   // at the end of this block.
```

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 `borrowed C?`, determined implicitly from the the initialization expression. Finally, an object of type `owned D` is created and assigned to `c`.

### 17.1.5 The `nil` Value

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

`nil` can be assigned to a variable of any nilable class type. There is a restriction for using `nil` as the default value or the actual argument of a function formal, or as the initializer for a variable or a field. Such a use is disallowed when the declared type of the formal/variable/field is non-nilable or generic, including generic memory management.
nil-expression:
   nil

17.1.6 Class Fields

A variable declaration within a class declaration defines a field within that class. Each class instance consists of one variable per each var or const field in the class.

Example (defineActor.chpl). The code

```chapel
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 §17.5.

Future. ref fields, which are fields corresponding to variable declarations with ref or const ref keywords, are an area of future work.

17.1.7 Class Methods

Methods on classes are referred to as to as class methods. See the methods section §14 for more information about methods.

Within a class method, the type of this is generally the non-nilable borrowed variant of the class type. It is different for type methods (see below) and it might be a different type if the class method is declared as a secondary method with a type expression (see §14).

For example:

Example (class-method-this-type.chpl).

```chapel
class C {
  proc primaryMethod() {
    assert(this.type == borrowed C);
  }
}
proc C.secondaryMethod() {
  assert(this.type == borrowed C);
}
proc (owned C?).secondaryMethodWithTypeExpression() {
  assert(this.type == owned C?);
}

var x:owned C? = new owned C();
x!.primaryMethod(); // within the method, this: borrowed C
x!.secondaryMethod(); // within the method, this: borrowed C
x.secondaryMethodWithTypeExpression(); // within the method, this: owned C?
```
For type methods on a class, `this` will accept any management or nilability variant of the class type and it will refer to that type in the body of the method. For example:

**Example (class-type-method-this.chpl).**
```chapel
class C {
proc type typeMethod() {
   writeln(this:string); // print out the type of 'this'
}
}
(C).typeMethod(); // prints 'C'
(owned C).typeMethod(); // prints 'owned C'
(borrowed C?).typeMethod(); // prints 'borrowed C?'
```

When a type method is defined only in a parent class, the type will be the corresponding variant of the parent class. For example:

**Example (class-type-method-inherit.chpl).**
```chapel
class Parent { }
class Child : Parent { }
proc type Parent.typeMethod() {
   writeln(this:string); // print out the type 'this'
}

Child.typeMethod(); // prints 'Parent'
(borrowed Child?).typeMethod(); // prints 'borrowed Parent?'
```

### 17.1.8 Nested Classes

A class defined within another class or record is a nested class. A nested class can be referenced only within its immediately enclosing class or record.

### 17.2 Inheritance

A class inherits, or derives, from the class specified in the class declaration’s `class-inherit` clause when such clause is present. Otherwise the class inherits from the predefined `object` class ([§17.2.1](#object-class)). In either case, there is exactly one parent class. There can be many classes that inherit from a particular parent class.

It is possible for a class to inherit from a generic class. Suppose for example that a class `C` inherits from class `ParentC`. In this situation, `C` will have type constructor arguments based upon generic fields in the `ParentC` as described in [§24.3.6](#object-class). Furthermore, a fully specified `C` will be a subclass of a corresponding fully specified `ParentC`.

#### 17.2.1 The `object` Class

All classes are derived from the `object` class, either directly or indirectly. If no class name appears in `class-inherit` clause, the class derives implicitly from `object`. Otherwise, a class derives from `object` indirectly through the class it inherits. A variable of type `object` can hold a reference to an object of any class type.
17.2.2 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 a getter method is overridden in the derived class, as discussed in §17.2.4.

17.2.3 Shadowing Base Class Fields

A field in the derived class can be declared with the same name as a field in the base class. Such a field shadows the field in the base class in that it is always referenced when it is accessed in the context of the derived class.

*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.

17.2.4 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 methods are considered for dynamic dispatch. In particular, dynamic dispatch will be used when the method receiver has a static type of the base class but refers to an instance of a derived class type.

In order to have identical signatures, two methods must have the same the names, intents, types, and order of formal arguments. The return type of the overriding method must either be the same as the return type of the base class’s method or be a subclass of the base class method’s return type.

Methods that override a base class method must be marked with the `override` keyword in the `procedure-kind`. Additionally, methods marked with `override` but for which there is no parent class method with an identical signature will result in a compiler error.

*Rationale.* This feature is designed to help avoid cases where class authors accidentally override a method without knowing it; or fail to override a method that they intended to due to not meeting the identical signature condition.

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.
17.3 Class New

To create an instance of a class, use a `new` expression. For example:

```
Example (class-new.chpl).

class C {
  var x: int;
}
var instance = new C(1);
```

The new expression can be defined by the following syntax:

```
new-expression:
  new type-expression ( argument-list )
```

An initializer for a given class is called by placing the `new` operator in front of a type expression. Any initializer arguments follow the class name in a parenthesized list.

Syntactically, the `type-expression` includes `owned`, `shared`, `borrowed`, and `unmanaged`. However these have important consequences for class new expressions. In particular, suppose `C` is a `type-expression` that results in a class type. Then:

- `new C()` is the same as `new owned C()`
- `new owned C()` allocates and initializes an instance that will be deleted at the end of the current block unless it is transferred to another `owned` variable. It results in something of type `owned C`.
- `new shared C()` allocates and initializes the instance that will be deleted when the last `shared` variable referring to it goes out of scope. Results in something of type `shared C`.
- `new borrowed C()` allocates and initializes an instance that will be automatically deleted at the end of the current block. This process is managed by an `owned` temporary. Unlike `new owned C()`, this results in a value of type `borrowed C` and ownership of the instance cannot be transferred out of the block. In other words, `new borrowed C()` is equivalent to `(new owned C()).borrow()`
- `new unmanaged C()` allocates and initializes an instance that must have `delete` called on it explicitly to avoid a memory leak. It results in something of type `unmanaged C`.

See also §17.1.1 and §17.1.2

17.4 Class Initializers

A `new` expression allocates memory for the desired class and invokes an `initializer` method on the uninitialized memory, passing any arguments following the class name. An initializer is implemented by a method named `init` and is responsible for initializing the fields of the class.

Any initializers declared in a program are `user-defined` initializers. If the program declares no initializers for a class, the compiler must generate an initializer for that class based on the types and initialization expressions of fields defined by that class.
17.4.1 User-Defined Initializers

A user-defined initializer is an initializer method explicitly declared in the program. An initializer declaration has the same syntax as a method declaration, with the restrictions that the name of the method must be `init` and there must not be a return type specifier. When an initializer is called, the usual function resolution mechanism (§13.13) is applied with the exception that an initializer may not be virtually dispatched.

A user-defined initializer is responsible for initializing all fields. An initializer may omit initialization of fields, but all fields that are initialized must be initialized in declaration order.

Initializers for generic classes (§24.3) handle generic fields without default values differently and may need to satisfy additional requirements. See Section 24.3.9 for details.

*Example (simpleInitializers.chpl)*. The following example shows a class with two initializers:

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

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

  proc init(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 initializer lets the user specify the initial coordinates and the second initializer lets the user specify the initial message when creating a MessagePoint.

Field Initialization Versus Assignment

Within the body of an initializer, the first use of a field as the left-hand side of an assignment statement will be considered initialization. Subsequent uses of the assignment operator on the field will invoke regular assignment as defined by the language.

*Example (fieldInitAssignment.chpl)*. The following example documents the difference between field initialization and field assignment.

```chapel
class PointDoubleX {
  var x, y : real;

  proc init(x: real, y: real) {
    this.x = x;  // initialization
    writeln("x = ", this.x);  // use of initialized field
    this.x = this.x * 2;  // assignment, use of initialized field
  }
} // class PointDoubleX
```
The first statement in the initializer initializes field $x$ to the value of the formal $x$. The second statement simply reads the value of the initialized field. The third statement reads the value of the field, doubles it, and assigns the result to the field $x$.

If a field is used before it is initialized, an compile-time error will be issued.

**Example (usedBeforeInitialized.chpl)**. In the following code:

```chapel
class Point {
    var x, y : real;
    proc init(x: real, y: real) {
        writeln(this.x); // Error: use of uninitialized field!
        this.x = x;
        this.y = y;
        writeln(this.y);
    }
}
var p = new Point(1.0, 2.0);
```

The first statement in the initializer reads the value of uninitialized field $x$, so the compiler will issue an error:

```
usedBeforeInitialized.chpl:4: In initializer:
usedBeforeInitialized.chpl:5: error: field "x" used before it is initialized
```

**Omitting Field Initializations**

In order to support productive and elegant initializers, the language allows field initializations to be omitted if the field has a type or if the field has an initialization expression. The compiler will insert initialization statements for such fields based on their types and default values.

**Example (fieldInitOmitted.chpl)**. In the following code:

```chapel
class LabeledPoint {
    var x : real;
    var y : real;
    var msg : string = 'Unlabeled';
    proc init(x: real, y: real) {
        this.x = x;
        this.y = y;
        // compiler inserts "this.msg = 'Unlabeled'";
    }
    proc init(msg : string) {
        // compiler inserts "this.x = 0.0;"
        // compiler inserts "this.y = 0.0;"
        this.msg = msg;
    }
}```
var A = new LabeledPoint(2.0, 3.0);
var B = new LabeledPoint("Origin");

The first initializer initializes the values of fields \( x \) and \( y \), and the compiler inserts initialization for the \( \text{msg} \) field by using its default value. The second initializer initializes the \( \text{msg} \) field, and the compiler inserts initialization for fields \( x \) and \( y \) based on the type of those fields (§8.1.1).

In order to reduce ambiguity and to ensure a well-defined order for side-effects, the language requires that all fields be initialized in field declaration order. This applies regardless of whether field initializations are omitted from the initializer body. If fields are initialized out of order, a compile-time error will be issued.

**Example (fieldsOutOfOrder.chpl).** In the following code:

```chapel
class Point3D {
  var x = 1.0;
  var y = 1.0;
  var z = 1.0;

  proc init(x: real) {
    this.x = x;
    // compiler inserts "this.y = 1.0;"
    this.z = y * 2.0;
  }

  proc init(x: real, y: real, z: real) {
    this.x = x;
    this.z = z;
    this.y = y; // Error!
  }
}

var A = new Point3D(1.0);
var B = new Point3D(1.0, 2.0, 3.0);
```

The first initializer leverages the well-defined order of omitted field initialization to use the default value of field \( y \) in order to explicitly initialize field \( z \).

The second initializer initializes field \( z \) before field \( y \), causing a compile-time error to be issued.

**Rationale.** Without this rule the compiler could insert default initialization for field \( y \) before \( z \) is explicitly initialized. The following statement would then be assignment to field \( y \), despite appearing to be initialization. This subtle difference may be confusing and surprising, and is avoided by requiring fields to be initialized in field declaration order.

**Limitations on Instance Usage in Initializers**

As the initializer makes progress, the class instance is incrementally initialized. In order to prevent usage of uninitialized memory, there are restrictions on usage of the class instance before it is fully initialized:

- Methods may not be invoked on partially-initialized instances
• `this` may not be passed to functions while partially-initialized

These rules allow all methods and functions to assume that class instances have been initialized, provided their value is not `nil`.

**Rationale.** The compiler could conceivably attempt to analyze methods and functions to determine which fields are used, and selectively allow method calls on partially-initialized class instances. Instead, it is simpler for the language to forbid method calls on partially-initialized instances.

Methods may be called and `this` may be passed to functions only after the built-in `complete` method is invoked. This method may not be overridden. If any fields have not been initialized by the time the `complete` method is invoked, they will be considered omitted and the compiler will insert initialization statements as described earlier. If the user does not invoke the `complete` method explicitly, the compiler will insert a call to `complete` at the end of the initializer.

**Rationale.** Due to support for omitted field initialization, there is potential for confusion regarding the overall status of initialization. This confusion is addressed in the design by requiring `complete` to explicitly mark the transition between partially and fully initialized instances.

**Implementors’ note.** Even if the user explicitly initializes every field, the `complete` method is still required to invoke other methods.

**Example (thisDotComplete.chpl).** In the following code:

```chapel
class LabeledPoint {
  var x, y : real;
  var max = 10.0;
  var msg : string = 'Unlabeled';

  proc init(x: real, y: real) {
    this.x = x;
    this.y = y;
    // compiler inserts initialization for 'max' and 'msg'
    this.complete(); // 'this' is now considered to be fully initialized
    this.verify();
    writeln(this);
  }

  proc init(msg : string) {
    // compiler inserts initialization for fields 'x', 'y', and 'max'
    this.msg = msg;
    // Illegal: this.verify();
    // Implicit 'this.complete();'
  }

  proc verify() {
    if x > max || y > max then
      halt("LabeledPoint out of bounds!");
  }
}

var A = new LabeledPoint(1.0, 2.0);
var B = new LabeledPoint("Origin");
```
The first initializer leverages the `complete` method to initialize the remaining fields and to allow for the usage of the `verify` method. Calling the `verify` method or passing `this` to `writeln` before the `complete` method is called would result in a compile-time error.

The second initializer exists to emphasize the rule that even though all fields are initialized after the initialization of the `msg` field, the compiler does not consider the type initialized until the `complete` method is called. If the second initializer tried to invoke the `verify` method, a compile-time error would be issued.

**Invoking Other Initializers**

In order to allow for code-reuse, an initializer may invoke another initializer implemented for the same type. Because the invoked initializer must operate on completely uninitialized memory, a compile-time error will be issued for field initialization before a call to `init`. Because each initializer either explicitly or implicitly invokes the `complete` method, all fields and methods may be used after such a call to `init`.

*Example (thisDotInit.chpl)*. In the following code:

```chapel
class Point3D {
    var x, y, z : real;

    proc init(x: real, y: real, z: real) {
        this.x = x;
        this.y = y;
        this.z = z;
        // implicit 'this.complete();'
    }

    proc init(u: real) {
        this.init(u, u, u);
        writeln(this);
    }
}

var A = new Point3D(1.0);
```

The second initializer leverages the first initializer to initialize all fields with the same value. After the `init` call the type is fully initialized, the `complete` method has been invoked, and so `this` can be passed to the `writeln` function.

**Initializing Fields in Conditional Statements**

Fields may be initialized inside of conditional statements, with the restriction that the same set of fields must be initialized in every branch. If the user omits any field initializations, the compiler will insert field initializations up to and including the field furthest in field declaration order between the conditional branches. If the else branch of a conditional statement is omitted, the compiler will generate an empty else branch and insert field initialization statements as needed.

*Example (initFieldConditional.chpl)*. In the following code:
class Point {
  var x, y : real;
  var r, theta : real;

  proc init(polar : bool, val : real) {
    if polar {
      // compiler inserts initialization for fields 'x' and 'y'
      this.x = val;
    } else {
      this.x = val;
      this.y = val;
      // compiler inserts initialization for field 'r'
    }
    // compiler inserts initialization for field 'theta'
  }
}

var A = new Point(true, 5.0);
var B = new Point(false, 1.0);

The compiler identifies field r as the latest field in both branches, and inserts omitted field initialization statements as needed to ensure that fields x, y, and r are all initialized by the end of the conditional.

Conditionals may also contain calls to parent initializers (§17.4.4) and other initializers defined for the current type, provided that the initialization state is the same at the end of the conditional statement.

Example (thisDotInitConditional.chpl). In the following code:

class Parent {
  var x, y : real;
}

class Child : Parent {
  var z : real;

  proc init(cond : bool, val : real) {
    if cond {
      super.init(val, val);
      this.z = val;
      this.complete();
    } else {
      this.init(val, val, val);
    }
  }

  proc init(x: real, y: real, z: real) {
    super.init(x, y);
    this.z = z;
  }
}

var c = new Child(true, 5.0);

The first initializer must invoke the complete method at the end of the if-branch in order to match the state at the end of the else-branch.

Miscellaneous Field Initialization Rules

Fields may not be initialized within loop statements or parallel statements.
17.4.2 The Compiler-Generated Initializer

A compiler-generated initializer for a class is created automatically if there are no initializers for that class in the program. The compiler-generated initializer has one argument for every field in the class, each of which has a default value equal to the field’s default value (if present) or the default value of the field’s type (if not). The order and names of arguments matches the order and names of field declarations within the class.

Generic fields are discussed in Section §24.3.8.

The compiler-generated initializer will initialize each field to the value of the corresponding actual argument.

**Example (defaultInitializer.chpl).** Given the class

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

there are no user-defined initializers for C, so new expressions will invoke C’s compiler-generated initializer. The x argument of the compiler-generated initializer 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 initializer as follows:

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

17.4.3 The postinit Method

The compiler-generated initializer is powerful and flexible, but cannot satisfy all initialization patterns desired by users. One way for users to leverage the compiler-generated initializer while adding their own functionality is to implement a method named postinit. The postinit method may also be implemented for types with user-defined initializers.

The compiler will insert a call to the postinit method after the initializer invoked by the new expression finishes, if the method exists. The postinit method accepts zero arguments and may not return anything. Otherwise, this method behaves like any other method.

**Example (postinit.chpl).** In the following code:

```chapel
class Point3D {
  var x, y : real;
  var max = 10.0;

  proc postinit() {
    verify();
  }

  proc verify() {
```
Classes

writeln("(", x, ", ", y, ")");
if x > max || y > max then
  writeln(" Point out of bounds!");

} }

var A = new Point3D();
var B = new Point3D(1.0, 2.0);
var C = new Point3D(y=5.0);
var D = new Point3D(50.0, 50.0);

Each of the new expressions invokes the compiler-generated initializer, then invokes the verify method via the postinit method:

(0.0, 0.0)
(1.0, 2.0)
(0.0, 5.0)
(50.0, 50.0)
Point out of bounds!

For classes that inherit, the user may invoke the parent’s postinit method or let the compiler insert a call automatically (§17.4.4).

17.4.4 Initializing Inherited Classes

User-defined initializers also allow for control over initialization of parent classes. All the fields of the parent type must be initialized before any fields of the child type, otherwise a compile-time error is issued. This allows for parent fields to be used in the definition of child fields. An initializer may invoke a parent’s initializer using the super keyword.

If the user does not explicitly call the parent’s initializer, the compiler will insert a call to the parent initializer with zero arguments at the start of the initializer.

Example (simpleSuperInit.chpl). In the following code:

class A {
  var a, b : real;

  proc init() {
    this.init(1.0);
  }

  proc init(val : real) {
    this.a = val;
    this.b = val * 2;
  }
}

class B : A {
  var x, y : real;

  proc init(val: real, x: real, y: real) {
    super.init(val);
    this.x = x;
    this.y = y;
  }
}
proc init() {
   // implicit super.init();
   this.x = a*2;
   this.y = b*2;
}

var b1 = new B(4.0, 1.0, 2.0);
var b2 = new B();

The first initializer explicitly calls an initializer for class A. Once the parent’s initializer is complete, fields of class B may be initialized.
The second initializer implicitly invokes the parent’s initializer with zero arguments, and then uses the parent’s fields to initialize its own fields.

As stated earlier, the compiler will insert a zero-argument call to the parent’s initializer if the user has not explicitly written one themselves. The exception to this rule is if the initializer body invokes another initializer on the current type (§17.4.1). This other initializer will either contain an implicit or explicit call to the parent initializer, and so the calling initializer should not attempt to initialize the parent itself. This also means that parent fields may not be accessed before explicit calls to init.

Example (superInitThisInit.chpl). In the following code:

class Parent {
   var x, y: real;
}

class Child : Parent {
   var z : real;
   
   proc init(x: real, y: real, z: real) {
      super.init(x, y);
      this.z = z;
   }
   
   proc init(z: real) {
      this.init(0.0, 0.0, z);
   }
}

var c = new Child(5.0);

The second initializer does not contain an implicit call to the parent’s initializer because it explicitly invokes another initializer.

Calling Methods on Parent Classes

Once super_init() returns, the dynamic type of this is the parent’s type until the complete method (§17.4.1) is invoked (except when the child’s fields are initialized and used). As a result, the parent’s methods may be called and this may be passed to functions as though it were of the parent type.

Rationale. After super_init() returns the instance is in some partially-initialized, but valid, state. Allowing this to be treated as the parent allows for additional functionality and flexibility for users.
Classes

Example (dynamicThisInit.chpl). In the following code:

```chapel
class Parent {
  var x, y : real;

  proc foo() {
    writeln("Parent.foo");
  }
}

class Child : Parent {
  var z : real;

  proc init(x: real, y: real, z: real) {
    super.init(x, y); // parent's compiler-generated initializer
    foo(); // Parent.foo
    this.z = z;
    this.complete();
    foo(); // Child.foo
  }

  override proc foo() {
    writeln("Child.foo");
  }
}

var c = new Child(1.0, 2.0, 3.0);
```

Once the parent’s initializer is finished, the parent method `foo` may be called. After the `complete` method is invoked, a call to `foo` resolves to the child’s overridden (§17.2.4) implementation:

```
Parent.foo
Child.foo
{x = 1.0, y = 2.0, z = 3.0}
```

The Compiler Generated Initializer for Inheriting Classes

The compiler-generated initializer for inheriting classes will have arguments with default values and names based on the field declarations in the parent class. Formals for the parent type will be listed before formals for the child type.

Example (compilerGeneratedInheritanceInit.chpl). In the following code:

```chapel
class Parent {
  var x, y: real;
}

class Child : Parent {
  var z: real;
}

var A = new Child();
var B = new Child(1.0, 2.0, 3.0); // x=1.0, y=2.0, z=3.0
var C = new Child(y=10.0);
```

Any new expressions using the `Child` type can invoke an initializer with three formals named `x`, `y`, and `z` that all have default values based on their types.
The postinit Method for Inheriting Classes

The `postinit` method on inheriting classes allows users to invoke the parent’s `postinit` method using the `super` keyword. If the user does not explicitly invoke the parent’s `postinit`, the compiler will insert the call at the top of the user’s `postinit` method. If the parent type has a `postinit` method but the inheriting class does not, the compiler will generate a `postinit` method that simply invokes the parent’s `postinit` method.

**Example (inheritancePostinit.chpl)**. In the following code:

```chapel
class Parent {
  var a, b : real;
  proc postinit() {
    writeln("Parent.postinit: ", a, ", ", b);
  }
}

class Child : Parent {
  var x, y : real;
  proc postinit() { // compiler inserts "super.postinit();"
    writeln("Child.postinit: ", x, ", ", y);
  }
}

var c = new Child(1.0, 2.0, 3.0, 4.0);
```

The compiler inserts a call to the parent’s `postinit` method in the child’s `postinit` method, and invokes the child’s `postinit` method after the compiler-generated initializer finishes:

```
Parent.postinit: 1.0, 2.0
Child.postinit: 3.0, 4.0
```

17.5 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. 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 or type field returns a parameter or type, respectively. In addition to being available for access with a class instance receiver, parameter and type fields can be accessed from the instantiated class type itself.
Example (useActor1.chpl). Given a variable `anActor` of type `Actor` as defined above, the code

```chapel
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`.

17.5.1 Variable Getter Methods

All field accesses are performed via getters. A getter is a method without parentheses with the same name as the field. It is defined in the field’s class and has a `ref` return intent (§13.7.1). If the program does not define it, the default getter, which simply returns the field, is provided.

Example (getterSetter.chpl). In the code

```chapel
class C {
    var setCount: int;
    var x: int;
    proc x ref {
        setCount += 1;
        return x;
    }
    proc x {
        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.

17.6 Class Method Calls

Class method calls are similar to other method calls which are described in §14.1. However, class method calls are subject to dynamic dispatch.

The receiver-clause (or its absence) specifies the method’s receiver in the same way it does for field accesses §17.5.

See (§14.2) for more details of about method receivers.

17.7 Common Operations

17.7.1 Class Assignment

Classes are assigned by reference. After an assignment from one variable of a class type to another, both variables reference the same class instance. Assignments from an `owned` variable to another `owned` or `shared` variable are an exception. They transfer ownership, leaving the source variable empty i.e. storing `nil`. 
Example (owned-assignment.chpl).

```chapel
// assume that C is a class
var a:owned C? = new owned C();
var b:owned C?; // default initialized to store 'nil'
b = a; // transfers ownership from a to b
writeln(a); // a is left storing 'nil'
```

In contrast, assignment for shared variables allows both variables to refer to the same class instance.

The following assignments between variables or expressions with different memory management strategies are disallowed:

- to owned from shared or borrowed, as it would not ensure unique ownership of the instance
- to shared from borrowed, as the original owner would be unaware of the shared ownership
- to owned, shared, or borrowed from unmanaged, as both the source and the destination would appear responsible for deleting the instance

### 17.8 Deleting Unmanaged Class Instances

Memory associated with unmanaged class instances can be reclaimed with the `delete` statement:

```chapel
delete-statement:
delete expression-list;
```

where the expression-list specifies the class objects whose memory will be reclaimed. Prior to releasing their memory, the deinitialization routines for these objects will be executed (§17.8.1). The expression-list can contain array expressions, in which case each element of that array will be deleted in parallel using a `forall` loop over the array. It is legal to delete a class variable whose value is `nil`, though this has no effect. If a class instance is referenced after it has been deleted, the behavior is undefined.

Example (delete.chpl). The following example allocates a new object `c` of class type `C` and then deletes it.

```chapel
var c : unmanaged C? = nil;
delete c;      // Does nothing: c is nil.

c = new unmanaged C();  // Creates a new object.
delete c;       // Deletes that object.

// The following statements reference an object after it has been deleted, so
// the behavior of each is "undefined":
// writeln(c.i); // May read from freed memory.
// c.i = 3;     // May overwrite freed memory.
// delete c;   // May confuse some allocators.
```
17.8.1 Class Deinitializer

A class author may create a deinitializer to specify additional actions to be performed when a class instance is deleted. A class deinitializer is a method named `deinit()`. It must take no arguments (aside from the implicit `this` argument). If defined, the deinitializer is called each time a `delete` statement is invoked with a valid instance of that class type. The deinitializer is not called if the argument of `delete` evaluates to `nil`. Note that when an `owned` or `shared` variable goes out of scope, it may call `delete` on a class instance which in turn will run the deinitializer and then reclaim the memory.

*Example (classDeinitializer.chpl).*

```chapel
class C {
  var i: int;
  proc deinit() { writeln("Bye, bye ", i); }
}

var c : unmanaged C? = nil;  // Does nothing: c is nil.
delete c;

c = new unmanaged C(1);  // Creates a new instance.
delete c;                // Deletes that instance: Writes out "Bye, bye 1"
                        // and reclaims the memory that was held by c.
{
  var own = new owned C(2);  // Creates a new owned instance
  // The instance is automatically deleted at
  // the end of this block, so "Bye, bye 2" is
  // output and then the memory is reclaimed.
}
A record is a data structure that is similar to a class except it 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. When the record variable goes out of scope, or at the end of the program if it is a global, it is deinitialized and its storage is deallocated.

A record declaration statement creates a record type §18.1. A variable of record type contains all and only the fields defined by that type (§18.1.1). Value semantics imply that the type of a record variable is known at compile time (i.e. it is statically typed).

A record can be created using the new operator, which allocates storage, initializes it via a call to a record initializer, 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 §24.3

18.1 Record Declarations

A record type is defined with the following syntax:

```
record-declaration-statement:
  simple-record-declaration-statement
  external-record-declaration-statement

simple-record-declaration-statement:
  record identifier { record-statement-list }

record-statement-list:
  record-statement
  record-statement record-statement-list

record-statement:
  variable-declaration-statement
  method-declaration-statement
  type-declaration-statement
  empty-statement
```

A record-declaration-statement defines a new type symbol specified by the identifier. As in a class declaration, the body of a record declaration can contain variable, method, and type declarations.

If a record declaration contains a type alias or parameter field, or it contains a variable or constant field without a specified type and without an initialization expression, then it declares a generic record type. Generic record types are described in §24.3.
If the `extern` keyword appears before the `record` keyword, then an external record type is declared. 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 (§32) for more information on external records.

*Future.* Privacy controls for classes and records are currently not specified, as discussion is needed regarding its impact on inheritance, for instance.

### 18.1.1 Record Types

A record type specifier simply names a record type, using the following syntax:

```chapel
class 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 24.3.6.

### 18.1.2 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

```chapel
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 §17.1.6.

### 18.1.3 Record Methods

A record method is a function or iterator that is bound to a record. See the methods section §14 for more information about methods.

Note that the receiver of a record method is passed by `ref` or `const ref` intent by default, depending on whether or not `this` is modified in the body of the method.
18.1.4 Nested Record Types

A record defined within another class or record is a nested record. A nested record can be referenced only within its immediately enclosing class or record.

18.2 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 with a call to an initializer (§17.4) that accepts zero actual arguments.

18.2.1 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.

18.2.2 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 (§17.4.2).

To create 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 (§18.2.1), except that the temporary record goes out of scope at the end of the enclosing block.

The initializers for a record are defined in the same way as those for a class (§17.4). Note that records do not support inheritance and therefore the initializer rules for inheriting classes (§17.4.4) do not apply to record initializers.

Example (recordCreation.chpl). The program
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 initializer with desired arguments. The records created with `new UniqueID()` are discarded after they are used.

As with classes, the user can provide their own initializers (§17.4.1). If any user-defined initializers are supplied, the default initializer cannot be called directly.

### 18.2.3 Record Deinitializer

A record author may specify additional actions to be performed before record storage is reclaimed by defining a record deinitializer. A record deinitializer is a method named `deinit()`. A record deinitializer takes no arguments (aside from the implicit `this` argument). If defined, the deinitializer is called on a record object after it goes out of scope and before its memory is reclaimed.

Example (recordDeinitializer.chpl).

```chapl
class C { var x: int; } // A class with nonzero size.
// If the class were empty, whether or not its memory was reclaimed
// would not be observable.

// Defines a record implementing simple memory management.
record R {
    var c: unmanaged C;
    proc init() {
        c = new unmanaged C(0);
    }
    proc deinit() {
        delete c;
    }
}
```
proc foo()
{
    var r: R; // Initialized using default initializer.
    writeln(r);
    // r will go out of scope here.
    // Its deinitializer will be called to free the C object it contains.
}

foo();

18.3 Record Arguments

Record formal arguments with the *in* intent will be copy-initialized into the function’s formal argument (§18.6.1). Record formal arguments with *inout* or *out* intent will be updated by the record assignment function (§18.6.2).

*Example (paramPassing.chpl)*. The program

```chapel
record MyColor {
    var color: int;
}
proc printMyColor(in 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);
```

produces

```
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.

18.4 Record Field Access

A record field is accessed the same way as a class field (§17.5). 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.
Accessing a parameter or type field returns a parameter or type, respectively. Also, parameter and type fields can be accessed from an instantiated record type in addition to from a record value.

### 18.4.1 Field Getter Methods

As in classes, field accesses are performed via getter methods (§17.5.1). 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.

### 18.5 Record Method Calls

Record method calls are written the same way as other method calls (§14.1). Unlike class methods, record methods are always resolved at compile time.

### 18.6 Common Operations

#### 18.6.1 Copy Initialization of Records

When a new record variable is created based upon an existing variable, we say that the new variable is *copy initialized* from the existing variable. For example:

```plaintext
var x: MyRecordType;
var y = x; // this is called copy initialization
```

We say that y is *copy initialized* from x. Since x and y both exist after y is initialized and they are different record variables, they must not be observed to alias. That is, modifications to a field in x should not cause the corresponding field in y to change.

Initializing a record variable with the result of a call returning a record by value simply captures the result of the call into the variable and does not cause copy initialization. For example:

```plaintext
proc returnsRecord() {
  var ret: MyRecordType;
  return ret;
}
var z = returnsRecord(); // captures result into z without copy initializing
```

*Future.* Specifying further details of copy initialization is future work.
18.6.2 Record Assignment

A variable of record type may be updated by assignment. The compiler provides a default assignment operator for each record type $R$ having the signature:

$$\text{proc } -(\text{ref } lhs:R, \text{rhs}:R) : \text{void where } lhs.\text{type} == rhs.\text{type};$$

In it, the value of each field of the record on the right-hand side is assigned to the corresponding field of the record on the left-hand side.

The compiler-provided assignment operator may be overridden as described in 11.3.

The following example demonstrates record assignment.

Example (assignment.chpl).

```chapel
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;
C.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 $A$.

18.6.3 Default Comparison Operators

Default functions to overload $==$ and $!=$ are defined for records if none are explicitly defined. These functions have the following signatures for a record $R$:

$$\text{proc } == (lhs:R, rhs:R) : \text{bool where } lhs.\text{type} == rhs.\text{type};$$
$$\text{proc } != (lhs:R, rhs:R) : \text{bool where } lhs.\text{type} == rhs.\text{type};$$

Each of these compare the fields, one at a time, returning $false$ if the property is not satisfied by the given pair of fields.

18.7 Differences between Classes and Records

The key differences between records and classes are listed below.
18.7.1 Declarations

Syntactically, class and record type declarations are identical, except that they begin with the `class` and `record` keywords, respectively. In contrast to classes, records do not support inheritance.

18.7.2 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.

18.7.3 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.

18.7.4 Arguments

Record arguments use the `const ref` intent by default - in contrast to class arguments which pass by `const` in intent by default.

Similarly, the `this` receiver argument is passed by `const` in by default for class methods. In contrast, it is passed by `ref` or `const ref` by default for record methods.

18.7.5 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.
18.7.6  The *delete* operator

Calling *delete* on a record is illegal.

18.7.7  Default Comparison Operators

For records, the compiler will supply default comparison operators if they are not supplied by the user. In contrast, the user cannot redefine `==` and `!=` for classes. The default comparison operators for a record examine the arguments’ fields, while the comparison operators for classes check whether the l.h.s. and r.h.s. refer to the same class instance or are both `nil`. 
19 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 initialized, it is in an unset state so that no field contains data.

19.1 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.

19.2 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-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.

19.2.1 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.
19.3 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.
20 Ranges

A range is a first-class, constant-space representation of a regular sequence of indices of either integer, boolean, or enumerated type. Ranges support iteration over the sequences they represent and are the basis for defining domains (§21).

Ranges are presented as follows:

- definition of the key range concepts §20.1
- range types §20.2
- range values §20.3
- range assignment §20.4.1
- operators on ranges §20.5
- predefined functions on ranges §20.6

20.1 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 a specific index value or $-\infty$.
- The high bound is either a specific index value or $+\infty$. The low and high bounds determine the span of the represented sequence. Chapel does not represent $\infty$ explicitly. Instead, infinite bound(s) are represented implicitly in the range’s type (§20.2). When the low and/or high bound is $\infty$, 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:
  - $\text{stride} > 0$ indicates an increasing sequence,
  - $\text{stride} < 0$ indicates a decreasing sequence.
- The alignment is either a specific index value or is ambiguous. It defines how the represented sequence’s members are aligned relative to the stride. 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.
More formally, the represented sequence for the range \((low, high, stride, alignmt)\) contains all indices \(ix\) such that:

\[
\begin{align*}
low & \leq ix \leq high \quad \text{and} \quad ix \equiv alignmt \pmod{|stride|} \quad \text{if} \quad alignmt \text{ is not ambiguous} \\
low & \leq ix \leq high \quad \text{if} \quad stride = 1 \text{ or } stride = -1 \\
\text{the represented sequence is undefined} & \quad \text{otherwise}
\end{align*}
\]

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. A common case of this occurs when the high bound is greater than the low bound.

We will say that a value \(ix\) is aligned w.r.t. the range \((low, high, stride, alignmt)\) if:

- \(alignmt\) is not ambiguous and \(ix \equiv alignmt \pmod{|stride|}\), or
- \(stride\) is 1 or -1.

Furthermore, \(\infty\) 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 \(-\infty\),
  - the represented sequence is decreasing and the high bound is \(+\infty\).

- 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 \(+\infty\),
  - the represented sequence is decreasing and the low bound is \(-\infty\).

- The aligned low bound is the smallest value that is greater than or equal to the low bound and is aligned w.r.t. the range, if such a value exists.
  The aligned low bound equals the smallest member of the represented sequence, when both exist.

- The aligned high bound is the largest value that is less than or equal to the high bound and is aligned w.r.t. the range, if such a value 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 it has a first index.
Ranges

20.2 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 $\infty$, the represented sequence also contains indices that are not representable by idxType. idxType must be an integral, boolean, or enumerated type and is int by default. The range’s low bound and high bound (when they are not $\infty$) 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 for integral ranges; for boolean and enumerated ranges, it is simply int.

- **boundedType** indicates which of the range’s bounds are not $\infty$. 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:

```cpp
enum BoundedRangeType { bounded, boundedLow, boundedHigh, boundedNone);
```

The value of boundedType determines which bounds of the range are specified (making the range “bounded”, as opposed to infinite, in the corresponding direction(s)) as follows:

- **bounded**: both bounds are specified.
- **boundedLow**: the low bound is specified (the high bound is $+\infty$).
- **boundedHigh**: the high bound is specified (the low bound is $-\infty$).
- **boundedNone**: neither bound is specified (both bounds are $\infty$).

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:

```cpp
range-type:
   range ( named-expression-list )
```

That is, a range type is obtained as if by invoking the range type constructor (§24.3.6) 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 32-bit integers, with both high and low bounds specified, and the ability to have a stride other than 1.

\[
\text{var } r: \text{range}(\text{int}(32), \text{BoundedRangeType.bounded, stridable=\text{true}}); 
\]

### 20.3 Range Values

A range value consists of the range’s four primary properties (§20.1): low bound, high bound, stride and alignment.

#### 20.3.1 Range Literals

Range literals are specified with the following syntax.

\[
\text{range-literal:} \\
\text{expression } \ldots \text{expression} \\
\text{expression } \ldots \text{expression} \\
\ldots \text{expression} \\
\ldots
\]

The expressions to the left and to the right of \( \ldots \), 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 type, then \( \text{idxType} \) is that type.
  - If both the low bound and the high bound expressions are given and an implicit conversion is allowed from one expression’s type to the other’s, then \( \text{idxType} \) is that type.
  - If only one bound expression is given and it has an integral, boolean, or enumerated type, then \( \text{idxType} \) is that type.
  - If neither bound expression is given, then \( \text{idxType} \) is int.
  - Otherwise, the range literal is not legal.

- **boundedType** is a value of the type \( \text{BoundedRangeType} \) 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,
Ranges

- `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.

### 20.3.2 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.

### 20.4 Common Operations

All operations on a range return a new range rather than modifying the existing one. This supports a coding style in which all ranges are `immutable` (i.e. declared as `const`).

**Rationale.**

The intention is to provide ranges as immutable objects.

Immutable objects may be cached without creating coherence concerns. They are also inherently thread-safe. In terms of implementation, immutable objects are created in a consistent state and stay that way: Outside of initializers, internal consistency checks can be dispensed with.

These are the same arguments as were used to justify making strings immutable in Java and C#.
20.4.1 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 is not stridable.

20.4.2 Range Comparisons

Ranges can be compared using equality and inequality.

```chapel
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.
```

20.4.3 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.

*Cray’s Chapel Implementation.* An attempt to iterate over a range causes an error if adding stride to the range’s last index overflows its index type, i.e. if the sum is greater than the index type’s maximum value, or smaller than its minimum value.

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 (§11.9.1), it generates as many indices as needed to match the other iterator(s).

*Example (zipWithUnbounded.chpl).* The code

```chapel
for i in zip(1..5, 3..) do
  write(i, "; ");
```

produces the output

```plaintext
(1, 3); (2, 4); (3, 5); (4, 6); (5, 7);
```
20.4.4 Range Promotion of Scalar Functions

Range values may be passed to a scalar function argument whose type matches the range’s index type. This results in a promotion of the scalar function as described in §27.5.

Example (rangePromotion.chpl). Given a function `addOne(x:int)` that accepts `int` values and a range `1..10`, the function `addOne()` can be called with `1..10` as its actual argument which will result in the function being invoked for each value in the range.

```chapel
proc addOne(x:int) {
    return x + 1;
}
var A:[1..10] int;
A = addOne(1..10);
```

The last statement is equivalent to:

```chapel
forall (a,i) in zip(A,1..10) do
    a = addOne(i);
```

20.5 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 §20.6.

range-expression:
expression
strided-range-expression
counted-range-expression
aligned-range-expression
sliced-range-expression

20.5.1 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:
The type of the step must be a signed or unsigned 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. It is an error for the step to be zero.

*Future.* We may 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.

```chapel
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.

### 20.5.2 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:

```chapel
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, but with the `stridable` parameter set to `true`. 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).**

```chapel
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).**

```chapel
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 (§20.6.4).

### 20.5.3 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 type of the count expression must be a signed or unsigned integer of the same bit size as the base range’s `idxType`, or an implicit conversion must be allowed to that type from the count’s type.

The type of the result of the `#` operator is the type of the range argument.

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 \( c \times stride - 1 \) 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.

If 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 \( \text{count} \times \text{stride} - 1 \) 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.

### 20.5.4 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
```
20.5.5 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 sequence cannot be expressed as a range of the original type, the slice expression evaluates to the empty range `1..0`. 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.

```chpl
define r = 1..20;
define r2 = r[3..];
define r3 = r[1.. by 2];
define r4 = r3[0.. by 3];
```

20.6 Predefined Functions on Ranges

20.6.1 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.
20.6.2 Range Properties

Most of the methods in this subsection report on the range properties defined in §20.1. A range’s represented sequence can be examined, for example, by iterating over the range in a for loop §11.9.

*Open issue.* The behavior of the methods that report properties that may be undefined, $\infty$, or ambiguous, may change.

**proc range.aligned : bool**

Reports whether the range’s alignment is unambiguous.

**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:

```chapel
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 $\infty$.

**proc range.hasLast() : bool**

Reports whether the range has the last index.

**proc range.hasLowBound() param: bool**
Ranges

Reports whether the range’s low bound is not $-\infty$.

**proc** `range.high : idxType`

Returns the range’s high bound. If the high bound is $+\infty$, 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 $-\infty$, the behavior is undefined. See also `hasLowBound`.

**proc** `range.size : idxType`

Same as `range.length`.

**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.

### 20.6.3 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:
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.

20.6.4 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 \((l, l - (i - 1), s, a)\),
Ranges

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 \texttt{interior()} preserves the alignment, and it uses the low and high bounds rather than \texttt{first} and \texttt{last} to establish the bounds of the resulting range. If the operand range is ambiguously aligned, then so is the resulting range.

\textbf{proc} \texttt{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.

\textbf{proc} \texttt{range.translate(i: integral)}

Returns a new range with its \texttt{low}, \texttt{high} and \texttt{alignment} values adjusted by \( i \). The \texttt{stride} value is preserved. If the range’s alignment is ambiguous, the behavior is undefined.
21 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 (§22), 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 §29, thus supporting global-view data structures.

In the next subsection, we introduce the key characteristics of domains. In §21.2, we discuss the types and values that can be associated with a base domain. In §21.3, we discuss the types and values of simple subdomains that can be created from those base domains. In §21.4, we discuss the types and values of sparse subdomains. The remaining sections describe the important manipulations that can be performed with domains, as well as the predefined operators and functions defined for domains.

21.1 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:

\[
\text{domain-type:} \\
\text{base-domain-type} \\
\text{simple-subdomain-type} \\
\text{sparse-subdomain-type}
\]

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 domains are ordered. All 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 (§21.5), and arrays (§22.11).

The runtime representation of a domain is controlled by its domain map. Domain maps are presented in §29.
21.2 Base Domain Types and Values

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.

21.2.1 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 homogeneous tuple type **rank*** **idxType**. If unspecified, **idxType** defaults to int and **stridable** defaults to false.

*Open issue.* We may represent a rectangular domain’s index type 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 uninitialized rectangular domain with three dimensions, with int indices:

```chapel
var D1 : domain(rank=3, idxType=int, stridable=false);
var D2 : domain(3);
```
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 \( n \times \text{idxType} \) 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.\(^1\)

\(^1\)This is also known as row-major ordering.

\textit{Future}. Domains defined using unbounded ranges may be supported.

Literal rectangular domain values are represented by a comma-separated list of range expressions of matching \text{idxType} enclosed in curly braces:

\[
\text{rectangular-domain-literal:} \quad \{ \text{range-expression-list} \} \\
\text{range-expression-list:} \quad \text{range-expression} \quad \text{range-expression, range-expression-list}
\]

The type of a rectangular domain literal is defined as follows:

- \text{rank} = the number of range expressions in the literal;
- \text{idxType} = the type of the range expressions;
- \text{stridable} = \text{true} if any of the range expressions are stridable, otherwise \text{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 \text{idxType}. Otherwise, the domain literal is invalid.

\textit{Example}. The expression \( \{1..5, 1..5\} \) defines a rectangular domain with type \text{domain(rank=2, idxType=int, stridable=false)}. It is a \( 5 \times 5 \) domain with the indices:

\[
(1,1), (1,2), \ldots, (1,5), (2,1), \ldots, (5,5).
\]  

A domain expression may contain bounds which are evaluated at runtime.

\textit{Example}. In the code

\begin{verbatim}
var D: domain(2) = {1..n, 1..n};
\end{verbatim}

\( D \) is defined as a two-dimensional, nonstridable rectangular domain with an index type of \text{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 domain type is the \text{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 array indices are iterated over using the domain’s \texttt{dim()} method, and each element is filled with some value. Then the array is printed out.

Thus, the code

\begin{verbatim}
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);
\end{verbatim}

produces

\begin{verbatim}
  8  9 10 11 12 13 14
29 30 31 32 33 34 35
\end{verbatim}

21.2.2 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 (hash tables). The type of indices of an associative domain, or its \texttt{idxType}, can be any primitive type except \texttt{void} or any class type.

Associative Domain Types

An associative domain type is parameterized by \texttt{idxType}, the type of the indices that it stores. The syntax is as follows:

\begingroup
\small
\begin{verbatim}
associative-domain-type:
  domain ( associative-index-type )
  domain ( opaque )

associative-index-type:
  type-expression
\end{verbatim}
\endgroup

The three expansions of \texttt{associative-domain-type} correspond to the three kinds of associative domain listed below.

1. In general, \texttt{associative-index-type} determines \texttt{idxType} of the associative domain type.

2. Opaque domains are a special case, indicated by the type \texttt{opaque}. Anonymous values of the type \texttt{opaque} are used as index values in this case.

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. Opaque domains can be used to build unstructured arrays that are 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 iteration order over
the indices of an associative domain is undefined.

Specification of an associative domain literal value follows a similar syntax as rectangular domain literal values. What
differentiates the two are the types of expressions specified in the comma separated list. Use of values of a type other
than ranges will result in the construction of an associative domain.

\[
\text{associative-domain-literal:} \\
\{ \text{associative-expression-list} \}
\]

\[
\text{associative-expression-list:} \\
\text{non-range-expression} \\
\text{non-range-expression, associative-expression-list}
\]

\[
\text{non-range-expression:} \\
\text{expression}
\]

It is required that the types of the values used in constructing an associative domain literal value be of the same type.
If the types of the indices does not match a compiler error will be issued.

Future. Due to implementation of \(==\) over arrays it is currently not possible to use arrays as indices
within an associative domain.

Open issue. Assignment of an associative domain literal results in a warning message being printed
alerting the user that whole-domain assignment has been serialized. This results from the resize operation
over associative arrays not being parsafe.

Example (associativeDomain.chpl). The following example illustrates construction of an associative
domain containing string indices “bar” and “foo”. Note that due to internal hashing of indices the order in
which the values of the associative domain are iterated is not the same as their specification order.

This code

\[
\text{var D : domain\{string\} = \{"bar", "foo"\};}
\text{writeln(D);} \\
\]

produces the output

\[
\{\text{foo, bar}\}
\]

If uninitialized, the default value of an associative domain is the empty index set.

Indices can be added to or removed from an associative domain as described in §21.8.6
21.3 Simple Subdomain Types and Values

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 (§21.4). A simple subdomain inherits its representation (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.

21.3.1 Simple Subdomain Types

A simple subdomain type is specified using the following syntax:

\[ \text{simple-subdomain-type: \subdomain ( domain-expression )} \]

This declares that \textit{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.

21.3.2 Simple Subdomain Values

The value of a simple subdomain is the set of all index values that the subdomain describes.

The default value of a simple subdomain type is the same as the default value of its parent’s type (§21.2.1 §21.2.2).

A simple subdomain variable can be initialized or assigned to with a tuple of values of the parent’s \texttt{idxType}. Indices can also be added to or removed from a simple subdomain as described in §21.8.6. It is an error to attempt to add an index to a subdomain that is not also a member of the parent domain.
21.4 Sparse Subdomain Types and Values

```
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 §22.10 for details about sparse arrays.

### 21.4.1 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.

### 21.4.2 Sparse Subdomain Values

A sparse subdomain’s value is simply the set of all index values that the domain describes. If the parent domain defines an iteration order over its indices, the sparse subdomain inherits that order.

There is no literal syntax for a sparse subdomain. However, a variable of a sparse subdomain type can be initialized using a tuple of values of the parent domain’s index type.

The default value for a sparse subdomain value is the empty set.

**Example.** The following code declares a two-dimensional dense domain \( D \), followed by a two dimensional sparse subdomain of \( D \) named \( \text{SpsD} \). Since \( \text{SpsD} \) 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);
```

21.5 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.

### 21.6 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 defines an iteration order of its indices, then the indices are visited in that order.

The type of the iterator variable for an iteration over a domain named `D` is that domain’s index type, `index(D)`.

### 21.7 Domains as Arguments

This section describes the semantics of passing domains as arguments to functions.

#### 21.7.1 Formal Arguments of Domain Type

When a domain value is passed to a formal argument of compatible domain type by default intent, it is passed by reference in order to preserve the domain’s identity.

#### 21.7.2 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 §27.5
Domains

Example. Given a function \texttt{foo()} that accepts real floating point values and an associative domain \( D \) of type \texttt{domain(real)}, \texttt{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 \texttt{int} declared over a one-dimensional domain \( D \) with \texttt{idxType int}, the array elements can be assigned their corresponding index values by writing:

\[
A = D;
\]

This is equivalent to:

\[
\text{forall } (a, i) \text{ in } \text{zip}(A, D) \text{ do } a = i;
\]

21.8 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.

21.8.1 Domain Assignment

All domain types support domain assignment.

\[
\begin{align*}
\text{domain-expression:} & \\
\quad & \text{domain-literal} \\
\quad & \text{domain-name} \\
\quad & \text{domain-assignment-expression} \\
\quad & \text{domain-striding-expression} \\
\quad & \text{domain-alignment-expression} \\
\quad & \text{domain-slice-expression} \\
\text{domain-literal:} & \\
\quad & \text{rectangular-domain-literal} \\
\quad & \text{associative-domain-literal} \\
\text{domain-assignment-expression:} & \\
\quad & \text{domain-name} = \text{domain-expression} \\
\text{domain-name:} & \\
\quad & \text{identifier}
\end{align*}
\]

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 §22.11.

It is an error to assign a stridable domain to an unstridable domain without an explicit conversion.
**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 \((1,1)\) to \((n,n)\). 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");
```

### 21.8.2 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.

### 21.8.3 Domain Alignment

The `align` operator can be applied to a rectangular domain value in order to change the alignment of a rectangular domain value. The right-hand operand to the `align` operator can either be an integral value or an integral tuple whose size matches the domain’s rank.

```
domain–alignment–expression:
domain–expression align 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 alignment value, the value of the resulting domain is computed by applying the integer value to each range in the value using the `align` operator. In the case of a tuple alignment value, the resulting domain’s value is computed by applying each tuple component to the corresponding range using the `align` operator.

### 21.8.4 Domain Slicing

Slicing is the application of an index set to a domain. It can be written using either parentheses or square brackets. The index set can be defined with either a domain or a list of ranges.

```
domain–slice–expression:
domain–expression [ slicing–index–set ]
domain–expression ( slicing–index–set )
```

```
slicing–index–set:
domain–expression
range–expression–list
```
The result of slicing, or a slice, is a new domain value that represents the intersection of the index set of the domain being sliced and the index set being applied. The type and domain map of the slice match the domain being sliced.

Slicing can also be performed on an array, resulting in aliasing a subset of the array’s elements (§22.6).

**Domain-based Slicing**

If the brackets or parentheses contain a domain value, its index set is applied for slicing.

*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 brackets or parentheses can contain 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. The Cartesian product of the ranges’ index sets is applied for slicing.

*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 \( \text{InnerD} \) domain describes the inner indices of \( D \), \( \text{Col2OfD} \) describes the 2nd column of \( D \), and \( \text{AllButLastRow} \) describes all of \( D \) except for the last row.

```c
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 §22.6.1.

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.

**21.8.5 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 21.8.4.
21.8.6 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 §22.11.

21.9 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.

21.9.1 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 (clearAssociativeDomain).* This function provides a way to produce an empty associative domain.

When run, the code

```haskell
enum Counter { one, two, three };
var D : domain ( Counter ) = {Counter.one, Counter.two};
writeln("D has ", D.numIndices, " indices.");
D.clear();
writeln("D has ", D.numIndices, " indices.");
```

prints out

```
D has 2 indices.
D has 0 indices.
```

**proc** Domain.dist : dmap

Returns the domain map that implements this domain

*Example (getDomainMap).* In the code
dist is used in a where-clause to determine the type of the argument’s distribution. The output is:

```
Block-distributed domain
```

**proc Domain.idxType**

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 \((-1):\text{idxType}\). 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 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.
21.9.2 Methods on Regular Domains

The methods described in this subsection can be applied to regular domains only.

\begin{verbatim}
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 \( \text{off} \) or \( \text{off}(d) \) is positive or contracted in dimension \( d \) if \( \text{off} \) or \( \text{off}(d) \) 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 \( \text{off} \) or \( \text{off}(d) \) indices for each dimension \( d \). If \( \text{off} \) or \( \text{off}(d) \) 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 \( \text{off} \) or \( \text{off}(d) \) indices for each dimension \( d \). If \( \text{off} \) or \( \text{off}(d) \) 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.size: capType
\end{verbatim}
Same as $Domain$.numIndices.

\begin{verbatim}
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 $d$.
\end{verbatim}

21.9.3 Methods on Irregular Domains

The following methods are available only on irregular domain types.

\begin{verbatim}
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
$d1$.

proc requestCapacity(s: int)

Resizes the domain internal storage to hold at least $s$ indices.
\end{verbatim}
22 Arrays

An array is a map from a domain’s indices to a collection of variables of homogeneous 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 (§29).

22.1 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-expression
```

The `domain-expression` must specify a domain that the array can be declared over. If the `domain-expression` is a domain literal, the curly braces around the literal may be omitted.

Example (decls.chpl). In the code

```chapel
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), . . . , (1, 10), (2, 1), . . . , (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`.

```chapel
var A: [D] real;
var x: A.eltType;
```

22.2 Array Values

An array’s value is the collection of its elements’ values. Assignments between array variables are performed by value as described in §22.5. Chapel semantics are defined so that the compiler will never need to insert temporary arrays of the same size as a user array variable.

Array literal values can be either rectangular or associative, corresponding to the underlying domain which defines its indices.

```
array-literal:
  rectangular-array-literal
  associative-array-literal
```
22.2.1 Rectangular Array Literals

Rectangular array literals are specified by enclosing a comma separated list of expressions representing values in square brackets. A 1-based domain will automatically be generated for the given array literal. The type of the array’s values will be the type of the first element listed. A trailing comma is allowed.

\[
\text{rectangular-array-literal:} \quad [ \text{expression-list} ] \\
[ \text{expression-list} , ]
\]

Example (decl-literal.chpl). The following example declares a 5 element rectangular array literal containing strings, then subsequently prints each string element to the console.

```chapel
var A = ["1", "2", "3", "4", "5"];
for i in 1..5 do writeln(A[i]);
```

Future. Provide syntax which allows users to specify the domain for a rectangular array literal.

Future. Determine the type of a rectangular array literal based on the most promoted type, rather than the first element’s type.

Example (decl-with-anon-domain.chpl). The following example declares a 2-element array \( \mathbf{A} \) containing 3-element arrays of real numbers. \( \mathbf{A} \) is initialized using array literals.

```
var \( \mathbf{A} \): [1..2] [1..3] real = [[1.1, 1.2, 1.3], [2.1, 2.2, 2.3]];
```

Open issue. We would like to differentiate syntactically between array literals for an array of arrays and a multi-dimensional array.

An rectangular array’s default value is for each array element to be initialized to the default value of the element type.

22.2.2 Associative Array Literals

Associative array values are specified by enclosing a comma separated list of index-to-value bindings within square brackets. It is expected that the indices in the listing match in type and, likewise, the types of values in the listing also match. A trailing comma is allowed.

\[
\text{associative-array-literal:} \quad [ \text{associative-expr-list} ] \\
[ \text{associative-expr-list} , ]
\]

\[
\text{associative-expr-list:} \quad \text{index-expr => value-expr} \\
\text{index-expr => value-expr, associative-expr-list}
\]
Arrays

index–expr:
   expression

value–expr:
   expression

Open issue. Currently it is not possible to use other associative domains as values within an associative array literal.

Example (decl-assocLiteral.chpl). The following example declares a 5 element associative array literal which maps integers to their corresponding string representation. The indices and their corresponding values are then printed.

```chapel
var A = [1 => "one", 10 => "ten", 3 => "three", 16 => "sixteen"];
for da in zip (A.domain, A) do
   writeln(da);
```

22.2.3 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 §29 for more details.

22.3 Array Indexing

Arrays can be indexed using index values from the domain over which they are declared. Array indexing is expressed using either parentheses or square brackets. This results in a reference to the element that corresponds to the index value.

Example (array-indexing.chpl). Given:

```chapel
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;
```

Except for associative arrays, 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.
22.3.1 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:

```chpl
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

```chpl
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.

22.3.2 Associative Array Indexing

Indices can be added to associative arrays through the array’s domain.

Example (assoc-add-index.chpl). Given:

```chpl
var D : domain<string>
var A : [D] int;
```

the array A initially contains no elements. We can change that by adding indices to the domain D:

```chpl
D.add("a");
D.add("b");
```

The array A can now be indexed with indices "a" and "b":

```chpl
A["a"] = 1;
A["b"] = 2;
var x = A["a"];`
22.4 Iteration over Arrays

All arrays support iteration via standard `for`, `forall` and `coforall` loops. These loops iterate over all of the array elements as described by its domain. A loop of the form:

\[
\text{[for|forall|coforall] } a \text{ in } A \text{ do } ...
\]

is semantically equivalent to:

\[
\text{[for|forall|coforall] } i \text{ in } A.\text{domain do } ...
\]

The iterator variable for an array iteration is a reference to the array element type.
22.5 Array Assignment

Array assignment is by value. Arrays can be assigned arrays, ranges, domains, iterators, or tuples as long as the two expressions are compatible in terms of number of dimensions and shape.

Example (assign.chpl). If \( A \) is an array variable and \( B \) is an expression of array, range, domain, or tuple type, or an iterator, then the assignment

\[
A = B;
\]

is equivalent to

\[
[(a,b) \text{ in zip}(A,B)] \ a = b;
\]

If the zipper iteration is illegal, then the assignment is illegal. This means, for example, that a range cannot be assigned to a multidimensional rectangular array because the two expressions don’t match in shape and can’t be zipped together. Notice that the assignment is implemented using parallelism when possible, and serially otherwise.

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

\[
\text{forall} \ a \in A \ do \ a = e;
\]

22.6 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

\[
\begin{align*}
\text{var } & \text{OuterD: domain}(2) = \{0..n+1, 0..n+1\}; \\
\text{var } & \text{InnerD: domain}(2) = \{1..n, 1..n\}; \\
\text{var } & A, B: [\text{OuterD}] \text{ real};
\end{align*}
\]

the assignment given by

\[
A[\text{InnerD}] = B[\text{InnerD}];
\]

assigns the elements in the interior of \( B \) to the elements in the interior of \( A \).
22.6.1 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 §22.6 and then using that subdomain to slice the array.

22.6.2 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

```chapel
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 $A$.

22.7 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 # operator to the array’s domain and using the result to slice the array as described in Section 22.6.1.

22.8 Array Arguments to Functions

By default, arrays are passed to function by ref or const ref depending on whether or not the formal argument is modified. The in, inout, and out intent can create copies of arrays.

When a formal argument has array type, the element type of the array can be omitted and/or the domain of the array can be queried or omitted. In such cases, the argument is generic and is discussed in §24.1.7.

If a formal array argument specifies a domain as part of its type signature, the domain of the actual argument must represent the same index set. If the formal array’s domain was declared using an explicit domain map, the actual array’s domain must use an equivalent domain map.
22.8.1 Array Promotion of Scalar Functions

Arrays may be passed to a scalar function argument whose type matches the array’s element type. This results in a promotion of the scalar function as defined in §27.5.

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.

22.9 Returning Arrays from Functions

Arrays return by value by default. The ref and const ref return intents can be used to return a reference to an array. Similarly to array arguments, the element type and/or domain of an array return type can be omitted.

22.10 Sparse Arrays

Sparse arrays in Chapel are those whose domain is sparse. 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 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.
22.11 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 default 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 (§24.1.7).

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.

22.12 Predefined Functions and Methods on Arrays

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

```plaintext
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.

**proc** `Array.size: this.domain.dim_type`

Same as `Array.numElements`. 


23 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.

23.1 Iterator Definitions

The syntax to declare an iterator is given by:

```
iterator-declaration-statement:
  privacy-specifier_opt iter iterator-name argument-list_opt yield-intent_opt yield-type_opt where-clause_opt
  iterator-body

iterator-name:
  identifier

yield-intent:
  const
  const ref
  ref
  param
  type

yield-type:
  : type-expression

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.
- The intent and type specified after the argument list refer to the type yielded, not the type returned (see previous bullet). However, they are syntactically the same as a *return-intent* and a *return-type*.

*Open issue.* Iterators that yield types or params are not currently supported.
23.2 The Yield Statement

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

\[
\text{yield-statement: yield expression ;}
\]

When an iterator is executed and a yield is encountered, the value of the yield expression is returned to the iterator’s callsite. However, the state of execution of the iterator is logically saved such that its execution continues from the point immediately following the yield statement. A yield statement in an iterator that yields references must yield an lvalue expression.

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.

23.3 Iterator Calls

Iterators are invoked using regular call expressions:

\[
\text{iteratable-call-expression: call-expression}
\]

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.

23.3.1 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.

23.3.2 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 this iterator

\[
\text{iter squares}(n: \text{int}); \text{int} \{
\begin{align*}
\text{for } i \in 1..n \text{ do} \\
\text{yield } i \ast i;
\end{align*}
\}
\text{var } A = \text{squares}(5);
\]

then the variable A will be an array storing:

\[
1 \ 4 \ 9 \ 16 \ 25
\]


23.3.3 Iterators and Generics

An iterator call expression can be passed to a generic function argument that has neither a declared type nor default value (§24.1.3). 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.

23.3.4 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:

```chapel
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.

23.3.5 Iterator Promotion of Scalar Functions

Iterator calls may be passed to a scalar function argument whose type matches the iterator’s yielded type. This results in a promotion of the scalar function as described in §27.5.

Example (iteratorPromotion.chpl). Given a function `addOne(x:int)` that accepts `int` values and an iterator `firstN()` that yields `int` values, `addOne()` can be called with `firstN()` as its actual argument. This pattern creates a new iterator that yields the result of applying `addOne()` to each value yielded by `firstN()`.

```chapel
proc addOne(x:int) { return x + 1; }
iter firstN(n:int) {
  for i in 1..n {
    yield i;
  }
}
for number in addOne(firstN(10)) {
  writeln(number);
}
```
23.4 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 iterable 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 (§27.7).

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.
24 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.

24.1 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.

24.1.1 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.

```chapel
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:

```chapel
var t2 = build2Tuple(int, string);
t2 = (1, "hello");
```
A formal type argument can include a formal type (a colon followed by a type). This pattern is sometimes useful to create generic functions accepting type arguments that only apply to a specific group of types.

*Example (typeColonArgument.chpl).* Suppose that we’d like to define a function that accepts a type argument and returns 1 represented in that type.

```chapel
proc getOne(type t:numeric) {
    return 1:t;
}
```

Now calls to this function will resolve to the appropriate version based upon the argument type supplied.

```chapel
var anInt8 = getOne(int(8));
var aReal = getOne(real);
```

### 24.1.2 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`.

```chapel
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.

### 24.1.3 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`. Additionally the parameter argument can allow any type be passed. The following code defines a function that returns a homogeneous tuple of size `p` where each component in the tuple is initialized to `x`:

```chapel
proc fillTuple(param p, 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)`. 

24.1.4 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:

```chapel
proc fillTuple(param p: int, x: ?t) {
  var result: p*t;
  for param i in 1..p do
    result(i) = x;
  return result;
}
```

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.

The code

```chapel
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;
}
```

produces the output

```
6
12.0
```

24.1.5 Formal Arguments of Generic Type

If the type of a formal argument is a generic type, there must exist an instantiation of that type that the actual argument can be implicitly coerced to ($\S$9.1). A copy of the function is instantiated for each unique instantiation of the formal’s type.

Example. The following code defines a function `writeTop` that takes an actual argument that is a generic stack (see §24.6) and outputs the top element of the stack. The function is generic on the type of its argument.

```chapel
proc writeTop(s: Stack) {
  write(s.top.item);
}
```
Types and parameters may be queried from the types of formal arguments as well. In the example above, the formal argument’s type could also be specified as `Stack(?t)` in which case the symbol `t` 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.

```chpl
class C {
    type t = int;
}
proc f(c: C) {
    // c.type is always int
}
proc g(c: C(?)) {
    // c.type may not be int
}
```

### 24.1.6 Formal Arguments of Partially Generic Type

The generic type for a formal argument may be specified with some queries or generic types and some concrete types or values. Using concrete types or values in this manner makes the argument *partially concrete* for the purpose of function resolution.

**Example (nested-type-queries.chpl).**

Given the code:

```chpl
class C {
    type elementType;
    type indexType;
    type containerType;
}
class Container {
    type containedType;
}
proc f(c: C(real,?t,?u)) {
    // ...
}
```

The function `f` can only apply when the `c.elementType==real`.

It’s also possible to use a generic type as an argument to `C`. The following function, `g`, can only apply when `c.containerType` is an instance of `Container`:

```chpl
proc g(c: C(?t,?u,Container)) {
    // ...
}
```

Similarly, a tuple type with query arguments forms a *partially concrete* argument.

**Example.** The function definition
Generics

```
proc f(tuple: (?t,real)) {
   // body
}
```

specifies that `tuple.size == 2 && tuple(2).type == real`.

Homogeneous tuple arguments of generic type are also supported:

```
record Number {
   var n;
}
proc f(tuple: 2*Number) {
}
```

specifies that \(f\) accepts a tuple with 2 elements, where each element has the same type, and that type is instantiation of `Number`.

Note that specifying a tuple consisting entirely of queried types does create a partially concrete argument because the size of the tuple is constrained.

```
Example (partially-concrete-star-tuple.chpl).
record Number {
   var n;
}
proc f(tuple: 2*Number) {
}
```

```
Example (partially-concrete-tuple-ambiguity.chpl).
The following program results in an ambiguity error:
proc f(tuple: (?,real)) {
}
proc f(tuple: (?,?)) {
}
f((1.0, 2.0));
```

since the tuple arguments in both versions of \(f\) are partially concrete.

24.1.7 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 §22.11 for rationale).

24.2 Function Visibility in Generic Functions

When resolving function calls made within generic functions, there is an additional source of visible functions. Besides functions visible to the generic function’s point of declaration, visible functions are also taken from one of the call sites 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.
Consider the following code which defines a generic function `bar`:

```chapel
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.

### 24.3 Generic Types

Generic types comprise built-in generic types, generic classes, and generic records.

#### 24.3.1 Built-in Generic Types

The types `integral`, `numeric` and `enum` are generic types that can only be instantiated with, respectively, the signed and unsigned integral types, all of the numeric types, and all enumerated types. The type `enumerated` is currently available as a synonym for `enum`. 
The type `record` can be instantiated with any record type.

The memory management strategies `owned`, `shared`, `borrowed`, and `unmanaged` (see §17.1.2) are also generic types that can be instantiated with any class using that memory management strategy. These types indicate generic nilability.

The types `class` and `class?`, on their own or in combination with memory management strategies, are also generic types. They can be instantiated as follows:

- `class` can instantiate with any non-nilable class using any memory management strategy
- `class?` can instantiate with any class using any memory management strategy but will use the nilable variant of that class in an instantiation. When used as an argument type, a value of non-nilable class type will be implicitly converted to the nilable type on the call. As a result, a formal of type `class?` can accept an actual of any class type.
- `owned` can instantiate with any `owned` class - taking the nilability from whatever it instantiated from.
- `owned` can instantiate with any non-nilable `owned` class.
- `owned` can instantiate from any nilable `owned` class. As with `class?`, it can also instantiate from a non-nilable `owned` class, in which case a implicit conversion would occur in a call.
- `shared`, `shared` class, `shared` class? behave similarly to the above but with `shared` management strategy.
- `borrowed`, `borrowed` class, `borrowed` class? behave similarly to the above but with `borrowed` management strategy.
- `unmanaged`, `unmanaged` class, `unmanaged` class? behave similarly to the above but with `unmanaged` management strategy.

### 24.3.2 Generic Classes and Records

The remainder of this section §24.3 specifies generic class and record types that are not built-in types (§24.3.1).

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.
24.3.3 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 from either a class or record instance or from the instantiated class or record type itself. 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 initializer (§24.3.8) for its class or record. This makes the compiler-generated initializer generic. The type alias also becomes an argument with intent type to the type constructor (§24.3.6). 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 (§24.3.9) or compiler-generated initializer 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.

```chpl
class Node {
  type eltType;
  var data: eltType;
  var next: unmanaged 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.

24.3.4 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 field can be accessed from a class or record instance or from the instantiated class or record type itself.

The parameter becomes an argument with intent param to the compiler-generated initializer (§24.3.8) for that class or record. This makes the compiler-generated initializer generic. The parameter also becomes an argument with intent param to the type constructor (§24.3.6). 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 (§24.3.9) or compiler-generated initializer 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.

```chpl
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.

### 24.3.5 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 default intent to the compiler-generated initializer (§24.3.8). That argument has no specified type and no default value. This makes the compiler-generated initializer generic. The field also becomes an argument with type intent and no default to the type constructor (§24.3.6). Correspondingly, an actual value must always be passed to the default initializer argument and an actual type to the type constructor argument.

The class or record is instantiated by binding the type of the field to the type of the value passed to the corresponding argument of a user-defined (§24.3.9) or compiler-generated initializer (§24.3.8). 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.

```chapel
class Node {
    var data;
    var next: unmanaged Node(data.type)? = nil;
}
```

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

```chapel
var list = new unmanaged Node(1);
list.next = new unmanaged 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)`.

### 24.3.6 The Type Constructor

A type constructor is automatically created for each class or record. A type constructor is a type function (§13.7.5) 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 in 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 initializer, 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 (§24.3.3, §24.3.4, §24.3.5). Such an instantiated type must be used as the
Generics

24.3.7 Generic Methods

All methods bound to generic classes or records, including initializers, are generic over the implicit this argument. This is in addition to being generic over any other argument that is generic.

24.3.8 The Compiler-Generated Initializer

If no user-defined initializers are supplied for a given generic class, the compiler generates one following in a manner similar to that for concrete classes (§17.4.2). However, the compiler-generated initializer for a generic class or record (§17.4.2) 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 has intent param 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 default 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 initializer are the same as for the type constructor (§24.3.6). 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.

24.3.9 User-Defined Initializers

If a generic field of a class or record does not have a default value or type alias, each user-defined initializer for that class must explicitly initialize that field.

Example (initializersForGenericFields.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 init(c1, v1, type t1, param p1) {
  this.t1 = t1;
  this.p1 = p1;
  this.c1 = c1;
  this.v1 = v1;
  // compiler inserts initialization for remaining fields
}

proc init(type t5, param p5, c5, v5, x5, type t1, param p1, c1, v1, x1) {
  this.t1 = t1;
  this.p1 = p1;
  this.c1 = c1;
  this.v1 = v1;
  this.x1 = x1;
  this.t5 = t5;
  this.p5 = p5;
  this.c5 = c5;
  this.v5 = v5;
  this.x5 = x5;
}

} // 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 initializers are required to initialize fields t1, p1, c1, and v1. Otherwise, field initializations may be omitted according to previously-described initializer semantics.
24.4 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`.

Example (compilerDiagnostics.chpl). The following code shows an example of using user-defined compiler diagnostics to generate warnings and errors:

```chapel
proc foo(x, y) {
    if (x.type != y.type) then
        compilerError("foo() called with non-matching types: ",
            x.type: string, " != ", y.type: string);
    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 whenever 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:

```plaintext
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
```

24.5 Creating General and Specialized Versions of a Function

The Chapel language facility supports three mechanisms for using generic functions along with concrete functions. These mechanisms allow users to create a general generic implementation and also a special implementation for specific concrete types.

The first mechanism applies to functions. According to the function resolution rules described in §13.13, functions accepting concrete arguments are selected in preference to those with a totally generic argument. So, creating a second version of a generic function that declares a concrete type will cause the concrete function to be used where possible:
Example (specializeGenericFunction.chpl).

```chpl
proc foo(x) {
    writeln("in generic foo(x)"),
}
proc foo(x:int) {
    writeln("in specific foo(x:int)"),
}
var myReal: real;
foo(myReal); // outputs "in generic foo(x)"
var myInt: int;
foo(myInt); // outputs "in specific foo(x:int)"
```

This program will run the generic foo function if the argument is a real, but it runs the specific version for int if the argument is an int.

The second mechanism applies when working with methods on generic types. When declaring a secondary method, the receiver type can be a parenthesized expression. In that case, the compiler will evaluate the parenthesized expression at compile time in order to find the concrete receiver type. Then, the resolution rules described above will cause the concrete method to be selected when applicable. For example:

Example (specializeGenericMethod.chpl).

```chpl
record MyNode {
    var field; // since no type is specified here, MyNode is a generic type
}
proc MyNode.foo() {
    writeln("in generic MyNode.foo()");
}
proc (MyNode(int)).foo() {
    writeln("in specific MyNode(int).foo()");
}

var myRealNode = new MyNode(1.0);
myRealNode.foo(); // outputs "in generic MyNode.foo()"
var myIntNode = new MyNode(1);
myIntNode.foo(); // outputs "in specific MyNode(int).foo()"
```

The third mechanism is to use a where clause. Where clauses limit a generic method to particular cases. Unlike the previous two cases, a where clause can be used to declare special implementation of a function that works with some set of types - in other words, the special implementation can still be a generic function. See also §13.10.

### 24.6 Example: A Generic Stack

Example (genericStack.chpl).

```chpl
class MyNode {
    type itemType; // type of item
    var item: itemType; // item in node
    var next: unmanaged MyNode(itemType)?; // reference to next node (same type)
}
record Stack {
    type itemType; // type of items
    var top: unmanaged MyNode(itemType)?; // top node on stack linked list

    proc push(item: itemType) {
        top = new unmanaged MyNode(itemType, item, top);
    }

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

    proc isEmpty return top == nil;
}
25 Input and Output

25.1 See Library Documentation

Chapel includes an extensive library for input and output that is documented in the standard library documentation. See

https://chapel-lang.org/docs/modules/standard/IO.html

and

https://chapel-lang.org/docs/builtins/ChapelIO.html
26 Task Parallelism and Synchronization

Chapel supports both task parallelism and data parallelism. This chapter details task parallelism as follows:

- §26.1 introduces tasks and task parallelism.
- §26.2 describes the begin statement, an unstructured way to introduce concurrency into a program.
- §26.3 describes synchronization variables, an unstructured mechanism for synchronizing tasks.
- §26.4 describes atomic variables, a mechanism for supporting atomic operations.
- §26.5 describes the cobegin statement, a structured way to introduce concurrency into a program.
- §26.6 describes the coforall loop, another structured way to introduce concurrency into a program.
- §26.7 specifies how variables from outer scopes are handled within begin, cobegin and coforall statements.
- §26.8 describes the sync statement, a structured way to control parallelism.
- §26.9 describes the serial statement, a structured way to suppress parallelism.
- §26.10 describes the atomic statement, a construct to support atomic transactions.

26.1 Tasks and Task Parallelism

A Chapel task is a distinct context of execution that may be 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 these tasks have completed. In addition, Chapel provides two constructs, the sync and serial statements, to insert synchronization and suppress parallelism. All four of these constructs can be implemented through judicious uses of the unstructured task-parallel constructs described in the previous paragraph.

Tasks are considered to be created when execution reaches the start of a begin, cobegin, or coforall statement. When the tasks are actually executed depends on the Chapel implementation and run-time execution state.

A task is represented as a call to a task function, whose body contains the Chapel code for the task. Variables defined in outer scopes are considered to be passed into a task function by default intent, unless a different task intent is specified explicitly by a task-intent-clause.

Accesses to the same variable from different tasks are subject to the Memory Consistency Model (§31). Such accesses can result from aliasing due to ref argument intents or task intents, among others.
26.2 The Begin Statement

The begin statement creates a task to execute a statement. The syntax for the begin statement is given by

\[
\text{begin} \quad \text{statement} \\
\text{begin task-intent-clause}_{\text{opt}} \quad \text{statement}
\]

Control continues concurrently with the statement following the begin statement.

Example (\texttt{beginUnordered.chpl}). The code

\begin{verbatim}
begin writeln("output from spawned task"); writeln("output from main task");
\end{verbatim}

executes two \texttt{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 \texttt{writeln} statement within it. However, execution will continue immediately after task creation with the next statement.

A begin statement creates a single task function, whose body is the body of the begin statement. The handling of the outer variables within the task function and the role of \texttt{task-intent-clause} are defined in §26.7.

Yield and return statements are not allowed in begin blocks. Break and continue statements may not be used to exit a begin block.

26.3 Synchronization Variables

Synchronization variables have a logical state associated with the value. The state of the variable is either \textit{full} or \textit{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, except that a single variable may only be written once. Consequently, when a sync variable is read, its state transitions to empty, whereas when a single variable is read, its state does not change. When either type of synchronization variable is written, its state transitions to full.

\texttt{sync} and \texttt{single} are type qualifiers and precede the type of the variable’s value in the declaration. Sync and single are supported for all Chapel primitive types (§7.1) except complex. They are also supported for enumerated types (§7.2) and variables of class type (§17.1.2). For sync variables of class type, the full/empty state applies to the reference to the class object, not to its member fields.

\textit{Rationale.} It is only well-formed to apply full-empty semantics to types that have no more than a single logical value. Booleans, integers, real and imaginary numbers, enums, and class references all meet this criteria. Since it is possible to read/write the individual elements of a complex value, it’s not obvious how the full-empty semantics would interact with such operations. While one could argue that record types with a single field could also be included, the user can more directly express such cases by declaring the field itself to be of sync type.
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 if it is a sync variable; all tasks are selected to proceed if it is a single variable.

A synchronization variable is specified with a sync or single type given by the following syntax:

```
sync-type:
sync type-expression

single-type:
single type-expression
```

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

```chapel
class Tree {
  var isLeaf: bool;
  var left, right: unmanaged 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. If all updates to an initialized sync variable are via compound assignment operators (or equivalently, traditional assignments that read and write the variable once), the full/empty state of the sync variable guarantees that the reads and writes will be interleaved in a manner that makes the updates atomic. For example, the code:

```chapel
var count$: sync int = 0;
 cobegin {
  count$ += 1;
  count$ += 1;
  count$ += 1;
}
```

creates three tasks that increment `count$`. If `count$` were not a sync variable, this code would be unsafe because two tasks could then read the same value before either had written its updated value, causing one of the increments to be lost.
Example (singleVar.chpl). The following code implements a simple split-phase barrier using a single variable.

```chapel
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 with a default intent either has a synchronization type or the formal is generic (§24.1.5) and the actual has a synchronization type, the actual must be an lvalue and is passed by reference. In these cases the formal itself is an lvalue, too. The actual argument is not read or written during argument passing; its state is not changed or waited on. The qualifier `sync` or `single` without the value type can be used to specify a generic formal argument that requires a `sync` or `single` actual.

When the actual argument is a `sync` or `single` and the corresponding formal has the actual’s base type or is implicitly converted from that type, a normal read of the actual is performed when the call is made, and the read value is passed to the formal.

### 26.3.1 Predefined Single and Sync Methods

The following methods are defined for variables of sync and single type.

```chapel
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. This method implements the normal read of a `sync` variable.

```chapel
proc (sync t).readFF(): t
```

```chapel
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. This method implements the normal read of a `single` variable.
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. This method implements the normal write of a sync or single variable.

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 \( \text{true} \) if the sync or single variable is full and \( \text{false} \) otherwise. This method is non-blocking and the state of the sync or single variable is unchanged when this method completes.

Note that \texttt{writeEF} and \texttt{readFE}/\texttt{readFF} methods (for \texttt{sync} and \texttt{single} variables, respectively) are implicitly invoked for normal writes and reads of synchronization variables.

**Example (syncMethods.chpl).** Given the following declarations

```chapl
var x$: sync int;
var y$: single int;
var z: int;
```

the code

```chapl
\texttt{x$} = 5;
\texttt{y$} = 6;
\texttt{z} = \texttt{x$} + \texttt{y$};
```

is equivalent to

```chapl
\texttt{x$.writeEF(5)};
\texttt{y$.writeEF(6)};
\texttt{z} = \texttt{x$.readFE()} + \texttt{y$.readFF()};
```
26.4 Atomic Variables

Atomic variables are variables that support atomic operations. Chapel currently supports atomic operations for bools, all supported sizes of signed and unsigned integers, as well as all supported sizes of reals.

Rationale. The choice of supported atomic variable types as well as the atomic operations was strongly influenced by the C11 standard.

Atomic is a type qualifier that precedes the variable’s type in the declaration. Atomic operations are supported for bools, and all sizes of ints, uints, and reals.

An atomic variable is specified with an atomic type given by the following syntax:

```
atomic-type:
  atomic type-expression
```

26.4.1 Predefined Atomic Methods

The following methods are defined for variables of atomic type. Note that not all operations are supported for all atomic types. The supported types are listed for each method.

Most of the predefined atomic methods accept an optional argument named order of type memoryOrder. The order argument is used to specify the ordering constraints of atomic operations. The supported memoryOrder values are:

- memoryOrder.relaxed
- memoryOrder.acquire
- memoryOrder.release
- memoryOrder.acqRel
- memoryOrder.seqCst

Unless specified, the default for the memoryOrder parameter is memoryOrder.seqCst.

Implementors’ note. Not all architectures or implementations may support all memoryOrder values. In these cases, the implementation should default to a more conservative ordering than specified.

```
proc (atomic T).read(order:memoryOrder = memoryOrder.seqCst): T

Reads and returns the stored value. Defined for all atomic types.
```

```
proc (atomic T).write(v: T, order:memoryOrder = memoryOrder.seqCst)

Stores v as the new value. Defined for all atomic types.
```

```
proc (atomic T).exchange(v: T, order:memoryOrder = memoryOrder.seqCst): T
```
Stores $v$ as the new value and returns the original value. Defined for all atomic types.

\[
\text{proc (atomic } T).\text{compareAndSwap(e: } t, \text{ v: } T, \text{ order:memoryOrder = memoryOrder.seqCst): bool}
\]

Stores $v$ as the new value, if and only if the original value is equal to $e$. Returns $\text{true}$ if $v$ was stored, $\text{false}$ otherwise. Defined for all atomic types.

\[
\text{proc (atomic } T).\text{add(v: } T, \text{ order:memoryOrder = memoryOrder.seqCst)}
\]
\[
\text{proc (atomic } T).\text{sub(v: } T, \text{ order:memoryOrder = memoryOrder.seqCst)}
\]
\[
\text{proc (atomic } T).\text{or(v: } T, \text{ order:memoryOrder = memoryOrder.seqCst)}
\]
\[
\text{proc (atomic } T).\text{xor(v: } T, \text{ order:memoryOrder = memoryOrder.seqCst)}
\]

Applies the appropriate operator ($+,-,|,\&,\sim$) to the original value and $v$ and stores the result. All of the methods are defined for integral atomic types. Only add and sub ($+,-$) are defined for real atomic types. None of the methods are defined for the $\text{bool}$ atomic type.

\[
\text{proc (atomic } T).\text{fetchAdd(v: } T, \text{ order:memoryOrder = memoryOrder.seqCst): } T
\]
\[
\text{proc (atomic } T).\text{fetchSub(v: } T, \text{ order:memoryOrder = memoryOrder.seqCst): } T
\]
\[
\text{proc (atomic } T).\text{fetchOr(v: } T, \text{ order:memoryOrder = memoryOrder.seqCst): } T
\]
\[
\text{proc (atomic } T).\text{fetchXor(v: } T, \text{ order:memoryOrder = memoryOrder.seqCst): } T
\]

Applies the appropriate operator ($+,-,|,\&,\sim$) to the original value and $v$, stores the result, and returns the original value. All of the methods are defined for integral atomic types. Only add and sub ($+,-$) are defined for real atomic types. None of the methods are defined for the $\text{bool}$ atomic type.

\[
\text{proc (atomic bool).testAndSet(order:memoryOrder = memoryOrder.seqCst): bool}
\]

Stores $\text{true}$ as the new value and returns the old value. Equivalent to $\text{exchange(true)}$. Only defined for the $\text{bool}$ atomic type.

\[
\text{proc (atomic bool).clear(order:memoryOrder = memoryOrder.seqCst)}
\]

Stores $\text{false}$ as the new value. Equivalent to $\text{write(false)}$. Only defined for the $\text{bool}$ atomic type.

\[
\text{proc (atomic } T).\text{waitFor(v: } T)
\]

Waits until the stored value is equal to $v$. The implementation may yield the running task while waiting. Defined for all atomic types.
26.5 The Cobegin Statement

The cobegin statement is used to introduce concurrency within a block. The cobegin statement syntax is

\[
\text{cobegin} \quad \text{statement}
\]

A new task and a corresponding task function are created for each statement in the block statement. Control continues when all of the tasks have finished. The handling of the outer variables within each task function and the role of task-intent-clause are defined in §26.7.

Return statements are not allowed in cobegin blocks. Yield statement may only be lexically enclosed in cobegin blocks in parallel iterators (§23.4). Break and continue statements may not be used to exit a cobegin block.

Example (cobeginAndEquivalent.chpl). The cobegin statement

\[
\text{cobegin} \{
\text{stmt1}();
\text{stmt2}();
\text{stmt3}();
\}
\]

is equivalent to the following code that uses only begin statements and single variables to introduce concurrency and synchronize:

\[
\text{var } s1$, s2$, s3$: \text{single bool} ;
\text{begin} \{ \text{stmt1}(); s1$ = \text{true} ; \}
\text{begin} \{ \text{stmt2}(); s2$ = \text{true} ; \}
\text{begin} \{ \text{stmt3}(); s3$ = \text{true} ; \}
s1$; s2$; s3$;
\]

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.

26.6 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

\[
\text{coforall} \quad \text{statement}:
\]

\[
\text{coforall} \quad \text{index–var–declaration in iterable–expression task–intent–clause$_{opt}$ do statement}
\]

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.

The single task function created for a coforall and invoked by each task contains the loop body. The handling of the outer variables within the task function and the role of task-intent-clause are defined in §26.7.

Return statements are not allowed in coforall blocks. Yield statement may only be lexically enclosed in coforall blocks in parallel iterators (§23.4). Break and continue statements may not be used to exit a coforall block.
Example (coforallAndEquivalent.chpl). The coforall statement

```chapel
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:

```chapel
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.

### 26.7 Task Intents

If a variable is referenced within the lexical scope of a `begin`, `cobegin`, or `coforall` statement and is declared outside that statement, it is subject to task intents. That is, it is considered to be passed as an actual argument to the corresponding task function at task creation time. All references to the variable within the task function implicitly refer to a shadow variable, i.e. the task function’s corresponding formal argument.

When the task construct is inside a method on a record and accesses a field of `this`, the field is treated as a regular variable. That is, it is passed as an actual argument to the task function and all references to the field within the task function implicitly refer to the corresponding shadow variable.

Each formal argument of a task function has the default argument intent by default. For variables of primitive and class types, this has the effect of capturing the value of the variable at task creation time and referencing that value instead of the original variable within the lexical scope of the task construct.

A formal can be given another argument intent explicitly by listing it with that intent in the optional `task-intent-clause`. For example, for variables of most types, the `ref` intent allows the task construct to modify the corresponding original variable or to read its updated value after concurrent modifications.

The syntax of the task intent clause is:

```chapel
  task-intent-clause:
    with ( task-intent-list )

  task-intent-list:
```
where the following intents can be used as a \texttt{formal-intent}: \texttt{ref, in, const, const in, const ref, task-private-var-decl} is defined in \S27.4. In addition, \texttt{task-intent-item} may define a \texttt{reduce} intent. Reduce intents are described in the \texttt{Reduce Intents} technical note in the online documentation:

https://chapel-lang.org/docs/technotes/reduceIntents.html

The implicit treatment of outer scope variables as the task function’s formal arguments applies to both module level and local variables. It applies to variable references within the lexical scope of a task construct, but does not extend to its dynamic scope, i.e., to the functions called from the task(s) but declared outside of the lexical scope. The loop index variables of a \texttt{coforall} statement are not subject to such treatment within that statement; however, they are subject to such treatment within nested task constructs, if any.

\textit{Rationale.} The primary motivation for task intents is to avoid some races on scalar/record variables, which are possible when one task modifies a variable and another task reads it. Without task intents, for example, it would be easy to introduce and overlook a bug illustrated by this simplified example:

\begin{verbatim}
{ 
  var i = 0;
  while i < 10 {
    begin {
      f(i);
    }
    i += 1;
  }
}
\end{verbatim}

If all the tasks created by the \texttt{begin} statement start executing only after the \texttt{while} loop completes, and \texttt{i} within the \texttt{begin} is treated as a reference to the original \texttt{i}, there will be ten tasks executing \texttt{f(10)}. However, the user most likely intended to generate ten tasks executing \texttt{f(0), f(1), ..., f(9)}. Task intents ensure that, regardless of the timing of task execution.

Another motivation for task intents is that referring to a captured copy in a task is often more efficient than referring to the original variable. That’s because the copy is a local constant, e.g. it could be placed in a register when it fits. Without task intents, references to the original variable would need to be implemented using a pointer dereference. This is less efficient and can hinder optimizations in the surrounding code, for example loop-invariant code motion.

Furthermore, in the above example the scope where \texttt{i} is declared may exit before all the ten tasks complete. Without task intents, the user would have to protect \texttt{i} to make sure its lexical scope doesn’t exit before the tasks referencing it complete.

We decided to treat \texttt{cobegin} and \texttt{coforall} statements the same way as \texttt{begin}. This is for consistency and to make the race-avoidance benefit available to more code.

We decided to apply task intents to module level variables, in addition to local variables. Again, this is for consistency. One could view module level variables differently than local variables (e.g. a module level variable is “always available”), but we favored consistency over such an approach.

We decided not to apply task intents to “closure” variables, i.e., the variables in the dynamic scope of a task construct. This is to keep this feature manageable, so that all variables subject to task intents can be
obtained by examining just the lexical scope of the task construct. In general, the set of closure variables can be hard to determine, unwieldy to implement and reason about, it is unclear what to do with extern functions, etc.

We do not provide `inout` or `out` as task intents because they will necessarily create a data race in a `cobegin` or `coforall`. `type` and `param` intents are not available either as they do not seem useful as task intents.

*Future.* For a given intent, we would also like to provide a blanket clause, which would apply the intent to all variables. An example of syntax for a blanket `ref` intent would be `ref *`.

### 26.8 The Sync Statement

The sync statement acts as a join of all dynamically encountered `begin` from within a statement. The syntax for the sync statement is given by

\[
\text{sync\textendash statement:} \\
\text{sync statement} \\
\text{sync block\textendash statement}
\]

Return statements are not allowed in sync statement blocks. Yield statement may only be lexically enclosed in sync statement blocks in parallel iterators (§23.4). 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.

```chapel
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

```chapel
sync {
    begin stmt1();
    begin stmt2();
}
```

is similar to the following cobegin statement

```chapel
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.
26.9 The Serial Statement

The serial statement can be used to dynamically disable parallelism. The syntax is:

\[
\text{serial-statement:} \\
\quad \text{serial expression}_{\text{opt}} \ 	ext{do statement} \\
\quad \text{serial expression}_{\text{opt}} \ 	ext{block-statement}
\]

where the optional expression evaluates to a boolean value. If the expression is omitted, it is as though ‘true’ were specified. Whatever the expression’s value, the statement following it is evaluated. If the expression is true, any dynamically encountered code that would normally create new tasks within the statement is instead executed by the original task without creating any new ones. In effect, execution is serialized. If the expression is false, code within the statement will generate tasks according to normal Chapel rules.

Example (serialStmt1.chpl). In the code

```chapel
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

```chapel
serial {
  begin stmt1();
  cobegin {
    stmt2();
    stmt3();
  }
  coforall i in 1..n do stmt4();
}
```

is equivalent to

```chapel
stmt1();
{
  stmt2();
  stmt3();
}
for i in 1..n do stmt4();
```

because the expression evaluated to determine whether to serialize always evaluates to true.
26.10 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:

```plaintext
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.
27 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:

- §27.1 describes the forall statement.
- §27.2 describes forall expressions
- §27.3 specifies how variables from outer scopes are handled within forall statements and expressions.
- §27.5 describes promotion.
- §27.6 describes reductions and scans.
- §27.7 describes the configuration constants for controlling default data parallelism.

Data-parallel constructs may result in accesses to the same variable from different tasks, possibly due to aliasing using ref argument intents or forall intents, among others. Such accesses are subject to the Memory Consistency Model (§31).

27.1 The Forall Statement

The forall statement is a concurrent variant of the for statement described in §11.9.

27.1.1 Syntax

The syntax of the forall statement is given by

```
forall-statement:
  forall index-var-declaration in iterable-expression task-intent-clause_opt do statement
  forall index-var-declaration in iterable-expression task-intent-clause_opt block-statement
  forall iterable-expression task-intent-clause_opt do statement
  forall iterable-expression task-intent-clause_opt block-statement
  [ index-var-declaration in iterable-expression task-intent-clause_opt ] statement
  [ iterable-expression task-intent-clause_opt ] 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 will resort to serial iteration when iterable-expression does not support parallel iteration. The forall form will result in an error when parallel iteration is not available.

The handling of the outer variables within the forall statement and the role of task-intent-clause are defined in §27.3.
27.1.2 Execution and Serializability

The forall statement evaluates the loop body once for each element yielded by the iterable-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 §23.4.

This differs from the semantics of the coforall loop, discussed in §26.6, where each iteration is guaranteed to run using a distinct task. The coforall loop thus has potentially higher overhead than a forall loop with the same number of iterations, but in cases where concurrency is required for correctness, it is essential.

In practice, the number of tasks that will be used to evaluate a forall loop is determined by the object or iterator that is leading the execution of the loop, as is the mapping of iterations to tasks.

This concept will be formalized in future drafts of the Chapel specification. For now, the primer on parallel iterators in the online documentation provides a brief introduction:

https://chapel-lang.org/docs/primers/parIters.html

Please also refer to User-Defined Parallel Zippered Iterators in Chapel, published in the PGAS 2011 workshop.

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.

A return statement may not be lexically enclosed in a forall statement. A yield statement may only be lexically enclosed in a forall statement that is within a parallel iterator §23.4. A break statement may not be used to exit a forall statement. A continue statement skips the rest of the current iteration of the forall loop.

Example (forallStmt.chpl). In the code

```chapel
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

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

27.1.3 Zipper Iteration

Zipper iteration has the same semantics as described in §11.9.1 and §23.4 for parallel iteration.
27.2 The Forall Expression

The forall expression is a concurrent variant of the for expression described in §10.22.

27.2.1 Syntax

The syntax of a forall expression is given by

```
forall expression
forall index var-declaration in iterable-expression task-intent-clause_opt do expression
forall iterable-expression task-intent-clause_opt do expression
[ index var-declaration in iterable-expression task-intent-clause_opt ] expression
[ iterable-expression task-intent-clause_opt ] 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.

As with the forall statement, the square bracketed form will resort to serial iteration when `iteratable-expression` does not support parallel iteration. The `forall` form will result in an error when parallel iteration is not available.

The handling of the outer variables within the forall expression and the role of `task-intent-clause` are defined in §27.3.

27.2.2 Execution

A forall expression is an iterator that executes a forall loop (§27.1), evaluates the body expression on each iteration of the loop, and yields each resulting value.

When a forall expression is used to initialize a variable, such as

```
var X = forall iterableExpression() do computeValue();
```

the variable will be inferred to have an array type. The array’s domain is defined by the `iterable-expression` following the same rules as for promotion, both in the regular case §27.5 and in the zipper case §27.5.2.

```
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 §27.1.2.

27.2.3 Zipper Iteration

Forall expression also support zippered iteration semantics as described in §11.9.1 and §23.4 for parallel iteration.
27.2.4 Filtering Predicates in Forall Expressions

A filtering predicate is an if expression that is immediately enclosed by a forall expression and does not have 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.

When a forall expression with a filtering predicate is captured into a variable, the resulting array has a 1-based one-dimensional domain.

Example (forallFilter.chpl). The following expression returns every other element starting with the first:

\[
\text{[i in } 1..\text{s.numElements}] \quad \text{if } \quad i \% 2 == 1 \quad \text{then} \quad s(i)
\]

27.3 Forall Intents

If a variable is referenced within the lexical scope of a forall statement or expression and is declared outside that forall construct, it is subject to forall intents, analogously to task intents (§26.7) for task-parallel constructs. That is, the variable is considered to be passed as an actual argument to each task function created by the object or iterator leading the execution of the loop. If no tasks are created, it is considered to be an actual argument to the leader or standalone iterator itself. All references to the variable within the forall construct implicitly refer to a shadow variable, i.e. the corresponding formal argument of the task function or the leader/standalone iterator.

When the forall construct is inside a method on a record and accesses a field of this, the field is treated as a regular variable. That is, it is subject to forall intents and all references to this field within the forall construct implicitly refer to the corresponding shadow variable.

Each formal argument of a task function or iterator has the default intent by default. For variables of primitive, enum, and class types, this has the effect of capturing the value of the variable at task creation time. Within the lexical scope of the forall construct, the variable name references the captured value instead of the original value.

A formal can be given another intent explicitly by listing it with that intent in the optional task-intent-clause. For example, for variables of most types, the ref intent allows the body of the forall loop to modify the corresponding original variable or to read its updated value after concurrent modifications. The in intent is an alternative way to obtain task-private variables (§27.4). A reduce intent can be used to reduce values across iterations of a forall or coforall loop. Reduce intents are described in the Reduce Intents technical note in the online documentation:

https://chapel-lang.org/docs/technotes/reduceIntents.html

Rationale. A forall statement or expression may create tasks in its implementation. Forall intents affect those tasks in the same way that task intents (§26.7) affect the behavior of a task construct such as a coforall loop.
27.4 Task-Private Variables

A task-private variable declared in a forall loop results in a separate shadow variable in each task created by the forall loop’s parallel iterator, as well as a "top-level" shadow variable created at the top level of the parallel iterator itself. In contrast to regular forall intents §27.3 these shadow variables are unrelated to outer variables of the same name, if any.

A given shadow variable is created at the start and destroyed at the end of its task. Within the lexical scope of the body of the forall statement or expression, the variable name refers to the shadow variable created in the task that executed the current yield statement.

The "top-level" shadow variable is created at the start and destroyed at the end of the parallel iterator. It is referenced in those iterations of the forall loop that are due to "top-level" yields, i.e. yields that are outside any of the task constructs that the iterator may have.

The syntax of a task-private variable declaration in a forall statement’s with-clause is:

\[
\text{task-private-var-decl:} \\
\text{task-private-var-kind identifier type-part opt initialization-part opt} \\
\text{task-private-var-kind:} \\
\text{const} \\
\text{var} \\
\text{ref}
\]

The declaration of a const or var task-private variable must have at least one of type-part and initialization-part. A ref task-private variable must have initialization-part and cannot have type-part. A ref shadow variable is a reference to the initialization-part as calculated at the start of the corresponding task or the iterator. ref shadow variables are never destroyed.

Cray’s Chapel Implementation. Currently task-private variables are not available for task constructs. A regular variable declared at the start of the begin/cobegin/coforall block can be used instead.

Example (task-private-variable.chpl). In the following example, the writeln() statement will observe the first shadow variable 4 times: twice each for the yields "before coforall" and "after coforall". An additional shadow variable will be created and observed twice for each of the three coforall tasks.

```chapel
var cnt: atomic int; // count our shadow variables
record R { var id = cnt.fetchAdd(1); }
iter myIter() { yield ""; } // serial iterator, unused
iter myIter(param tag) where tag == iterKind.standalone {
  for 1..2 do
    yield "before coforall"; // shadow var 0 ("top-level")
    coforall 1..3 do
      for 1..2 do
        yield "inside coforall"; // shadow vars 1..3
        for 1..2 do
          yield "after coforall"; // shadow var 0, again
    }
}
forall str in myIter()
```
with (var tpv: R)  // declare a task-private variable
do
  writeln("shadow var: ", tpv.id, " yield: ", str);

27.5 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 §13.13.

Functions that can be promoted include procedures, operators, casts, and methods. Also note that since class and record field access is performed with getter methods (§17.5.1), field access can also be promoted.

If the original function returns a value or a reference, the corresponding promoted expression is an iterator yielding each computed value or reference.

When a promoted expression is used to initialize a variable, such as `var X = A.x;` in the above example, the variable’s type will be inferred to be an array. The array’s domain is defined by the expression that causes promotion:

<table>
<thead>
<tr>
<th>input expression</th>
<th>resulting array’s domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>array domain</td>
<td>that array’s domain</td>
</tr>
<tr>
<td>range</td>
<td>that domain</td>
</tr>
<tr>
<td>iterator</td>
<td>one-dimensional domain</td>
</tr>
<tr>
<td></td>
<td>1-based one-dimensional</td>
</tr>
</tbody>
</table>

Future. We would like to allow the iterator author to specify the shape of the iterator, i.e. the domain of the array that would capture the result of the corresponding promoted expression, such as

```chapel
var myArray = myScalarFunction(myIterator());
```

This will be helpful, for example, when the iterator yields one value per an array or domain element that it iterates over internally.

Example (promotion.chpl). Given the array

```chapel
var A: [1..5] int = [i in 1..5] i;
```

and the function

```chapel
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.

Example (field-promotion.chpl). Given an array of points, such as `A` defined below:
Data Parallelism

27.5.1 Default Arguments

When a call is promoted and that call relied upon default arguments (§13.4.2), the default argument expression can be evaluated many times. For example:

\[ \text{Example (promotes-default.chpl).} \]

\[
\text{var}\ \text{counter: atomic int;}
\]
\[
\text{proc}\ \text{nextCounterValue():int} \{\n  \text{var}\ i = \text{counter.fetchAdd}(1);\n  \text{return}\ i;\n\}\n\]
\[
\text{proc}\ \text{assignCounter(ref}\ \text{x:int,}\ \text{counter=nextCounterValue())} \{\n  x = \text{counter};\n\}\n\]

Here the function assignCounter has a default argument providing the next value from an atomic counter as the value to set.

\[
\text{var}\ A: [1..5] \text{int};\n\text{assignCounter(A);}\n\]

The assignCounter call uses both the default argument for counter as well as promotion. When these features are combined, the default argument will be evaluated once per promoted element. As a result, after this command, A will contain the elements 0 1 2 3 4 in some order.

27.5.2 Zipper Promotion

Promotion also supports zippered iteration semantics as described in §11.9.1 and §23.4 for parallel iteration.

Consider a function \( f \) with formal arguments \( s_1, s_2, \ldots \) that are promoted and formal arguments \( a_1, a_2, \ldots \) that are not promoted. The call

\[ f(s_1, s_2, \ldots, a_1, a_2, \ldots) \]

is equivalent to

\[ [(e_1, e_2, \ldots) \text{ in } \text{zip}(s_1, s_2, \ldots)] \ f(e_1, e_2, \ldots, a_1, a_2, \ldots) \]
Data Parallelism

The usual constraints of zipper iteration apply to zipper promotion so the promoted actuals must have the same shape.

A zipper promotion can be captured in a variable, such as var X = f(s1, s2, ..., a1, a2, ...); using the above example. If so, the domain of the resulting array is defined by the first argument that causes promotion. The rules are the same as in the non-zipper case.

Example (zipper-promotion.chpl). Given a function defined as

```chapel
proc foo(i: int, j: int) {
    return (i, j);
}
```

and a call to this function written

```chapel
writeln(foo(1..3, 4..6));
```

then the output is

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

27.5.3 Whole Array Operations and Evaluation Order

Whole array operations are a form of promotion as applied to operators rather than functions.

Whole array assignment is one example. It is implicitly parallel. The array assignment statement:

```chapel
LHS = RHS;
```

is equivalent to

```chapel
forall (e1, e2) in zip(LHS, RHS) do
    e1 = e2;
```

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:

```chapel
```

and

```chapel
```

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.
27.6 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 30).

27.6.1 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:

\[
\text{reduce-expression:} \\
\quad \text{reduce-scan-operator reduce iterable-expression} \\
\quad \text{class-type reduce iterable-expression} \\
\quad \text{reduce-scan-operator: one of} \\
\quad \quad + * && || & | ^ \min \max \minloc \maxloc
\]

Chapel’s predefined reduction operators are defined by \textit{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). When a minimum, maximum, minimum-with-location, or maximum-with-location reduction encounters a NaN, the result is a NaN.

The expression on the right-hand side of the \textit{reduce} keyword can be of any type that can be iterated over, provided the reduction operator can be applied to the values yielded by the iteration. 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.

For the minimum-with-location and maximum-with-location reductions, the argument on the right-hand side of the \textit{reduce} keyword must be a 2-tuple. Its first component is the collection of values for which the minimum/maximum value is to be computed. The second argument component is a collection of indices with the same size and shape that provides names for the locations of the values in the first component. The reduction returns a tuple containing the minimum/maximum value in the first argument component and the value at the corresponding location in the second argument component.

Example (\textit{reduce-loc.chpl}). The first line below computes the smallest element in an array \(A\) as well as its index, storing the results in \(\text{minA}\) and \(\text{minALoc}\), respectively. It then computes the largest element in a forall expression making calls to a function \(\text{foo}\()), storing the value and its number in \(\text{maxVal}\) and \(\text{maxValNum}\).

\begin{verbatim}
var (minA, minLoc) = minloc reduce zip(A, A.domain);
var (maxVal, maxValNum) = maxloc reduce zip([i in 1..n] foo(i), 1..n);
\end{verbatim}
User-defined reductions are specified by preceding the keyword `reduce` by the class type that implements the reduction interface as described in §30.

### 27.6.2 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 iterable-expression
  class-type scan iterable-expression
```

The predefined scans are defined by `reduce-scan-operator`. These are identical to the predefined reductions and are described in §27.6.1.

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

```go
var A: [1..3] int = 1;
```

that is initialized such that each element contains one, then the code

```go
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 30.

### 27.7 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:

<table>
<thead>
<tr>
<th>Config Const</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dataParTasksPerLocale</code></td>
<td>int</td>
<td><code>top level .maxTaskPar</code> (see §28.1.2)</td>
</tr>
<tr>
<td><code>dataParIgnoreRunningTasks</code></td>
<td>bool</td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>dataParMinGranularity</code></td>
<td>int</td>
<td><code>1</code></td>
</tr>
</tbody>
</table>
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 module level configuration constants are used to specify their default behavior within each locale.
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.

28.1 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.

28.1.1 Locale Types

The identifier locale is a class type that abstracts a locale as described above. Both data and tasks can be associated with a value of locale type. A Chapel implementation may define subclass(es) of locale for a richer description of the target architecture.

28.1.2 Locale Methods

The locale type supports the following methods:

\[
\text{proc locale.callStackSize: uint(64);} \]

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.

\[
\text{proc locale.id: int;} \]

Returns a unique integer for each locale, from 0 to the number of locales less one.

\[
\text{proc locale.maxTaskPar: int(32);} \]

Returns an estimate of the maximum parallelism available for tasks on a given locale.
proc locale.name: string;

Returns the name of the locale.

proc numPUs(logical: bool = false, accessible: bool = true);

Returns the number of processing unit instances available on a given locale. Basically these are the things
that execute instructions. If logical is false then the count reflects physical instances, often referred to as
cores. Otherwise it reflects logical instances, such as hardware threads on multithreaded CPU architectures. If
accessible is true then the count includes only those processors the OS has made available to the program.
Otherwise it includes all processors that seem to be present.

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.

### 28.1.3 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).

```chapel
config const numLocales: int;
config LocaleSpace: domain(1) = [0..numLocales-1];
config 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 opportu-
nities 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

```chapel
const MyLocales: [0..numLocales*4] locale
  = [loc in 0..numLocales*4] Locales(loc%numLocales);
on MyLocales[1] ...
```

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.
28.1.4 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.

28.1.5 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:

```plaintext
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.

### 28.2 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
non expression block-statement
```

The locale of the expression is automatically queried as described in §28.1.5 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 statements may only be lexically enclosed in on statements in parallel iterators §23.4

### 28.2.1 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
```
29 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 §29.1, domain values §29.2, and arrays §29.3
- domain maps are not retained upon domain assignment §29.4
- standard layouts and distributions, such as Block and Cyclic, are documented under Standard Library in Cray Chapel online documentation here: https://chapel-lang.org/docs/modules/layoutdist.html
- specification of user-defined domain maps is forthcoming; please refer to the Domain Map Standard Interface page under Technical Notes in Cray Chapel online documentation here: https://chapel-lang.org/docs/technotes/dsi.html

29.1 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 (§29.2).

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 \textit{dmap value} in a \textit{dmapped} clause:

\begin{verbatim}
  mapped-domain-type:
    domain-type \textbf{dmapped} dmap-value
\end{verbatim}

\begin{verbatim}
  dmap-value:
    expression
\end{verbatim}

A dmap value consists of an instance of a domain map class wrapped in an instance of the predefined record \textit{dmap}. The domain map class is chosen and instantiated by the user. \textit{dmap} behaves like a generic record with a single generic field, which holds the domain map instance.

\textit{Example}. The code
\begin{verbatim}
  use BlockDist;
  var MyBlockDist: dmap(Block(rank=2));
\end{verbatim}

declares a variable capable of storing dmap values for a two-dimensional Block distribution. The Block distribution is described in more detail in the standard library documentation. See: \url{https://chapel-lang.org/docs/modules/dists/BlockDist.html}

\textit{Example}. The code
\begin{verbatim}
  use BlockDist;
  var MyBlockDist: dmap(Block(rank=2)) = new dmap(new Block({1..5,1..6}));
\end{verbatim}

creates a dmap value wrapping a two-dimensional Block distribution with a bounding box of \((1..5, 1..6)\) over all of the locales.

\textit{Example}. The code
\begin{verbatim}
  use BlockDist;
  var MyBlockDist = new dmap(new Block({1..5,1..6}));
  type MyBlockedDom = domain(2) \textbf{dmapped} MyBlockDist;
\end{verbatim}

defines a two-dimensional rectangular domain type that is mapped using a Block distribution.

The following syntactic sugar is provided within the \textit{dmapped} clause. If a \textit{dmapped} clause starts with the name of a domain map class, it is considered to be an initialization expression as if preceded by \textit{new}. The resulting domain map instance is wrapped in a newly-created instance of \textit{dmap} implicitly.

\textit{Example}. The code
\begin{verbatim}
  use BlockDist;
  type BlockDom = domain(2) \textbf{dmapped} Block({1..5,1..6});
\end{verbatim}

is equivalent to
\begin{verbatim}
  use BlockDist;
  type BlockDom = domain(2) \textbf{dmapped} new dmap{new Block({1..5,1..6})};
\end{verbatim}
29.2 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 (§21.2.1) has a default domain map.

Example. In the following code

```plaintext
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

```plaintext
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.

29.3 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

```plaintext
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.
Domain Maps Are Not Retained upon Domain Assignment

Domain assignment (§21.8.1) 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

```plaintext
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

```plaintext
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.
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:

31 Memory Consistency Model

In this section, we describe Chapel’s memory consistency model. The model is based on sequential consistency for data-race-free programs as adopted by C11, C++11, Java, UPC, and Fortran 2008.

Sequential consistency (SC) means that all Chapel tasks agree on the interleaving of memory operations and this interleaving results in an order is consistent with the order of operations in the program source code. Conflicting memory operations, i.e., operations to the same variable, or memory location, and one of which is a write, form a data race if they are from different Chapel tasks and can be executed concurrently. Accesses to the same variable from different tasks can result from the tasks referencing the same variable directly – or indirectly via aliases. Aliases arise, for example, when using ref variables, argument intents, return intents, task intents and forall intents.

Any Chapel program with a data race is not a valid program, and an implementation cannot be relied upon to produce consistent behavior. Valid Chapel programs will use synchronization constructs such as sync, single, or atomic variables or higher-level constructs based on these to enforce ordering for conflicting memory operations.

The following design principles were used in developing Chapel’s memory consistency model:

1. Sequential programs have program order semantics. Programs that are completely sequential cannot have data races and should appear to execute as though each statement was executed one at a time and in the expected order.

2. Chapel’s fork-join constructs introduce additional order dependencies. Operations within a task cannot behave as though they started before the task started. Similarly, all operations in a task must appear to be completed to a parent task when the parent task joins with that task.

3. Multi-locale programs have the same memory consistency model as single-locale programs. The Chapel language seeks to allow a single description of an algorithm to work with different data distributions. A result of this property is that an expression of a program must be correct whether it is working on local or distributed data.

4. Chapel’s memory model should be as relaxed as possible while still consistent with these design principles. In particular, making all operations sequentially consistent is not likely to enable good performance. At the same time, sequential consistency should be available to programmers when requested.

See A Primer on Memory Consistency and Cache Coherence by Sorin, et al. for more background information on memory consistency models. This chapter will proceed in a manner inspired by the XC memory model described there.

31.1 Sequential Consistency for Data-Race-Free Programs

Sequential consistency for data-race-free programs is described in terms of two orders: program order and memory order. The program order $<_p$ is a partial order describing serial or fork-join parallelism dependencies between variable reads and writes. The memory order $<_m$ is a total order that describes the semantics of synchronizing memory
operations (via atomic, sync or single variables) with sequential consistency. Non-SC atomic operations (described in Section 31.2) do not create this total order.

Note that sync/single variables have memory consistency behavior equivalent to a sequence of SC operations on atomic variables. Thus for the remainder of the chapter, we will primarily focus on operations on atomic variables.

We will use the following notation:

- \( L(a) \) indicates a load from a variable at address \( a \). \( a \) could refer to local or remote memory.
- \( S(a) \) indicates a store to a variable at address \( a \). \( a \) could refer to local or remote memory.
- \( A_{sc}(a) \) indicates an atomic operation on a variable at address \( a \) with sequential consistency. The variable at address \( a \) could refer to local or remote memory. Atomic operations must be completed as a single operation (i.e. atomically), and so it is not possible to observe an intermediate state from an atomic operation under any circumstances.
- \( A_{r}(a, o) \) indicates an atomic operation on a variable at address \( a \) with ordering constraint \( o \), where \( o \) can be one of relaxed, acquire, or release (see Section 31.2). As with \( A_{sc}(a) \), relaxed atomic operations must be completed as a single operation.

- \( L(a), S(a), A_{sc}(a), \) and \( A_{r}(a, o) \) are also called memory operations
- \( X <_{p} Y \) indicates that \( X \) precedes \( Y \) in program order
- \( X <_{m} Y \) indicates that \( X \) precedes \( Y \) in memory order
- \( t = \text{begin}(X) \) starts a new task named \( t \) to execute \( X \)
- \( \text{waitFor}(t_1..t_n) \) waits for tasks \( t_1..t_n \) to complete
- \( \text{on}(L) \) migrates the running task to locale \( L \). Note that while the on statement may change the locale on which the current task is running, it has no impact on the memory consistency requirements.

For the purposes of describing this memory model, it is assumed that Chapel programs will be translated into sequences of memory operations, begin statements, and waitFor statements. The translation of a Chapel program into a sequence of memory operations must preserve sequential program semantics. That is, if we have a snippet of a Chapel program without task operations, such as \( X; Y; \), the statements \( X \) and \( Y \) will be converted into a sequence of load, store, and atomic operations in a manner that preserves the behavior of a this serial portion of the program. That is, \( X = x_1, x_2, ... \) and \( Y = y_1, y_2, ... \) where \( x_i \) and \( y_j \) are each a sequence of load, store, or atomic operations and we have \( x_i <_{p} y_j \).

Likewise, for the purposes of this memory model, Chapel’s parallelism keywords are viewed as a sequence of operations including the primitives of starting a task (\( \text{begin} \)) and waiting for some number of tasks (\( \text{waitFor}(t_1..t_n) \)). In particular:

- forall (including promotion) creates some number of tasks \( m \) to execute the \( n \) iterations of the loop \( (t_i = \text{begin\{some\-loop\-bodies\}} \) for each task \( i = 1..m \)) and waits for them to complete (\( \text{waitFor}(t_1..t_m) \)). The number of tasks \( m \) is defined by the implementation of the parallel iterator (See Section 23 for details on iterators).
- coforall creates one task per loop iteration \( (t_i = \text{begin\{loop\-body\}} \) for all loop iterations \( i = 1..n \)) and then waits for them all to complete (\( \text{waitFor}(t_1..t_n) \)).
- **cobegin** creates one task per enclosed statement \((t_i = \text{begin}\{X_i\})\) for statements \(X_1..X_n\) and then waits for them all to complete \((\text{waitFor}(t_1..t_n))\).

- **begin** creates a task to execute the enclosed statement \((t = \text{begin}\{X\})\). The **sync** statement waits for all tasks \(t_i\) created by a **begin** statement in the dynamic scope of the **sync** statement that are not within other, nested **sync** statements \((\text{waitFor}(t_1..t_n)\) for all \(n\) such tasks).

### 31.1.1 Program Order

Task creation and task waiting create a conceptual tree of program statements. The task bodies, task creation, and task wait operations create a partial order \(<_p\) of program statements. For the purposes of this section, the statements in the body of each Chapel task will be implemented in terms of load, store, and atomic operations.

- If we have a program snippet without tasks, such as \(X; Y;\), where \(X\) and \(Y\) are memory operations, then \(X <_p Y\).
- The program \(X; \text{begin}\{Y\}; Z;\) implies \(X <_p Y\). However, there is no particular relationship between \(Y\) and \(Z\) in program order.
- The program \(t = \text{begin}\{Y\}; \text{waitFor}(t); Z;\) implies \(Y <_p Z\).
- \(X <_p Y\) and \(Y <_p Z\) imply \(X <_p Z\)

### 31.1.2 Memory Order

The memory order \(<_m\) of SC atomic operations in a given task respects program order as follows:

- If \(A_{sc}(a) <_p A_{sc}(b)\) then \(A_{sc}(a) <_m A_{sc}(b)\)

Every SC atomic operation gets its value from the last SC atomic operation before it to the same address in the total order \(<_m\):

- Value of \(A_{sc}(a) = \text{Value of max}_{<_m}(A'_{sc}(a)|A'_{sc}(a) <_m A_{sc}(a))\)

For data-race-free programs, every load gets its value from the last store before it to the same address in the total order \(<_m\):

- Value of \(L(a) = \text{Value of max}_{<_m}(S(a)|S(a) <_m L(a) \text{ or } S(a) <_p L(a))\)

For data-race-free programs, loads and stores are ordered with SC atomics. That is, loads and stores for a given task are in total order \(<_m\) respecting the following rules which preserve the order of loads and stores relative to SC atomic operations:

- If \(L(a) <_p A_{sc}(b)\) then \(L(a) <_m A_{sc}(b)\)
• If $S(a) <_p A_{sc}(b)$ then $S(a) <_m A_{sc}(b)$
• If $A_{sc}(a) <_p L(b)$ then $A_{sc}(a) <_m L(b)$
• If $A_{sc}(a) <_p S(b)$ then $A_{sc}(a) <_m S(b)$

For data-race-free programs, loads and stores preserve sequential program behavior. That is, loads and stores to the same address in a given task are in the order $<_m$ respecting the following rules which preserve sequential program behavior:

• If $L(a) <_p L'(a)$ then $L(a) <_m L'(a)$
• If $L(a) <_p S(a)$ then $L(a) <_m S(a)$
• If $S(a) <_p S'(a)$ then $S(a) <_m S'(a)$

### 31.2 Non-Sequentially Consistent Atomic Operations

Sequential consistency for atomic operations can be a performance bottleneck under some circumstances. Chapel provides non-SC atomic operations to help alleviate such situations. Such uses of atomic operations must be done with care and should generally not be used to synchronize tasks.

Non-SC atomic operations are specified by providing a *memory order* argument to the atomic operations. See Section 26.4.1 for more information on the memory order types.

### 31.2.1 Relaxed Atomic Operations

Although Chapel’s relaxed atomic operations (*memory_order_relaxed*) do not complete in a total order by themselves and might contribute to non-deterministic programs, relaxed atomic operations cannot contribute to a data race that would prevent sequential consistency.

When relaxed atomics are used only for atomicity and not as part of synchronizing tasks, their effect can be understood in the memory consistency model described in 31.1 by treating them as ordinary loads or stores with two exceptions:

• Atomic operations (including relaxed atomic operations) cannot create data races.
• All atomic operations (including relaxed atomic operations) will eventually be visible to all other threads. This property is not true for normal loads and stores.
31.3 Unordered Memory Operations

*Open issue.* Syntax for unordered operations has not yet been finalized.

*Open issue.* Should Chapel provide a memory fence that only completes unordered operations started by the current task?

*Open issue.* Should unordered operations on a particular memory address always wait for the address to be computed?

*Open issue.* Does starting a task or joining with a task necessarily wait for unordered operations to complete?

Rather than issuing normal loads and stores to read or write local or remote memory, a Chapel program can use unordered loads and stores when preserving sequential program behavior is not important. The following notation for unordered memory operations will be used in this section:

- $UL(a)$ indicates an unordered load from a variable at address $a$. $a$ could point to local or remote memory.
- $US(a)$ indicates an unordered store to a variable at address $a$. Again, $a$ could point to local or remote memory.

The unordered loads and stores $UL(a)$ and $US(a)$ respect fences but not program order. As in Section 31.1.2, unordered loads and stores are ordered with SC atomics. That is, unordered loads and stores for a given task are in total order $<_m$ respecting the following rules which preserve the order of unordered loads and stores relative to SC atomic operations:

- If $UL(a) <_p A_{sc}(b)$ then $UL(a) <_m A_{sc}(b)$
- If $US(a) <_p A_{sc}(b)$ then $US(a) <_m A_{sc}(b)$
- If $A_{sc}(a) <_p UL(b)$ then $A_{sc}(a) <_m UL(b)$
- If $A_{sc}(a) <_p US(b)$ then $A_{sc}(a) <_m US(b)$

Unordered loads and stores do not preserve sequential program behavior.

31.3.1 Unordered Memory Operations Examples

Unordered operations should be thought of as happening in a way that overlaps with the program task. Unordered operations started in different program statements can happen in any order unless an SC atomic operation orders them.

Since unordered operations started by a single task can happen in any order, totally sequential programs can have a data race when using unordered operations. This follows from our original definition of data race.
Memory Consistency Model

```plaintext
var x: int = 0;
unordered_store(x, 10);
unordered_store(x, 20);
writeln(x);
```

The value of \( x \) at the end of this program could be 0, 10, or 20. As a result, programs using unordered loads and stores are not sequentially consistent unless the program can guarantee that unordered operations can never operate on the same memory at the same time when one of them is a store. In particular, the following are safe:

- Unordered stores to disjoint regions of memory.
- Unordered loads from potentially overlapping regions of memory when no store could overlap with the loads.
- Unordered loads and stores to the same memory location when these are always separated by an SC atomic operation.

Unordered loads and stores are available as a performance optimization. For example, a program computing a permutation on an array might want to move data between two arrays without requiring any ordering:

```plaintext
const n = 10;
// P is a permutation on 1..n, in this case reversing its input
var P = for i in 1..n by -1 do i;
// A is an array to permute
var A = for i in 1..n do i;
// Compute, in B, the permutation applied to A
var B:[1..n] int;

for i in 1..n {
  unordered_store(B[P[i]], A[i]);
}
```

### 31.4 Examples

**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.

The program

```plaintext
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 = 14; // an arbitrary number
  for i in 1..numToWrite do
    A[i] = i/10.0;
  done$ = numToWrite; // fence writes to A and signal done
}
```
produces the output

\[
\begin{align*}
A[1] &= 0.1 \\
A[2] &= 0.2 \\
A[3] &= 0.3 \\
A[4] &= 0.4 \\
A[5] &= 0.5 \\
A[6] &= 0.6 \\
A[7] &= 0.7 \\
A[8] &= 0.8 \\
A[9] &= 0.9 \\
A[10] &= 1.0 \\
A[12] &= 1.2 \\
A[14] &= 1.4
\end{align*}
\]

**Example (syncSpinWait.chpl).** One consequence of Chapel’s memory consistency model is that a task cannot spin-wait on a normal variable waiting for another task to write to that variable. The behavior of the following code is undefined:

```chapel
var x: int;
cobegin with (ref x) {
  while x != 1 do ; // INCORRECT spin wait
  x = 1;
}
```

In contrast, spinning on a synchronization variable has well-defined behavior:

```chapel
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.

**Example (atomicSpinWait.chpl).** Atomic variables provide an alternative means to spin-wait. For example:

```chapel
var x: atomic int;
cobegin {
  while x.read() != 1 do ; // spin wait - monopolizes processor
  x.write(1);
}
```

**Example (atomicWaitFor.chpl).** The main drawback of the above example is that it prevents the thread executing the spin wait from doing other useful work. Atomic variables include a waitFor method that will block the calling thread until a read of the atomic value matches a particular value. In contrast to the spin wait loop above, waitFor will allow other tasks to be scheduled. For example:

```chapel
var x: atomic int;
cobegin {
  x.waitFor(1);
  x.write(1);
}
```

**Future.** Upon completion, Chapel’s atomic statement (§26.10) will serve as an additional means of correctly synchronizing between tasks.
32 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 §32.1. Details on sharing types, variables and procedures are supplied in §32.2.

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 desirable 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.

32.1 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.

32.1.1 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 code 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`.

It is possible to use the `external-name` syntax to create an `extern` function that presents a different name to Chapel code than the name of the function actually used when linking. The `external-name` expression must evaluate to a `param string`. For example, the code below declares a function callable in Chapel as `c_atoi` but that will actually link with the C `atoi` function.

```
extern "atoi" proc c_atoi(arg:c_string):c_int;
```

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.\footnote{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.}

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.

32.1.2 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

external-name:
   expression
```

\footnote{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.}
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.

As with the `extern-name` for `extern proc`, if this syntax element is provided, then it must be a `param string` and will be used to determine the name of the function to use when linking. For example, the code below declares a function callable in C as `chapel_addone` but it is callable from Chapel code as `addone`:

```chapel
export "chapel_addone" proc addone(arg:c_int):c_int {
    return arg+1;
}
```

_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.

### 32.2 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.

#### 32.2.1 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.

<table>
<thead>
<tr>
<th>C Type</th>
<th>Chapel Type</th>
<th>C Type</th>
<th>Chapel Type</th>
<th>C Type</th>
<th>Chapel Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int8_t</code></td>
<td><code>int(8)</code></td>
<td><code>uint8_t</code></td>
<td><code>uint(8)</code></td>
<td><code>_real32</code></td>
<td><code>real(32)</code></td>
</tr>
<tr>
<td><code>int16_t</code></td>
<td><code>int(16)</code></td>
<td><code>uint16_t</code></td>
<td><code>uint(16)</code></td>
<td><code>_real64</code></td>
<td><code>real(64)</code></td>
</tr>
<tr>
<td><code>int32_t</code></td>
<td><code>int(32)</code></td>
<td><code>uint32_t</code></td>
<td><code>uint(32)</code></td>
<td><code>_imag32</code></td>
<td><code>imag(32)</code></td>
</tr>
<tr>
<td><code>int64_t</code></td>
<td><code>int(64)</code></td>
<td><code>uint64_t</code></td>
<td><code>uint(64)</code></td>
<td><code>_imag64</code></td>
<td><code>imag(64)</code></td>
</tr>
<tr>
<td><code>chpl_bool</code></td>
<td><code>bool</code></td>
<td><code>const char*</code></td>
<td><code>c_string</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>_complex64</code></td>
<td><code>complex(64)</code></td>
<td><code>complex</code></td>
<td><code>complex(128)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

**Referring to External C Types**

An externally-defined type can be referenced using a external type declaration with the following syntax.

\[ \text{external-type-alias-declaration-statement:} \]
\[ \text{extern type type-alias-declaration-list; } \]

In each type-alias-declaration, if the type-expression 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 homogeneous tuple type. For example, the C typedef
\begin{verbatim}
typedef double vec[3];
\end{verbatim}
can be described in Chapel using
\begin{verbatim}
extern type vec = 3*real(64);
\end{verbatim}

**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.

\[ \text{external-record-declaration-statement:} \]
\[ \text{extern external-name_opt simple-record-declaration-statement} \]

For example, consider an external C structure defined in `foo.h` called `fltdbl`.
\begin{verbatim}
typedef struct _fltdbl {
    float x;
    double y;
} fltdbl;
\end{verbatim}

This type could be referred to within a Chapel program using
\begin{verbatim}
extern record fltdbl {
    var x: real(32);
    var y: real(64);
}
\end{verbatim}

\(^2\)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.
Interoperability

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. Some effort is made to preserve the values of the undefined fields when copying these structs but Chapel cannot guarantee the contents or memory story of fields of which it has no knowledge.

If the optional external-name is supplied, then it is used verbatim as the exported struct symbol.

A C header file containing the struct’s definition in C must be specified on the chpl compiler command line. Note that only typedef’d C structures are supported by default. That is, in the C header file, the struct must be supplied with a type name through a typedef declaration. If this is not true, you can use the external-name part to apply the struct specifier. As an example of this, given a C declaration of:

```c
struct Vec3 {
    double x, y, z;
};
```

in Chapel you would refer to this struct via

```chapel
extern "struct Vec3" record Vec3 { 
    var x, y, z: real(64); 
}
```

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 a pointer to an external record) as follows:

```chapel
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.

```chapel
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.
282

32.2.2 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:

```chapel
extern var bar: foo;
```

would tell the Chapel compiler about an external C variable named bar of type foo. Similarly,

```chapel
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 the Chapel compiler does not necessarily parse C code, external params are not supported.

32.2.3 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:

```chapel
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.

If the above function returns a C double, it would be declared as:

```chapel
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

```chapel
extern proc foo(x: int, y): real;
var a, b: int;
foo(a, b);
```

---

3The return type cannot be inferred, since an extern procedure declaration has no body.
Interoperability

would imply that the external function foo takes two 64-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 corresponding actual argument is omitted at the callsite. For example:

```chapel
default 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

```chapel
extern proc printf(fmt: c_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:

```chapel
default 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.

32.2.4 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 a 64-bit integer can be declared as:

```chapel
export proc foo(): int { ... }
```

If the optional external-name is supplied, that is the name used in linking with external code. For example, if we declare

```chapel
export "myModule_foo" proc foo(): int { ... }
```

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.

32.2.5 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.

---

4In practice, this will typically only be useful if the external function is a macro or built-in that can handle type identifiers.
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*, *out*, or *ref* intent. The C code implementing that function must dereference the argument to extract its value.
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

\[ \text{binary-digit: one of } 0 1 \]

\[ \text{binary-digits: } \text{binary-digit} \text{ binary-digit separator binary-digits} \]

\[ \text{bool-literal: one of } \text{true false} \]

\[ \text{bytes-literal: } \text{interpreted-bytes-literal} \text{ uninterpreted-bytes-literal} \]

\[ \text{digit: one of } 0 1 2 3 4 5 6 7 8 9 \]

\[ \text{digits: } \text{digit} \text{ digit separator digits} \]

\[ \text{double-quote-delimited-characters: } \text{string-character double-quote-delimited-characters_{opt}} \text{ double-quote-delimited-characters_{opt}} \]

\[ \text{exponent-part: } \text{e sign_{opt} digits} \text{ E sign_{opt} digits} \]

\[ \text{hexadecimal-digit: one of } 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f \]

\[ \text{hexadecimal-digits: } \text{hexadecimal-digit} \text{ hexadecimal-digit separator hexadecimal-digits} \]

\[ \text{hexadecimal-escape-character: } \text{\x hexadecimal-digits} \]

\[ \text{identifier: } \text{letter-or-underscore legal-identifier-char_{opt}} \]
imaginary–literal:
  real–literal i
  integer–literal i

integer–literal:
  digits
  0x hexadecimal–digits
  0X hexadecimal–digits
  0o octal–digits
  0O octal–digits
  0b binary–digits
  0B binary–digits

interpreted–bytes–literal:
  b" double–quote–delimited–characters_{opt} "
  b' single–quote–delimited–characters_{opt} '

interpreted–string–literal:
  " double–quote–delimited–characters_{opt} "
  ' single–quote–delimited–characters_{opt} '

legal–identifier–char:
  letter–or–underscore
  digit
  $ 

legal–identifier–chars:
  legal–identifier–char legal–identifier–chars_{opt}

letter–or–underscore:
  letter
  _

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

octal–digit: one of
  0 1 2 3 4 5 6 7

goctal–digits:
  octal–digit
  octal–digit separator–octal–digits

p–exponent–part:
  p sign_{opt} digits
  P sign_{opt} digits

real–literal:
  digits_{opt} . digits exponent–part_{opt}
  digits_{opt} exponent–part
  0x hexadecimal–digits_{opt} . hexadecimal–digits p–exponent–part_{opt}
  0X hexadecimal–digits_{opt} . hexadecimal–digits p–exponent–part_{opt}
  0x hexadecimal–digits_{opt} p–exponent–part
  0X hexadecimal–digits_{opt} p–exponent–part
Collected Lexical and Syntax Productions

separator\-binary\-digits:
  binary\-digit
  _ binary\-digit separator\-binary\-digits
  _ separator\-binary\-digits

separator\-digits:
  digit
  _ digit separator\-digits
  _ separator\-digits

separator\-hexadecimal\-digits:
  hexadecimal\-digit
  _ hexadecimal\-digit separator\-hexadecimal\-digits
  _ separator\-hexadecimal\-digits

separator\-octal\-digits:
  octal\-digit
  _ octal\-digit separator\-octal\-digits
  _ separator\-octal\-digits

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:
  interpreted\-string\-literal
  uninterpreted\-string\-literal

uninterpreted\-bytes\-literal:
  b"" uninterpreted\-double\-quote\-delimited\-characters ""
  b'' uninterpreted\-single\-quote\-delimited\-characters ''

uninterpreted\-double\-quote\-delimited\-characters:
  uninterpreted\-double\-quote\-string\-character uninterpreted\-double\-quote\-delimited\-characters_{opt}

uninterpreted\-double\-quote\-string\-character:
  any character except three double quotes in a row

uninterpreted\-single\-quote\-delimited\-characters:
  uninterpreted\-single\-quote\-string\-character uninterpreted\-single\-quote\-delimited\-characters_{opt}
uninterpreted–single–quote–string–character:
any character except three single quotes in a row

uninterpreted–string–literal:
"""uninterpreted–double–quote–delimited–characters """
"""uninterpreted–single–quote–delimited–characters """

A.2 Alphabetical Syntax Productions

aligned–range–expression:
  range–expression align expression

argument–list:
  ( formalsOpt )

array–literal:
  rectangular–array–literal
  associative–array–literal

array–type:
  [ domain–expression ] type–expression

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

assignment–statement:
  lvalue–expression assignment–operator expression

associative–array–literal:
  [ associative–expr–list ]
  [ associative–expr–list , ]

associative–domain–literal:
  [ associative–expression–list ]

associative–domain–type:
  domain ( associative–index–type )
  domain ( opaque )

associative–expr–list:
  index–expr => value–expr
  index–expr => value–expr, associative–expr–list

associative–expression–list:
  non–range–expression
  non–range–expression, associative–expression–list

associative–index–type:
  type–expression
atomic-statement:
  atomic statement

atomic-type:
  atomic type-expression

base-domain-type:
  rectangular-domain-type
  associative-domain-type

basic-class-type:
  identifier
  identifier ( named-expression-list )

begin-statement:
  begin task-intent-clause_opt statement

binary-expression:
  expression binary-operator expression

binary-operator: one of
  + - * / % ** & | ^ << >> && || == != <= >= < >

by #

block-statement:
  { statements_opt }

break-statement:
  break identifier_opt ;

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

cast-expression:
  expression : type-expression

class-declaration-statement:
  class identifier class-inherit_opt { class-statement-list_opt }

class-inherit:
  : basic-class-type

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:
  basic−class−type
  owned basic−class−type
  shared basic−class−type
  borrowed basic−class−type
  unmanaged basic−class−type

cobegin−statement:
  cobegin task−intent−clause_opt block−statement

coforall−statement:
  coforall index−var−declaration in iterable−expression task−intent−clause_opt do statement
  coforall index−var−declaration in iterable−expression task−intent−clause_opt block−statement
  coforall iterable−expression task−intent−clause_opt do statement
  coforall iterable−expression task−intent−clause_opt 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

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

defer−statement:
  defer statement

delete−statement:
  delete expression−list :

dmap−value:
  expression

do−while−statement:
  do statement while expression ;

domain−alignment−expression:
  domain−expression align expression
domain-assignment-expression:
  domain-name = domain-expression

domain-expression:
  domain-literal
  domain-name
  domain-assignment-expression
  domain-striding-expression
  domain-alignment-expression
  domain-slice-expression

domain-literal:
  rectangular-domain-literal
  associative-domain-literal

domain-name:
  identifier

domain-slice-expression:
  domain-expression [ slicing-index-set ]
  domain-expression ( slicing-index-set )

domain-striding-expression:
  domain-expression by expression

domain-type:
  base-domain-type
  simple-subdomain-type
  sparse-subdomain-type

domain-type:

else-part:
  else statement

empty-statement:
  ;

enum-constant-expression:
  enum-type . identifier

enum-constant-list:
  enum-constant
  enum-constant , enum-constant-list

enum-constant:
  identifier init-part

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

enum-type:
  identifier

exclude-list:
  identifier-list
  *
Collected Lexical and Syntax Productions

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:
  variable-expression ;
  member-access-expression ;
  call-expression ;
  new-expression ;
  let-expression ;

expression:
  literal-expression
  nil-expression
  variable-expression
  enum-constant-expression
  call-expression
  type-expression
  iterable-call-expression
  member-access-expression
  new-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-name:
  expression

external-procedure-declaration-statement:
  extern external-name_opt proc function-name argument-list return-intent_opt return-type_opt

external-record-declaration-statement:
  extern external-name_opt 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 iterable-expression do expression
  for iterable-expression do expression

for-statement:
  for index-var-declaration in iterable-expression do statement
  for index-var-declaration in iterable-expression block-statement
  for iterable-expression do statement
  for iterable-expression block-statement

forall-expression:
  forall index-var-declaration in iterable-expression task-intent-clause_opt do expression
  forall iterable-expression task-intent-clause_opt do expression
  [ index-var-declaration in iterable-expression task-intent-clause_opt ] expression
  [ iterable-expression task-intent-clause_opt ] expression

forall-statement:
  forall index-var-declaration in iterable-expression task-intent-clause_opt do statement
  forall index-var-declaration in iterable-expression task-intent-clause_opt block-statement
  forall iterable-expression task-intent-clause_opt do statement
  forall iterable-expression task-intent-clause_opt block-statement
  [ index-var-declaration in iterable-expression task-intent-clause_opt ] statement
  [ iterable-expression task-intent-clause_opt ] statement

formal-intent:
  const
  const in
  const ref
  in
  out
  inout
  ref
  param
  type

formal-type:
  : type-expression
  : ? 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-expr:
  expression

index-type:
  index ( domain-expression )

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

init-part:
  = expression

initialization-part:
  = expression

integer-parameter-expression:
  expression

io-expression:
  expression
  io-expression io-operator expression

io-operator:
  <~>?

io-statement:
  io-expression io-operator expression

iteratable-call-expression:
  call-expression

iteratable-expression:
  expression
  zip ( expression-list )

iterator-body:
  block-statement
  yield-statement
iterator-declaration-statement:
  privacy-specifier_{opt} iter iterator-name argument-list_{opt} yield-intent_{opt} yield-type_{opt} where-clause_{opt}
  iterator-body

iterator-name:
  identifier

label-statement:
  label identifier statement

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

limitation-clause:
  except exclude-list
  only rename-list_{opt}

literal-expression:
  bool-literal
  integer-literal
  real-literal
  imaginary-literal
  string-literal
  bytes-literal
  range-literal
  domain-literal
  array-literal

locale-access-expression:
  expression . locale

lvalue-expression:
  variable-expression
  member-access-expression
  call-expression
  parenthesized-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:
  procedure-kind_{opt} proc-or-iter this-intent_{opt} type-binding_{opt} 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:
  privacy-specifier_opt prototype-specifier_opt module module-identifier block-statement

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

module-identifier:
  identifier

module-or-enum-name-list:
  module-or-enum-name limitation-clause_opt
  module-or-enum-name , module-or-enum-name-list

module-or-enum-name:
  identifier
  identifier . module-or-enum-name

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

named-expression:
  expression
  identifier = expression

new-expression:
  new type-expression ( argument-list )

nil-expression:
  nil

no-initialization-part:
  = noinit

non-range-expression:
  expression

on-statement:
  on expression do statement
  on expression block-statement

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

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
Collected Lexical and Syntax Productions

parenthesesless−function−identifier:
    identifier

parenthesized−expression:
    ( expression )

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

primitive−type:
    void
    nothing
    bool primitive−type−parameter−part\textsubscript{opt}
    int primitive−type−parameter−part\textsubscript{opt}
    uint primitive−type−parameter−part\textsubscript{opt}
    real primitive−type−parameter−part\textsubscript{opt}
    imag primitive−type−parameter−part\textsubscript{opt}
    complex primitive−type−parameter−part\textsubscript{opt}
    string
    bytes
    enum
    record
    class
    owned
    shared
    unmanaged
    borrowed

privacy−specifier:
    private
    public

proc−or−iter:
    proc
    iter

procedure−declaration−statement:
    privacy−specifier\textsubscript{opt} procedure−kind\textsubscript{opt} proc function−name argument−list\textsubscript{opt} return−intent\textsubscript{opt} return−type\textsubscript{opt} where−clause\textsubscript{opt}
    function−body

procedure−kind:
    inline
    export
    extern
    override

prototype−specifier:
    prototype

query−expression:
    ? identifier\textsubscript{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–statement–list:
  record–statement
  record–statement record–statement–list
record–statement:
  variable–declaration–statement
  method–declaration–statement
  type–declaration–statement
  empty–statement
record–type:
  identifier
  identifier ( named–expression–list )
rectangular–array–literal:
  [ expression–list ]
  [ expression–list , ]
rectangular–domain–literal:
  { range–expression–list }
rectangular–domain–type:
  domain ( named–expression–list )
reduce–expression:
  reduce–scan–operator reduce iteratable–expression
  class–type reduce iteratable–expression
reduce–scan–operator: one of
  + * & & || & | ^ min max minloc maxloc
remote-variable-declaration-statement:
  on expression variable-declaration-statement

rename-base:
  identifier as identifier
  identifier

rename-list:
  rename-base
  rename-base, rename-list

return-intent:
  const
  const ref
  ref
  param
  type

return-statement:
  return expression_opt :

return-type:
  : type-expression

scan-expression:
  reduce scan-operator scan iterable-expression
  class-type scan iterable-expression

select-statement:
  select expression { when-statements }

serial-statement:
  serial expression_opt do statement
  serial expression_opt block-statement

simple-record-declaration-statement:
  record identifier { record-statement-list }

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

single-type:
  single type-expression

sliced-range-expression:
  range-expression ( range-expression )
  range-expression [ range-expression ]

slicing-index-set:
  domain-expression
  range-expression-list

sparse-subdomain-type:
  sparse subdomain_opt ( domain-expression )
statement:
  block−statement
  expression−statement
  assignment−statement
  swap−statement
  io−statement
  conditional−statement
  select−statement
  while−do−statement
  do−while−statement
  for−statement
  label−statement
  break−statement
  continue−statement
  param−for−statement
  use−statement
  defer−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
Collected Lexical and Syntax Productions

swap-operator:
  <=>

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

sync-statement:
  sync statement
  sync block-statement

sync-type:
  sync type-expression

dsynchronization-type:
  sync-type
  single-type
  atomic-type

task-intent-clause:
  with ( task-intent-list )

task-intent-item:
  formal-intent identifier
  task-private-var-decl

task-intent-list:
  task-intent-item
  task-intent-item, task-intent-list

task-private-var-decl:
  task-private-var-kind identifier type-part, initialization-part, opt

task-private-var-kind:
  const
  var
  ref

this-intent:
  param
  type
  ref
  const ref
  const

tuple-component-list:
  tuple-component
  tuple-component, tuple-component-list

tuple-component:
  expression
  –

tuple-expand-expression:
  ( ... expression )
tuple-expression:
  ( tuple-component , )
  ( tuple-component , tuple-component-list )
  ( tuple-component , tuple-component-list , )

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

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

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

type-alias-declaration-statement:
  privacy-specifier-opt config-opt type type-alias-declaration-list ;
  external-type-alias-declaration-statement

type-alias-declaration:
  identifier = type-expression
  identifier

type-binding:
  identifier .
  ( expr ) .

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

type-expression:
  primitive-type
  enum-type
  structured-type
  dataparallel-type
  synchronization-type
  lvalue-expression
  if-expression
  unary-expression
  binary-expression

type-list:
  type-expression
  type-expression , type-list

type-part:
  : type-expression

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:
  privacy-specifier_opt use module-or-enum-name-list ;

value-exp:
  expression

variable-argument-expression:
  ... expression
  ... ? identifier_opt
  ...

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

variable-declaration-statement:
  privacy-specifier_opt config-or-extern_opt variable-kind variable-declaration-list ;

variable-declaration:
  identifier-list type-part_opt initialization-part
  identifier-list type-part no-initialization-part_opt

variable-expression:
  identifier

variable-kind:
  param
  const
  var
  ref
  const ref

when-statement:
  when expression-list do statement
  when expression-list block-statement
  otherwise statement
  otherwise do 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-intent:
  const
  const ref
  ref
  param
  type

yield-statement:
  yield expression ;

yield-type:
  : type-expression

A.3 Depth-First Lexical Productions

bool-literal: one of
  true false

bytes-literal:
  interpreted-bytes-literal
  uninterpreted-bytes-literal

interpreted-bytes-literal:
  b" double-quote-delimited-characters_{opt} "
  b' 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
hexadecimal-digits:
  hexadecimal-digit
  hexadecimal-digit separator 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

separator–hexadecimal-digits:
  hexadecimal-digit
  _ hexadecimal-digit separator hexadecimal-digits
  _ separator hexadecimal-digits

single–quote-delimited–characters:
  string–character single–quote–delimited–characters\_opt
  " single–quote–delimited–characters\_opt

uninterpreted–bytes–literal:
  b""" uninterpreted–double–quote–delimited–characters """""""
  b"" uninterpreted–single–quote–delimited–characters ""

uninterpreted–double–quote–delimited–characters:

uninterpreted–double–quote–string–character:
  any character except three double quotes in a row

uninterpreted–single–quote–delimited–characters:

uninterpreted–single–quote–string–character:
  any character except three single quotes in a row

identifier:
  letter–or–underscore legal–identifier–chars\_opt

letter–or–underscore:
  letter
  _

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–or–underscore
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
  0x hexadecimal-digits_{opt} . hexadecimal-digits p-exponent-part_{opt}
  0X hexadecimal-digits_{opt} . hexadecimal-digits p-exponent-part_{opt}
  0x hexadecimal-digits _opt p-exponent-part
  0X hexadecimal-digits _opt p-exponent-part

digits:
  digit
  digit separator-digits

separator-digits:
  digit
  _
  digit separator-digits
  _ separator-digits

exponent-part:
  e sign_{opt} digits
  E sign_{opt} digits

sign: one of
  +  ~

p-exponent-part:
  p sign_{opt} digits
  P sign_{opt} digits

integer-literal:
  digits
  0x hexadecimal-digits
  0X hexadecimal-digits
  0o octal-digits
  0O octal-digits
  0b binary-digits
  0B binary-digits

octal-digits:
  octal-digit
  octal-digit separator-octal-digits

octal-digit: one of
  0 1 2 3 4 5 6 7

separator-octal-digits:
  octal-digit
  _
  octal-digit separator-octal-digits
  _ separator-octal-digits
binary-digits:
  binary-digit
  binary-digit separator binary-digits

binary-digit: one of
  0 1

separator-binary-digits:
  binary-digit
  separator binary-digits

string-literal:
  interpreted-string-literal
  uninterpreted-string-literal

interpreted-string-literal:
  " double-quote-delimited-characters opt "
  ' single-quote-delimited-characters opt '

uninterpreted-string-literal:
  " uninterpreted-double-quote-delimited-characters "
  ' uninterpreted-single-quote-delimited-characters '

A.4 Depth-First Syntax Productions

module-declaration-statement:
  privacy-specifier opt prototype-specifier opt module module-identifier block-statement

privacy-specifier:
  private
  public

prototype-specifier:
  prototype

module-identifier:
  identifier

block-statement:
  { statements opt }

statements:
  statement
  statement statements

statement:
  block-statement
  expression-statement
  assignment-statement
  swap-statement
  io-statement
  conditional-statement
Collected Lexical and Syntax Productions

select-statement
while-do-statement
do-while-statement
for-statement
label-statement
break-statement
continue-statement
param-for-statement
use-statement
defer-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:
  variable-expression ;
  member-access-expression ;
  call-expression ;
  new-expression ;
  let-expression ;

variable-expression:
  identifier

member-access-expression:
  field-access-expression
  method-call-expression

field-access-expression:
  receiver-clause_{opt} identifier

receiver-clause:
  expression .

expression:
  literal-expression
  nil-expression
variable-expression
enum-constant-expression
call-expression
type-expression
iteratable-call-expression
member-access-expression
new-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
  bytes-literal
  range-literal
  domain-literal
  array-literal

range-literal:
  expression .. expression
  expression ..
  .. expression
  ..

domain-literal:
  rectangular-domain-literal
  associative-domain-literal

rectangular-domain-literal:
  { range-expression-list }

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

range-expression:
  expression
**Collected Lexical and Syntax Productions**

- `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` ]

**associative-domain-literal:**
  - { `associative-expression-list` }

**associative-expression-list:**
  - `non-range-expression`
  - `non-range-expression`, `associative-expression-list`

**non-range-expression:**
  - `expression`

**array-literal:**
  - `rectangular-array-literal`
  - `associative-array-literal`

**rectangular-array-literal:**
  - [ `expression-list` ]
  - [ `expression-list`, ]

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

**associative-array-literal:**
  - [ `associative-expr-list` ]
  - [ `associative-expr-list`, ]

**associative-expr-list:**
  - `index-expr => value-expr`
  - `index-expr => value-expr`, `associative-expr-list`

**index-expr:**
  - `expression`
value–expr:
  expression

nil–expression:
  nil

enum–constant–expression:
  enum–type . identifier

enum–type:
  identifier
type–expression:
  primitive–type
  enum–type
  structured–type
dataparallel–type
  synchronization–type
  lvalue–expression
  if–expression
  unary–expression
  binary–expression

primitive–type:
  void
  nothing
  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
  bytes
  enum
  record
  class
  owned
  shared
  unmanaged
  borrowed

primitive–type–parameter–part:
  ( integer–parameter–expression )

integer–parameter–expression:
  expression

structured–type:
  class–type
  record–type
  union–type
  tuple–type
class–type:
  basic–class–type
  owned basic–class–type
  shared basic–class–type
  borrowed basic–class–type
  unmanaged basic–class–type

basic–class–type:
  identifier
  identifier ( named–expression–list )

named–expression–list:
  named–expression
  named–expression , named–expression–list

named–expression:
  expression
  identifier = expression

record–type:
  identifier
  identifier ( named–expression–list )

union–type:
  identifier
  identifier

tuple–type:
  ( type–expression , type–list )
  ( type–expression , )

type–list:
  type–expression
  type–expression , 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 )
Collected Lexical and Syntax Productions

associative-domain-type:
  domain ( associative-index-type )
  domain ( opaque )

associative-index-type:
  type-expression

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

domain-expression:
  domain-literal
  domain-name
  domain-assignment-expression
  domain-striding-expression
  domain-alignment-expression
  domain-slice-expression

domain-name:
  identifier

domain-assignment-expression:
  domain-name = domain-expression

domain-striding-expression:
  domain-expression by expression

domain-alignment-expression:
  domain-expression align expression

domain-slice-expression:
  domain-expression [ slicing-index-set ]
  domain-expression ( slicing-index-set )

slicing-index-set:
  domain-expression
  range-expression-list

sparse-subdomain-type:
  sparse subdomain_opt ( domain-expression )

mapped-domain-type:
  domain-type dmapped dmap-value

dmap-value:
  expression

array-type:
  [ domain-expression ] type-expression

index-type:
  index ( domain-expression )
synchronization-type:
  sync-type
  single-type
  atomic-type

sync-type:
  sync type-expression

single-type:
  single type-expression

atomic-type:
  atomic type-expression

iteratable-call-expression:
  call-expression

query-expression:
  ? identifier_opt

cast-expression:
  expression : type-expression

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

parenthesized-expression:
  ( expression )

unary-expression:
  unary-operator expression

unary-operator: one of
  + - ~ !

binary-expression:
  expression binary-operator expression

binary-operator: one of
  + - * / % ** & ^ << >> && || == != <= >= < >

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

for-expression:
  for index-var-declaration in iterable-expression do expression
  for iterable-expression do expression

forall-expression:
  forall index-var-declaration in iterable-expression task-intent-clause_opt do expression
  forall iterable-expression task-intent-clause_opt do expression
  [ index-var-declaration in iterable-expression task-intent-clause_opt ] expression
  [ iterable-expression task-intent-clause_opt ] expression
index-var-declaration:
  identifier
tuple-grouped-identifier-list

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

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

iteratable-expression:
  expression
  zip ( expression-list )

task-intent-clause:
  with ( task-intent-list )

task-intent-list:
  task-intent-item
task-intent-item, task-intent-list

task-intent-item:
  formal-intent identifier
task-private-var-decl

formal-intent:
  const
cost in
cost ref
in
out
inout
ref
param


task-private-var-decl:
  task-private-var-kind identifier type-part\opt initialization-part\opt

task-private-var-kind:
  const
  var
  ref

type-part:
  : type-expression

initialization-part:
  = expression

reduce-expression:
  reduce scan-operator reduce iteratable-expression
class-type reduce iteratable-expression
reduce–scan–operator: one of
  + * && || & ^ min max minloc maxloc

scan–expression:
  reduce–scan–operator scan iterable–expression
  class–type scan iterable–expression

module–access–expression:
  module–identifier–list . identifier

module–identifier–list:
  module–identifier
  module–identifier . module–identifier–list

tuple–expression:
  ( tuple–component , )
  ( tuple–component , tuple–component–list )
  ( tuple–component , tuple–component–list , )

tuple–component:
  expression
  –

tuple–component–list:
  tuple–component
  tuple–component , tuple–component–list

tuple–expand–expression:
  ( ... expression )

locale–access–expression:
  expression . locale

mapped–domain–expression:
  domain–expression dmapped dmap–value

method–call–expression:
  receiver–clause_mod expression ( named–expression–list )
  receiver–clause_mod expression [ named–expression–list ]
  receiver–clause_mod parenthesesless–function–identifier

parenthesesless–function–identifier:
  identifier

call–expression:
  lvalue–expression ( named–expression–list )
  lvalue–expression [ named–expression–list ]
  parenthesesless–function–identifier

new–expression:
  new type–expression ( argument–list )

argument–list:
  ( formals_mod )
formals:
formal
formal, formals

formal:
formal-intent, identifier formal-type default-expression
formal-intent, identifier formal-type variable-argument-expression
formal-intent, tuple-grouped-identifier-list formal-type default-expression
formal-intent, tuple-grouped-identifier-list formal-type variable-argument-expression

default-expression:
= expression

formal-type:
: type-expression
: ? identifier

variable-argument-expression:
... expression
... ? identifier
...

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

assignment-statement:
lvalue-expression assignment-operator expression

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

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

swap-operator:
<=>

io-statement:
io-expression io-operator expression

io-expression:
expression
io-expression io-operator expression

io-operator:
<=>

conditional-statement:
if expression then statement else-part
if expression block-statement else-part

else-part:
else statement
Collected Lexical and Syntax Productions

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
  otherwise do 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 iterable-expression do statement
  for index-var-declaration in iterable-expression block-statement
  for iterable-expression do statement
  for iterable-expression block-statement

label-statement:
  label identifier statement

break-statement:
  break identifieropt ;

continue-statement:
  continue identifieropt ;

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:
  privacy-specifieropt, use module-or-enum-name-list ;

module-or-enum-name-list:
  module-or-enum-name limitation-clauseopt
  module-or-enum-name, module-or-enum-name-list

limitation-clause:
  except exclude-list
  only rename-listopt
Collected Lexical and Syntax Productions

exclude-list:
    identifier-list
    *

rename-list:
    rename-base
    rename-base, rename-list

defer-statement:
    defer statement

defer-statement:
    defer statement

empty-statement:
    :

return-statement:
    return expression_opt ;

yield-statement:
    yield expression ;

module-declaration-statement:
    privacy-specifier_opt prototype-specifier_opt module module-identifier-block-statement

procedure-declaration-statement:
    privacy-specifier_opt procedure-kind_opt proc function-name argument-list_opt return-intent_opt return-type_opt where-clause_opt
    function-body

procedure-kind:
    inline
    export
    extern
    override

function-name:
    identifier
    operator-name

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

return-intent:
    const
    const ref
    ref
    param
    type
return-type:
  : type-expression

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

iterator-declaration-statement:
  privacy-specifier opt iter iterator-name argument-list yield-intent opt yield-type opt where-clause opt

iterator-name:
  identifier

yield-intent:
  const
  const ref
  ref
  param
  type

yield-type:
  : type-expression

iterator-body:
  block-statement
  yield-statement

method-declaration-statement:
  procedure-kind opt proc-or-iter this-intent opt type-binding opt function-name argument-list opt
  return-intent opt return-type opt where-clause opt function-body

proc-or-iter:
  proc
  iter

this-intent:
  param
  type
  ref
  const ref
  const

type-binding:
  identifier .
  ( expr ) .
Collected Lexical and Syntax Productions

\[
\begin{align*}
type-declaration-statement: \\
\quad&\text{enum-declaration-statement} \\
\quad&\text{class-declaration-statement} \\
\quad&\text{record-declaration-statement} \\
\quad&\text{union-declaration-statement} \\
\quad&\text{type-alias-declaration-statement}
\end{align*}
\]

\[
\text{enum-declaration-statement:}
\quad\text{enum identifier} \{ \text{enum-constant-list} \}
\]

\[
\text{enum-constant-list:}
\quad\text{enum-constant}
\quad\text{enum-constant, enum-constant-list\_opt}
\]

\[
\text{enum-constant:}
\quad\text{identifier init-part\_opt}
\]

\[
\text{init-part:}
\quad= \text{expression}
\]

\[
\text{class-declaration-statement:}
\quad\text{class identifier class-inherit\_opt} \{ \text{class-statement-list\_opt} \}
\]

\[
\text{class-inherit:}
\quad: \text{basic-class-type}
\]

\[
\text{class-statement-list:}
\quad\text{class-statement}
\quad\text{class-statement class-statement-list}
\]

\[
\text{class-statement:}
\quad\text{variable-declaration-statement}
\quad\text{method-declaration-statement}
\quad\text{type-declaration-statement}
\quad\text{empty-statement}
\]

\[
\text{record-declaration-statement:}
\quad\text{simple-record-declaration-statement}
\quad\text{external-record-declaration-statement}
\]

\[
\text{simple-record-declaration-statement:}
\quad\text{record identifier} \{ \text{record-statement-list} \}
\]

\[
\text{record-statement-list:}
\quad\text{record-statement}
\quad\text{record-statement record-statement-list}
\]

\[
\text{record-statement:}
\quad\text{variable-declaration-statement}
\quad\text{method-declaration-statement}
\quad\text{type-declaration-statement}
\quad\text{empty-statement}
\]

\[
\text{external-record-declaration-statement:}
\quad\text{extern external-name\_opt simple-record-declaration-statement}
\]
external-name:
  expression

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:
  privacy-specifier_opt 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-expression
  identifier

external-type-alias-declaration-statement:
  extern type type-alias-declaration-list ;

variable-declaration-statement:
  privacy-specifier_opt config-or-extern_opt variable-kind variable-declaration-list ;

config-or-extern: one of
  config extern

variable-kind:
  param
  const
  var
  ref
  const ref

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

variable-declaration:
  identifier-list type-part_opt initialization-part
  identifier-list type-part no-initialization-part_opt

no-initialization-part:
  = nolist
remote-variable-declaration-statement:
  on expression variable-declaration-statement

on-statement:
  on expression do statement
  on expression block-statement

cobegin-statement:
  cobegin task-intent-clause_opt block-statement

coforall-statement:
  coforall index-var-declaration in iterable-expression task-intent-clause_opt do statement
  coforall index-var-declaration in iterable-expression task-intent-clause_opt block-statement
  coforall iterable-expression task-intent-clause_opt do statement
  coforall iterable-expression task-intent-clause_opt block-statement

begin-statement:
  begin task-intent-clause_opt statement

csync-statement:
  sync statement
  sync block-statement

serial-statement:
  serial expression_opt do statement
  serial expression_opt block-statement

atomic-statement:
  atomic statement

forall-statement:
  forall index-var-declaration in iterable-expression task-intent-clause_opt do statement
  forall index-var-declaration in iterable-expression task-intent-clause_opt block-statement
  forall iterable-expression task-intent-clause_opt do statement
  forall iterable-expression task-intent-clause_opt block-statement
  [ index-var-declaration in iterable-expression task-intent-clause_opt ] statement
  [ iterable-expression task-intent-clause_opt ] statement

delete-statement:
  delete expression-list ;
Index

!=, 74
!= (record), 173
!= (string), 75
#, 76
# (domain), 203
& 69
&&, 72
&&=, 81
&=, 81
* 66
** 68
**=, 81
*=, 81
+ 64
+ (unary), 63
+=, 81
- 65
- (unary), 63
-=, 81
... (tuple expansion), 141
/ 67
/=, 81
: (cast), 58
<, 73
<=, 71
<==, 81
<=, 73
= (see also assignment), 81
=, 74
== (record), 173
== (string), 75
>, 73
>=, 74
>>=, 81
? (type query), 57
% 68
%-, 81
~ 69
^ 70
^= 81

acknowledgments, 19
actual arguments, 101, 102
add (atomic var), 243
align, 76
  on ranges, 185
  on rectangular domains, 202
and (atomic var), 243
argument
  intents, 106
argument passing
  domains, 200
arguments
  array, 214
  records, 171
arrays, 208
  actual arguments, 214
  assignment, 213, 257
  association with domains, 216
  associative
    literals, 209
  count operator, 214
  domain, 217
  domain maps, 210, 267
  element type, 208
eLTType, 216
  indexing, 210
  initialization, 208
  iteration, 212
  literals, 208
numElements, 217
predefined functions, 216
promotion, 215
rank, 217
rectangular
  default values, 209
  literals, 209
  reshape, 217
  returning, 215
runtime representation, 210
size, 217
slicing, 213
  rectangular, 214
sparse, 215
types, 208
values, 208
assignment, 81
array, 213
class, 164
### Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain</td>
<td>53</td>
</tr>
<tr>
<td>enumeration</td>
<td>52</td>
</tr>
<tr>
<td>explicit</td>
<td>50</td>
</tr>
<tr>
<td>class</td>
<td>52</td>
</tr>
<tr>
<td>domain</td>
<td>53</td>
</tr>
<tr>
<td>enumeration</td>
<td>52</td>
</tr>
<tr>
<td>numeric</td>
<td>51</td>
</tr>
<tr>
<td>range</td>
<td>53</td>
</tr>
<tr>
<td>string to bytes</td>
<td>53</td>
</tr>
<tr>
<td>tuple to complex</td>
<td>52</td>
</tr>
<tr>
<td>type to string</td>
<td>54</td>
</tr>
<tr>
<td>implicit</td>
<td>47</td>
</tr>
<tr>
<td>allowed types</td>
<td>47</td>
</tr>
<tr>
<td>boolean</td>
<td>48</td>
</tr>
<tr>
<td>borrow</td>
<td>49</td>
</tr>
<tr>
<td>class</td>
<td>49</td>
</tr>
<tr>
<td>generic types</td>
<td>50</td>
</tr>
<tr>
<td>nilable</td>
<td>49</td>
</tr>
<tr>
<td>numeric</td>
<td>48</td>
</tr>
<tr>
<td>occurs at</td>
<td>47</td>
</tr>
<tr>
<td>parameter</td>
<td>49</td>
</tr>
<tr>
<td>type arguments</td>
<td>50</td>
</tr>
<tr>
<td>numeric</td>
<td>48</td>
</tr>
<tr>
<td>parameter</td>
<td>49</td>
</tr>
<tr>
<td>range</td>
<td>53</td>
</tr>
<tr>
<td>source type</td>
<td>47</td>
</tr>
<tr>
<td>string to bytes</td>
<td>53</td>
</tr>
<tr>
<td>target type</td>
<td>47</td>
</tr>
<tr>
<td>tuple to complex</td>
<td>52</td>
</tr>
<tr>
<td>type to string</td>
<td>54</td>
</tr>
<tr>
<td>data parallelism</td>
<td>250</td>
</tr>
<tr>
<td>configuration constants</td>
<td>259</td>
</tr>
<tr>
<td>evaluation order</td>
<td>257</td>
</tr>
<tr>
<td>forall</td>
<td>250</td>
</tr>
<tr>
<td>forall expressions</td>
<td>252</td>
</tr>
<tr>
<td>forall intents</td>
<td>253</td>
</tr>
<tr>
<td>knobs for default data parallelism</td>
<td>259</td>
</tr>
<tr>
<td>leader iterator</td>
<td>251</td>
</tr>
<tr>
<td>promotion</td>
<td>255</td>
</tr>
<tr>
<td>reductions</td>
<td>258</td>
</tr>
<tr>
<td>scans</td>
<td>258</td>
</tr>
<tr>
<td>shadow variables</td>
<td>253</td>
</tr>
<tr>
<td>task-private variables</td>
<td>254</td>
</tr>
<tr>
<td>dataParIgnoreRunningTasks</td>
<td>259</td>
</tr>
<tr>
<td>dataParMinGranularity</td>
<td>259</td>
</tr>
<tr>
<td>dataParTasksPerLocale</td>
<td>259</td>
</tr>
<tr>
<td>deallocation</td>
<td></td>
</tr>
<tr>
<td>unmanaged class instances</td>
<td>165</td>
</tr>
<tr>
<td>declarations</td>
<td></td>
</tr>
<tr>
<td>records</td>
<td>167</td>
</tr>
<tr>
<td>union</td>
<td>176</td>
</tr>
<tr>
<td>variables</td>
<td>40</td>
</tr>
<tr>
<td>multiple</td>
<td>42</td>
</tr>
<tr>
<td>default initialization</td>
<td></td>
</tr>
<tr>
<td>variables</td>
<td>41</td>
</tr>
<tr>
<td>default values</td>
<td>105</td>
</tr>
<tr>
<td>defer</td>
<td>91</td>
</tr>
<tr>
<td>deinitializer</td>
<td></td>
</tr>
<tr>
<td>modules</td>
<td>98</td>
</tr>
<tr>
<td>deinitializer</td>
<td></td>
</tr>
<tr>
<td>classes</td>
<td>166</td>
</tr>
<tr>
<td>records</td>
<td>170</td>
</tr>
<tr>
<td>delete</td>
<td></td>
</tr>
<tr>
<td>class unmanaged instances</td>
<td>165</td>
</tr>
<tr>
<td>illegal for records</td>
<td>175</td>
</tr>
<tr>
<td>derived class</td>
<td>150</td>
</tr>
<tr>
<td>distributions (see also domain maps, distributions)</td>
<td>265</td>
</tr>
<tr>
<td>dmap value</td>
<td>266</td>
</tr>
<tr>
<td>dmapped clause</td>
<td>266</td>
</tr>
<tr>
<td>domain maps</td>
<td>265</td>
</tr>
<tr>
<td>distribution</td>
<td>265</td>
</tr>
<tr>
<td>dmap value</td>
<td>266</td>
</tr>
<tr>
<td>dmapped clause</td>
<td>266</td>
</tr>
<tr>
<td>domain assignment</td>
<td>268</td>
</tr>
<tr>
<td>for arrays</td>
<td>267</td>
</tr>
<tr>
<td>for domain values</td>
<td>267</td>
</tr>
<tr>
<td>layout</td>
<td>265</td>
</tr>
<tr>
<td>domain maps for domain types</td>
<td>265</td>
</tr>
<tr>
<td>domain-based slicing</td>
<td>203</td>
</tr>
<tr>
<td>domains</td>
<td>193</td>
</tr>
<tr>
<td>#</td>
<td>203</td>
</tr>
<tr>
<td>+</td>
<td>207</td>
</tr>
<tr>
<td>-</td>
<td>207</td>
</tr>
<tr>
<td>adding indices</td>
<td>204</td>
</tr>
<tr>
<td>alignment</td>
<td>202</td>
</tr>
<tr>
<td>as arguments</td>
<td>200</td>
</tr>
<tr>
<td>assignment</td>
<td>201</td>
</tr>
<tr>
<td>association with arrays</td>
<td>216</td>
</tr>
<tr>
<td>associative</td>
<td>196</td>
</tr>
<tr>
<td>default values</td>
<td>197</td>
</tr>
<tr>
<td>initialization</td>
<td>197</td>
</tr>
<tr>
<td>literals</td>
<td>197</td>
</tr>
<tr>
<td>values</td>
<td>197</td>
</tr>
<tr>
<td>clear</td>
<td>204</td>
</tr>
<tr>
<td>common methods</td>
<td>204</td>
</tr>
<tr>
<td>count operator</td>
<td>205</td>
</tr>
<tr>
<td>dim</td>
<td>206</td>
</tr>
<tr>
<td>dims</td>
<td>206</td>
</tr>
<tr>
<td>dist</td>
<td>204</td>
</tr>
<tr>
<td>expand</td>
<td>206</td>
</tr>
<tr>
<td>exterior</td>
<td>206</td>
</tr>
</tbody>
</table>
Index

else, 77 83
enum (generic type), 227
enumerated (generic type), 227
enumerated types, 34
abstract, 35
congrete, 35
iterating, 35
semi-concrete, 35
size, 35
error handling, 125
ersors, 125
catch, 126
handling, 125
throwing, 125
try, 126
exchange (atomic var), 242
execution environment, 262
explicit return type, 113
exploratory programming, 99
expression statement, 80
expressions, 55
associativity, 60
binary operator, 62
call, 57
cast, 58
conditional, 77
enumeration constant, 56
for, 78
filtering predicates, 78
forall, 252
and conditional expressions, 253
filtering, 253
semantics, 252
syntax, 252
zipper iteration, 252
if-then-else, 77
indexing, 57
literal, 56
lvalue, 58
member access, 57
operator, 62
parenthesized, 56
precedence, 60
reduction, 258
scan, 259
statement, 80
type query, 57
variable, 56
fetchAdd (atomic var), 243
fetchAnd (atomic var), 243
fetchOr (atomic var), 243
fetchSub (atomic var), 243
fetchXor (atomic var), 243
field access, 171
class, 163
Index

fields
  class, 149
  generic, 228
  parameter, 229
  records, 168
  type alias, 229
  variable and constant, without types, 230

for
  for loops, 86
  filtering predicates, 78

forall
  forall (see also statements, forall), 250
  forall expressions (see also expressions, forall), 252
  forall intents, 253

formal arguments
  array, 226
  defaults, 105
  generic, 224
  naming, 105
  partially concrete, 225
  partially generic, 225
  with queried types, 224
  without types, 223

function calls
  function calls, 57
  functions, 101

functions
  actual arguments, 101
  arguments
    defaults, 105
    formal, 105
    intents, 106
    named, 105
  arrays
    predefined, 216
    as parameters, 112
    as types, 112
    call site, 101
    candidates, 115
  const ref keyword and, 111
  default argument values, 105
  formal arguments, 101
  generic, 222
  iterators, 218
  lvalues, 110
  main, 99
  method, 101
  most specific, 116
  named arguments, 105
  nested, 114
  operator, 101
  overloading, 114
  parameter function, 112

procedure
  procedure, 101
  procedure definition, 102

ref keyword and
  ref keyword and, 110

resolution
  legal argument mapping, 116
  most specific, 116
  valid mapping, 115
  return intent, 110
  return intent overloads, 111
  return types, 113
  explicit, 113
  implicit, 113

tuples
  predefined, 143
  type functions, 112
  varargs, 109
  variable number of arguments, 109
  visible, 115
  where, 114
  without parentheses, 104

generic functions and special versions, 233

generic specialization, 233
generics, 222
classes, 228
examples
  stack, 234
  fields, 228
  function visibility, 226
  functions, 222
  initializers
    compiler-generated, 231
    user-defined, 231
  instantiated type, 230
  methods, 231
  records, 228
type constructor, 230
types, 227
getter method
class, 164

here, 263

identifiers, 24
if, 77
imaginary, 33
implicit modules, 95
implicit return type, 113
in (intent), 106
indexing, 57
arrays, 210
associative arrays, 211
Index

rectangular arrays, 211
inheritance, 150
initialization
  arrays, 208
  modules, 98
  record, 169
  sparse domains, 199
initializer-inheritance, 160
initializers, 152
  compiler-generated, 159
    for generic classes or records, 231
  initializer-inheritance, 160
  type constructors, 230
  user-defined, 153
    for generic classes or records, 231
inout (intent), 107
input/output, 236
instance methods, 121
int, 33
integral (generic type), 227
ints, 106
  abstract, 108
  array default, 109
  concrete, 106
  const, 108
  const in, 107
  const ref, 107
  const ref return, 111
  default, 108
  default for owned, 109
  default for shared, 109
  in, 106
  inout, 107
  out, 106
  param, 223
  record this default, 109
  ref, 107
  ref return, 110
  type, 222
interoperability, 277
  argument passing, 283
C structs
  external, 280
C types
  external, 280
  standard, 279
Chapel functions
  calling, 278
Chapel procedures
  calling, 283
external C types, 280
external functions
  calling, 277
  opaque types, 281
  overview, 277
  shared data, 282
  shared procedures, 282
  sharing, 279
  standard C types, 279
IO
  operator, 82
  statement, 82
irregular domains
  methods, 207
isFull (sync var), 241
iteration
  array, 212
  domain, 200
  tuple, 137
  zipper, 87
iterators, 218
  and arrays, 219
  and generics, 220
  calls, 219
  definition, 218
  in for loops, 219
  in forall loops, 219
  parallel, 221
  promotion, 220
  recursive, 220
  yield, 219
keywords, 24
label, 88
language overview, 20
language principles, 20
  general parallel programming, 20
  generic programming, 22
  locality-aware programming, 21
  object-oriented programming, 21
layouts (see also domain maps, layouts), 265
leading the execution of a loop, 251
let, 77
lexical structure, 23
  braces, 30
  brackets, 30
  case sensitivity, 23
  comments, 23
  identifiers, 24
  keywords, 24
  literals, 25
Index

operator, 29
parentheses, 30
punctuation, 29
tokens, 23
white space, 23
literal expressions, 56
literals
  primitive type, 25
local, 261
locale, 261
Locales, 262
locales, 261
  callStackSize, 261
  here, 263
  id, 261
  local, 261
  maxTaskPar, 261
  methods, 261
  name, 262
  numPUs, 262
  physicalMemory, 262
  remote, 261
loops
  forall (see also statements, forall), 250
lvalues, 58
main, 22
mapped
domain maps, 265
member access, 57
memory consistency model, 270
  examples, 275
  non-sequentially consisten atomic operations, 273
  sequential consistency for data-race-free programs, 270
  unordered memory operations, 274
method calls, 120
methods, 120
  base class
    overriding, 151
    calling, 120
    class
      getter, 164
      classes, 149
      generic, 231
      indexing, 123
      instance, 121
      iterating, 123
      primary, 120
      receiver, 120
      records, 168
    secondary, 120
    these, 123
    this, 123
    type, 121
module, 94
modules, 22
  access, 96
  and files, 95
  definitions, 94
  deinitialization, 98
  deinitialization order, 100
  implicit, 95
  initialization, 98
  initialization order, 100
  nested, 95
  prototype, 94
  qualified naming, 97
  sub-modules, 95
  top-level, 94
  using, 89
  97
named arguments, 105
nested classes, 150
nested function, 114
nested records, 169
new
  classes, 150
  new, 152
  nil, 148
  not provided for records, 174
noinit, 42
  variables, 42
notation, 15
nothing, 32
numeric (generic type), 227
numLocales, 262
object, 150
objects, 144
on, 264
opaque domains
types, 196
operators, 72
  !=, 74
  != (string), 75
  #, 76
  # (domain), 203
  # (on arrays), 214
  # (range), 186
  &., 69
  &&, 72
| *       | 66 |
| *+     | 68 |
| +      | 64 |
| + (string) | 70 |
| + (unary) | 63 |
| -      | 65 |
| - (unary) | 63 |
| /      | 67 |
| : (cast) | 58 |
| <      | 73 |
| <=     | 73 |
| ==     | 74 |
| == (string) | 75 |
| >      | 75 |
| >=     | 74 |
| ? (type query) | 57 |
| %      | 68 |
| ~      | 69 |
| ^      | 70 |
| addition | 64 |
| align | 76 |
| align (domain) | 202 |
| align (range) | 185 |
| arithmetic | 62 |
| range | 187 |
| assignment | 81 |
| compound | 81 |
| simple | 81 |
| associativity | 60 |
| binary | 62 |
| bitwise | 69 |
| and | 69 |
| complement | 69 |
| exclusive or | 70 |
| or | 74 |
| by | 76 |
| by (domain) | 202 |
| by (range) | 184 |
| compound assignment | 81 |
| concatenation | 81 |
| string | 76 |
| division | 67 |
| equality | 74 |
| exponentiation | 68 |
| greater than | 73 |
| greater than or equal | 74 |
| IO | 82 |
| less than | 73 |
| less than or equal | 73 |
| let | 77 |
| lexical structure | 29 |
| logical | 71 |
| and | 72 |
| not | 71 |
| or | 72 |
| modulus | 68 |
| multiplication | 66 |
| negation | 63 |
| overloading | 114 |
| precedence | 60 |
| procedure | 101 |
| range | 20 |
| count | 76 |
| relational | 73 |
| shift | 71 |
| simple assignment | 81 |
| string concatenation | 76 |
| subtraction | 65 |
| swap | 82 |
| tuple | 142 |
| binary | 142 |
| relational | 142 |
| unary | 142 |
| unary | 62 |
| or (atomic var) | 243 |
| organization | 17 |
| otherwise | 84 |
| out (intent) | 106 |
| overloading | 114 |
| overloading functions (see also functions, overloading) | 114 |
| overloading operators (see also operators, overloading) | 114 |
| overview | 20 |
| parallel iterators | 221 |
| parallelism | 250 |
| data | 250 |
| task | 237 |
| param | 44 87 |
| parameter function | 112 |
| parameters | 44 |
| configuration | 45 |
| in classes or records | 229 |
| predefined functions | 207 |
| + (domain) | 207 |
| - (domain) | 207 |
| add (atomic var) | 243 |
| aligned | 189 |
| alignedHigh | 189 |
Index

alignedLow, 189
alignHigh (range), 191
alignLow (range), 191
alignment, 189
and (atomic var), 243
arrays, 216
boundedType, 188
boundsCheck, 190
callStackSize, 261
clear, 204
clear (atomic bool), 243
compareAndSwap (atomic var), 243
dim (domain), 206
dims (domain), 206
dist, 204
domain (array), 217
eltType (array), 216
exchange (atomic var), 242
expand (domain), 206
expand, 191
exterior (domain), 206
exterior (range), 191
fetchAdd (atomic var), 243
fetchAnd (atomic var), 243
fetchOr (atomic var), 243
fetchSub (atomic var), 243
fetchXor (atomic var), 243
first, 189
hasFirst, 189
hasHighBound, 189
hasLast, 189
hasLowBound, 189
high, 190
high (domain), 206
id, 261
ident, 190
idxType, 188
indexOrder, 190
interior (domain), 206
interior (range), 191
isAmbiguous, 190
isFull (sync var), 241
isIrregularDom, 205
isOpaqueDom, 205
isRectangularDom, 205
isSparseDom, 205
isTuple, 143
isTupleType, 143
last, 190
length, 190
locale, 261
low, 190
low (domain), 206
max, 143
maxTaskPar, 261
member, 191
member (domain), 205
min, 143
name, 262
numElements (array), 217
numIndices (domain), 205
numPUs, 262
offset (range), 192
or (atomic var), 243
physicalMemory, 262
rank (array), 217
rank (domain), 206
read (atomic var), 242
readFE (sync var), 240
readFF (sync var), 240
readXX (sync var), 241
requestCapacity, 207
reset (sync var), 241
reshape (array), 217
size, 143
size (array), 217
size (domain), 206
size (enum), 35
stridable, 188
stridable (domain), 207
stride, 190
stride (domain), 207
sub (atomic var), 243
testAndSet (atomic bool), 243
translate (domain), 207
translate (range), 192
tuples, 143
waitFor (atomic var), 243
write (atomic var), 242
writeEF (sync var), 241
writeFF (sync var), 241
writeXF (sync var), 241
xor (atomic var), 243

primary methods, 120, 149
proc, 102
procedure, 101
procedures
definition, 102
program execution, 98
program initialization, 98
promotion, 255
arrays, 215
Index

default arguments, 256
domain, 200
iterator, 220
range, 184
zipper iteration, 256

range-based slicing, 203
ranges, 178
#., 186

align, 185
align operator, 185
aligned, 189
aligned high bound, 179
aligned integer, 179
aligned low bound, 179
alignedHigh, 189
alignedLow, 189
alignHigh, 191
alignLow, 191
alignment, 178
ambiguous, 178, 179
alignment, 189

arithmetic operators, 187
assignment, 183
bounded, 180
boundedHigh, 180
boundedLow, 180
boundedNone, 180
boundedType, 180
boundedType, 188
boundsCheck, 190
by operator, 184
comparisons, 183
concepts, 178
count operator, 186
default values, 182
empty, 179
expand, 191
exterior, 191
first, 189
first index, 179
hasFirst, 189
hasHighBound, 189
hasLast, 189
hasLowBound, 189
high, 190
high bound, 178
ident, 190
idxType, 180
idxType, 188
indexOrder, 190

interior, 191
isAmbiguous, 190
iterable, 179
iteration, 183
zippered, 183
last, 190
last index, 179
length, 190
literals, 181
low, 190
low bound, 178
member, 191
offset, 192
operations, 182
operators, 184
other queries, 190
predefined functions, 188
promotion, 184
properties, 189
represented sequence, 178
decreasing, 179
increasing, 179
sequence, 178
size, 190
slicing, 188
stridable, 180
stridable, 188
stride, 178
stride, 190
stride type, 180
strided, 184
transformations, 191
translate, 192
type accessors, 188
types, 180
values, 181

rank-change slicing, 203
read (atomic var), 242
readFE (sync var), 240
readFF (sync var), 240
readXX (sync var), 241
real, 13
receiver, 121
class, 163
record, 164
record (generic type), 227
records, 167
! =, 173
==, 173
allocation, 169
arguments, 171

Index

assignment, 173, 174
copy initialization, 172
declarations, 167

differences with classes, 174
deinitializer, 170
delete illegal, 175
differences with classes, 173
equality, 173
field access, 171
field access, 171
fields, 168
generic, 168
generics, 228
getters, 172
inequality, 173
initialization, 172
method calls, 168
methods, 168
nested, 169
record types, 168
types, 168
variable declarations, 169

rectangular array literals, 209
rectangular arrays
indexing, 211
rectangular domains (see also domains, rectangular), 194
reduction expressions, 258
reductions, 258
ref, 45
ref (intent), 107
ref (return intent), 110
regular domains
methods, 206
remote, 261
reset (sync var), 241
return (see also statements, return), 113
returning
array, 215

scan expressions, 259
scans, 258
scope, 14
secondary methods, 120, 149
select, 84
serial, 248
shadow variables, 253, 254
shadowing
base class fields, 151
single, 238

slicing
array, 213
arrays

rectangular, 214
domain-based, 203
domains, 202
range-based, 203
rank-change, 203
sparse domains
default value, 199
initialization, 199
literals
lack thereof, 199
specific instantiations, 233

statements
assignment, 81
atomic, 249
begin, 238
break, 88
cobegin, 244
coforall, 244
conditional, 83
continue, 88
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
defer, 91
expression, 80
for, 86
forall, 250
forall intents, 253
semantics, 251
shadow variables, 253, 254
syntax, 250
task-private variables, 254
zipper iteration, 251

IO, 82
jumps, 88
label, 88
on, 264
otherwise, 84
param for, 87
return, 113
return intent, 110
return type, 113
select, 84
serial, 248
swap, 82
sync, 247
use, 89
when, 84
while, 84
string, 84
sub (atomic var), 243
subdomains, 198
Index

simple, 198
  default values, 198
  types, 198
  values, 198
sparse, 199
  types, 199
  values, 199
types
  simple, 198
swap
  operator, 82
  statement, 82
sync, 238, 247
synchronization, 237
synchronization types
  actual arguments, 240
  formal arguments, 240
synchronization variables
  predefined methods on, 240
  single, 238
  sync, 238
task creation, 237
task function, 237
task intents, 245
task parallelism, 237
  task creation, 237
  task function, 237
  task functions, 245
  task intents, 245
task-private variables, 254
testAndSet (atomic bool), 243
then, 77, 83
these, 123
this, 121, 123
throw, 125
throws, 125
try, 126
tuples, 135
  assignment, 137
  assignments grouped as, 138
  destructuring, 138
  expanding in place, 141
  formal arguments grouped as, 140
  homogeneous, 135
  indexing, 137
  indices grouped as, 140
  isTuple, 143
  isTupleType, 145
  iteration, 137
  max, 143
min, 143
  omitting components, 138, 141
  operators, 142
  predefined functions, 143
  size, 143
  types, 135
  values, 136
  variable declarations grouped as, 139
type aliases
  in classes or records, 229
type inference, 42
  local, 42
  of return types, 113
type methods, 121
types, 31
  * tuples, 135
  aliases, 39
  associative domains, 196
  bool, 33
  bytes, 34
  complex, 34
  datapixel, 38
domains
  domain maps for, 265
  sparse, 199
  enumerated, 34
  using, 89
generic, 227
imaginary, 33
int, 33
locale, 261
nothing, 32
opaque domains, 196
primitive, 31
range, 180
real, 33
records, 168
rectangular domains, 194
string, 34
structured, 36
subdomains
  simple, 198
  sparse, 199
synchronization, 38
tuple, 135
uint, 33
unions, 176
void, 32
uint, 33
union, 176
Index

union types, 176
unions, 176
assignment, 177
fields, 176
use, 89
user-defined compiler diagnostics, 233
user-defined compiler errors, 233
user-defined compiler warnings, 233
user-defined initializers, 153
values
domains
domain maps for, 267
sparse, 199
subdomains
simple, 198
sparse, 199
tuple, 136
variable declarations
remote, 264
variables, 40
configuration, 45
declarations, 40
multiple, 42
default initialization, 41
default values, 41
in classes or records, 230
local, 44
module level, 44
records, 169
ref, 45
void, 32
waitFor (atomic var), 243
when, 84
where, 114
implicit, 225
while, 84
while loops, 84
white space, 23
whole array assignment, 257
whole array operations, 257
write (atomic var), 242
writeEF (sync var), 241
writeFF (sync var), 241
writeXF (sync var), 241
xor (atomic var), 243
yield, 219
zipper iteration, 87