Chapel: Language Basics

Steve Deitz
Cray Inc.
The HelloWorld Program

- **Fast Prototyping**
  ```chapel```
  writeln("hello, world");
  ```

- **Structured Programming**
  ```chapel```
  def main() {
    writeln("hello, world");
  }
  ```

- **Production-Level**
  ```chapel```
  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
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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Default</th>
<th>Bit Width</th>
<th>Supported Bit Widths</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>logical value</td>
<td>false</td>
<td>impl-dep</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td>int</td>
<td>signed integer</td>
<td>0</td>
<td>32</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td>uint</td>
<td>unsigned integer</td>
<td>0</td>
<td>32</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td>real</td>
<td>real floating point</td>
<td>0.0</td>
<td>64</td>
<td>32, 64</td>
</tr>
<tr>
<td>imag</td>
<td>imaginary floating point</td>
<td>0.0i</td>
<td>64</td>
<td>32, 64</td>
</tr>
<tr>
<td>complex</td>
<td>complex floating points</td>
<td>0.0 + 0.0i</td>
<td>128</td>
<td>64, 128</td>
</tr>
<tr>
<td>string</td>
<td>character string</td>
<td>&quot;&quot;</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Syntax

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

### Examples

```plaintext
int(64) // 64-bit int
real(32) // 32-bit real
uint     // 32-bit uint
```
Variables, Constants, and Parameters

- **Syntax**

  \[
  \text{declaration:}
  \]
  \[
  \begin{align*}
  \text{var} & \quad \text{identifier} \quad [: \quad \text{type}] \quad [= \quad \text{init-expr}] \\
  \text{const} & \quad \text{identifier} \quad [: \quad \text{type}] \quad [= \quad \text{init-expr}] \\
  \text{param} & \quad \text{identifier} \quad [: \quad \text{type}] \quad [= \quad \text{init-expr}] \\
  \end{align*}
  \]

- **Semantics**

  - Const-ness: not, at runtime, at compile-time
  - Omitted \textit{type}, type is inferred from \textit{init-expr}
  - Omitted \textit{init-expr}, value is assigned default for type

- **Examples**

  \[
  \begin{align*}
  \text{var} & \quad \text{count}: \quad \text{int}; \\
  \text{const} & \quad \text{pi}: \quad \text{real} \quad = \quad 3.14159; \\
  \text{param} & \quad \text{debug} \quad = \quad \text{true}; \\
  \end{align*}
  \]
Config Declarations

- **Syntax**

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

- **Semantics**

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

- **Examples**

  ```plaintext
  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;
  ```
## Basic Operators and Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
<th>Description</th>
<th>Overloadable</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>:</code></td>
<td>left</td>
<td>cast</td>
<td>no</td>
</tr>
<tr>
<td><code>**</code></td>
<td>right</td>
<td>exponentiation</td>
<td>yes</td>
</tr>
<tr>
<td><code>! ~</code></td>
<td>right</td>
<td>logical and bitwise negation</td>
<td>yes</td>
</tr>
<tr>
<td><code>* / %</code></td>
<td>left</td>
<td>multiplication, division and modulus</td>
<td>yes</td>
</tr>
<tr>
<td><code>unary + -</code></td>
<td>right</td>
<td>positive identity and negation</td>
<td>yes</td>
</tr>
<tr>
<td><code>+ -</code></td>
<td>left</td>
<td>addition and subtraction</td>
<td>yes</td>
</tr>
<tr>
<td><code>&lt;&lt; &gt;&gt;</code></td>
<td>left</td>
<td>shift left and shift right</td>
<td>yes</td>
</tr>
<tr>
<td><code>&lt;= &gt;= &lt; &gt;</code></td>
<td>left</td>
<td>ordered comparison</td>
<td>yes</td>
</tr>
<tr>
<td><code>== !=</code></td>
<td>left</td>
<td>equality comparison</td>
<td>yes</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>left</td>
<td>bitwise/logical and</td>
<td>yes</td>
</tr>
<tr>
<td><code>^</code></td>
<td>left</td>
<td>bitwise/logical xor</td>
<td>yes</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>left</td>
<td>bitwise/logical or</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>left</td>
<td>short-circuiting logical and</td>
<td>via isTrue</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
<td>left</td>
</tr>
</tbody>
</table>
## Assignments

<table>
<thead>
<tr>
<th>Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>simple assignment</td>
</tr>
<tr>
<td>+= -= *= /= %= *= &amp;=</td>
<td>= ^= &amp;&amp;</td>
</tr>
<tr>
<td></td>
<td>(e.g., ( x += y ); is equivalent to ( x = x + y );)</td>
</tr>
<tr>
<td>&lt;==&gt;</td>
<td>swap</td>
</tr>
</tbody>
</table>
Implicit Conversions (Coercions)

<table>
<thead>
<tr>
<th>Type</th>
<th>Valid Target Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>int(32)</td>
<td>int(64), real(64), complex(128), string</td>
</tr>
<tr>
<td>int(64)</td>
<td>real(64), complex(128), string</td>
</tr>
<tr>
<td>uint(32)</td>
<td>int(64), uint(64), real(64), complex(128), string</td>
</tr>
<tr>
<td>uint(64)</td>
<td>real(64), complex(128), string</td>
</tr>
<tr>
<td>real(32)</td>
<td>real(64), complex(64), complex(128), string</td>
</tr>
<tr>
<td>real(64)</td>
<td>complex(128), string</td>
</tr>
</tbody>
</table>

- **Notes**
  - No loss of information (with a few *exceptions*)
  - Real values do not coerce to integers (as in C)

- **Examples**

```chapel
const threePointZero: real = 3;
const c = 1.0 + 2.0i; // uses + over complex
```
Explicit Conversions (Casts)

- **Syntax**
  
  \[
  \text{cast-expr:}
  \begin{align*}
  \text{expr} & : \text{type} \\
  \end{align*}
  \]

- **Semantics**
  - Converts type of `expr` to `type`
  - Supported between all primitive types

- **Examples**
  
  ```chapel
  const three = pi:int;
  const c = (1.0, 2.0):complex;
  ```
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
  - Tuples
  - Ranges
  - Arrays
  - For loops
  - Traditional constructs
- Miscellaneous
Tuple Values

- Syntax

```
tuple-expr:  
  ( component-list )

component-list:
  expr , expr
  expr , component-list
```

- Semantics

- Light-weight first-class data structure

- Examples

```
var i3: (int, int, int) = (0, 0, 0);
var i3_2: 3*int = (0, 0, 0);
var triple: (int, string, real) = (1, "two", 3.0);
```
Range Values

• Syntax

\[
\text{range-expr:} \\
[low] .. [high] [by stride]
\]

• Semantics

• Regular sequence of integers

\[\text{stride} > 0: \text{low}, \text{low}+\text{stride}, \text{low}+2*\text{stride}, \ldots \leq \text{high}\]

\[\text{stride} < 0: \text{high}, \text{high}+\text{stride}, \text{high}+2*\text{stride}, \ldots \geq \text{low}\]

• Default \text{stride} = 1, default \text{low} or \text{high} is unbounded

• Examples

\[
\begin{align*}
1..6 \text{ by 2} & \quad // \quad 1, 3, 5 \\
1..6 \text{ by } -1 & \quad // \quad 6, 5, 4, 3, 2, 1 \\
3.. \text{ by 3} & \quad // \quad 3, 6, 9, 12, \ldots 
\end{align*}
\]
Array Types

• Syntax

\[
\text{array-type:} \quad [\text{index-set-exp}] \text{type}
\]

• Semantics

• Stores an element of type for each index in set

• Examples

\[
\begin{align*}
\text{var A: } & [1..3] \text{ int, } & \text{ // 3-element array of ints} \\
\text{B: } & [1..3, 1..5] \text{ real, } & \text{ // 2D array of reals} \\
\text{C: } & [1..3][1..5] \text{ real; } & \text{ // array of arrays of reals}
\end{align*}
\]

Much more on arrays in data parallelism part
For Loops

• Syntax

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

• Semantics
  • Executes loop body once per loop iteration
  • Indices in index-exp are new variables

• Examples

```plaintext
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**

  ```chapel
tensor-for-loop:
  for index-exp in [ iterator-exp-list ] { stmt-list }
  
  zipper-for-loop:
  for index-exp in ( iterator-exp-list ) { stmt-list }
  ```

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

- **Examples**

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

- While loops
  
  ```chapel
  while cond {
    compute();
  }
  ```

- Select statements
  
  ```chapel
  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

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

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

• Example of function arguments

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

writeCoord(2.0);  // (2.0, 0.0)
writeCoord(y=2.0);  // (0.0, 2.0)
```
An abstraction for loop control
• Yields (returns) indices for each iteration
• Otherwise, like a function

Example

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

```chapel
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

```chapel
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

```chapel
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
```
Methods are functions associated to data.

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

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

note: `this` is implicit

Methods can be defined for any type.

```chapel
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:

```chapel
def foo(type t, x: t) { ... }
def bar(param bitWidth, x: int(bitWidth)) { ... }
```

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

```chapel
def goo(x, y) { ... }
def sort(A: []) { ... }
```
Generic types are replicated for each unique instantiation. They can be defined by explicit type and param fields:

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

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

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

```chapel
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
  - Tuples
  - Ranges
  - Arrays
  - For loops
  - Traditional constructs
- Miscellaneous
  - Functions and iterators
  - Records and classes
  - Generics