Chapel: Language Basics

Steve Deitz

Cray Inc.

The HelloWorld Program

- Fast Prototyping

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

- Structured Programming

```
def main() {
  writeln("hello, world");
}
```

- Production-Level

```
module HelloWorld {
  def main() {
    writeln("hello, world");
  }
}
```

Chapel Stereotypes and Generalizations

- Syntax
  - Basics from C, C#, C++, Java, Ada, Perl, ...
  - Specifics from many other languages
- Semantics
  - Imperative, block-structured, array paradigms
  - Optional object-oriented programming (OOOP)
  - Static typing for performance and safety
  - Elided types for convenience and generic coding
- Features
  - No pointers and few references
  - No compiler-inserted array temporaries

Chapel Influences

ZPL, HPF: data parallelism, index sets, distributed arrays

CRAY MTA C/Fortran: task parallelism, synchronization

CLU, Ruby, Python: iterators

ML, Scala, Matlab, Perl, Python, C#: latent types

Java, C#: OOP, type safety

C++: generic programming/templates

Outline

- High-Level Comments
- Elementary Concepts
  - Lexical structure
  - Types, variables, and constants
  - Input and output
- Data Structures and Control
- Miscellaneous

Lexical Structure

- Comments

```
/* standard
   C-style */
// standard C++ style
```

- Identifiers
  - Composed of A-Z, a-z, 0-9, _, and $
  - Starting with A-Z, a-z, and _
- Case-sensitive
- Whitespace-aware
  - Composed of spaces, tabs, and linefeeds
  - Separates tokens and ends //-comments

Primitive Types

Type	Description	Default	Bit Width	Supported Bit Widths
bool	logical value	false	impl-dep	8, 16, 32, 64
int	signed integer	0	32	8, 16, 32, 64
uint	unsigned integer	0	32	8, 16, 32, 64
real	real floating point	0.0	64	32, 64
imag	imaginary floating point	0.0i	64	32, 64
complex	complex floating points	0.0 + 0.0i	128	64, 128
string	character string	""	NA	NA

- Syntax

```
primitive-type:
  type-name [( bit-width )]
```

- Examples

```
int(64)   // 64-bit int
real(32)  // 32-bit real
uint      // 32-bit uint
```

Variables, Constants, and Parameters

- Syntax

```
declaration:
  var identifier [: type] [= init-expr]
  const identifier [: type] [= init-expr]
  param identifier [: type] [= init-expr]
```

- Semantics
  - Const-ness: not, at runtime, at compile-time
  - Omitted *type*, type is inferred from *init-expr*
  - Omitted *init-expr*, value is assigned default for type
- Examples

```
var count: int;
const pi: real = 3.14159;
param debug = true;
```

Config Declarations

- Syntax

```
config-declaration:
  config declaration
```

- Semantics
  - Supports command-line overrides
  - Requires global-scope declaration
- Examples

```
config param intSize = 32;
config const start: int(intSize) = 1;
config var epsilon = 0.01;
```

```
chpl -sintSize=16 -o a.out myProgram.chpl
a.out --start=2 --epsilon=0.001;
```

Input and Output

- Input
  - read(expr-list): reads values into the arguments
  - read(type-list): returns values read of given types
  - readln variant: also reads through new line
- Output
  - write(expr-list): writes arguments
  - writeln variant: also writes new line
- Support for arbitrary types (including user-defined)
- File and string I/O via method variants of the above

Outline

- High-Level Comments
- Elementary Concepts
- Data Structures and Control
  - Ranges
  - Arrays
  - For loops
  - Traditional constructs
- Miscellaneous

Range Values

- Syntax

```
range-expr:
  [low] .. [high] [by stride]
```

- Semantics
  - Regular sequence of integers

    stride > 0: *low, low+stride, low+2*stride, ...* ≤ *high*

    stride < 0: *high, high+stride, high+2*stride, ...* ≥ *low*
  - Default *stride* = 1, default *low* or *high* is unbounded
- Examples

```
1..6 by 2        // 1, 3, 5
1..6 by -1       // 6, 5, 4, 3, 2, 1
3.. by 3         // 3, 6, 9, 12, ...
```

Array Types

- Syntax

```
array-type:
  [ index-set-expr ] type
```

- Semantics
  - Stores an element of *type* for each index in set
- Examples

```
var A: [1..3] int,           // 3-element array of ints
    B: [1..3, 1..5] real, // 2D array of reals
    C: [1..3][1..5] real; // array of arrays of reals
```

Much more on arrays in data parallelism part

For Loops

- Syntax

```
for-loop:
  for index-expr in iterator-expr { stmt-list }
```

- Semantics
  - Executes loop body once per loop iteration
  - Indices in index-expr are new variables
- Examples

```
var A: [1..3] string = ("DO", "RE", "MI");

for i in 1..3 do write(A(i));        // DOREMI
for a in A { a += "LA"; write(a); } // DOLARELAMILA
```

Zipper "()" and Tensor "[]" Iteration

- Syntax

```
tensor-for-loop:
  for index-expr in [ iterator-expr-list ] { stmt-list }

zipper-for-loop:
  for index-expr in ( iterator-expr-list ) { stmt-list }
```

- Semantics
  - Tensor iteration is over all pairs of yielded indices
  - Zipper iteration is over all yielded indices pair-wise
- Examples

```
for i in [1..2, 1..2] do  // (1,1), (1,2), (2,1), (2,2)

for i in (1..2, 1..2) do  // (1,1), (2,2)
```

Traditional Control

- Conditional statements

```
if cond then computeA() else computeB();
```

- While loops

```
while cond {
  compute();
}
```

- Select statements

```
select key {
  when value1 do compute1();
  when value2 do compute2();
  otherwise compute3();
}
```

Outline

- High-Level Comments
- Elementary Concepts
- Data Structures and Control
- Miscellaneous
  - Functions and iterators
  - Records and classes
  - Generics

Function Examples

- Example to compute the area of a circle

```
def area(radius: real)
  return 3.14 * radius**2;

writeln(area(2.0));    // 12.56
```

- Example of function arguments

```
def writeCoord(x: real = 0.0, y: real = 0.0) {
  writeln("(", x, ", ", y, ")");
}

writeCoord(2.0);      // (2.0, 0.0)
writeCoord(y=2.0);    // (0.0, 2.0)
```

What is an Iterator?

- An abstraction for loop control
  - Yields (returns) indices for each iteration
  - Otherwise, like a function
- Example

```
def string_chars(s: string) {
  var i = 1, limit = length(s);
  while i <= limit {
    yield s.substring(i);
    i += 1;
  }
}

for c in string_chars(s) do ...
```

Iterator Advantages

- Separation of concerns
  - Loop logic is abstracted from computation
- Efficient implementations
  - When the values cannot be pre-computed
    - Memory is insufficient
    - Infinite or cyclic
    - Side effects
  - When not all of the values need to be used

Records

- User-defined data structures
  - Contain variable definitions (fields)
  - Contain function definitions (methods)
  - Value-semantics (assignment copies fields)
  - Similar to C++ classes
- Example

```
record circle { var x, y, radius: real; }
var c1, c2: circle;
c1.x = 1.0; c1.y = 1.0; c1.radius = 2.0;
c2 = c1; // copy of value
```

Classes

- Reference-based records
  - Reference-semantics (assignment aliases)
  - Dynamic allocation
  - OOP-capable
  - Similar to Java classes
- Example

```
class circle { var x, y, radius: real; }
var c1, c2: circle;
c1 = new circle(x=1.0, y=1.0, radius=2.0);
c2 = c1; // c2 is an alias of c1
```

Method Examples

Methods are functions associated to data.

```
def circle.area()
  return 3.14 * this.radius**2;

writeln(c1.area());
```

note: this is implicit

Methods can be defined for any type.

```
def int.square
  return this**2;

writeln(5.square);
```

(parentheses optional)

Generic Functions

Generic functions are replicated for each unique call site. They can be defined by explicit type and param arguments:

```
def foo(type t, x: t) { ...

def bar(param bitWidth, x: int(bitWidth)) { ...
```

Or simply by eliding an argument type (or type part):

```
def goo(x, y) { ...

def sort(A: []) { ...
```

Generic Types

Generic types are replicated for each unique instantiation. They can be defined by explicit type and param fields:

```
class Table { param numFields: int; ...

class Matrix { type eltType; ...
```

Or simply by eliding a field type (or type part):

```
record Triple { var x, y, z; }
```

Questions?

- High-Level Comments
- Elementary Concepts
  - Lexical structure
  - Types, variables, and constants
  - Input and output
- Data Structures and Control
  - Ranges
  - Arrays
  - For loops
  - Traditional constructs
- Miscellaneous
  - Functions and iterators
  - Records and classes
  - Generics