Chapel: Language Basics

The Hello World Program

- Fast prototyping

```
writeln(“hello, world”);
```

- Production-grade

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

Characteristics of Chapel

- Syntax
  - Basics from C and Modula
  - Influences from many other languages
- Semantics
  - Imperative, block-structured
  - Optional object-oriented programming (OOP)
  - Elided types for convenience and generic coding
  - Static typing for performance and safety
- Design points
  - 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
  - Starting with A-Z, a-z, _, $
- Case-sensitive
- Whitespace-aware
  - Composed of spaces, tabs, and linefeeds
  - Separates tokens and ends //-comments

Primitive Types

Type	Description	Default Value	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	“”	N/A	N/A

- 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
  - Constness at runtime (**const**), at compile-time (**param**)
  - Omitted *init-expr*: value is assigned default for type
  - Omitted *type*: type is inferred from *init-expr*

- Examples

```
var count: int;           // initialized to 0
const pi: real = 3.14159;
param debug = true;       // inferred to be bool
```

Config Declarations

- Syntax

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

- Semantics
  - Supports command-line overrides
  - Must be declared at module (file) scope
- Examples

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

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

Variables Examples

- examples/primers/variables.chpl

Basic Operators and Precedence

Operator	Description	Associativity	Overloadable
`:`	cast	`left`	no
`**`	exponentiation	`right`	yes
`! ~`	logical and bitwise negation	`right`	yes
`* / %`	multiplication, division and modulus	`left`	yes
unary `+ -`	positive identity and negation	`right`	yes
`+ -`	addition and subtraction	`left`	yes
`<< >>`	shift left and shift right	`left`	yes
`<= >= < >`	ordered comparison	`left`	yes
`== !=`	equality comparison	`left`	yes
`&`	bitwise/logical and	`left`	yes
`^`	bitwise/logical xor	`left`	yes
`|`	bitwise/logical or	`left`	yes
`&&`	short-circuiting logical and	`left`	via `isTrue`
`||`	short-circuiting logical or	`left`	via `isTrue`

Assignments

Kind	Description
`=`	simple assignment

`+=  -=  *=  /=  %=`
`**=  &=  |=  ^=`
`&&=  ||=  <<=  >>=`
	compound assignment
	(*e.g.,* `x += y;` is equivalent to `x = x + y;`)

`<=>`	swap

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 all types (including user-defined)
- File and string I/O via method variants of the above

File I/O examples

- examples/primers/fileIO.chpl

Outline

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

Tuple Values

- Syntax

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

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

- Semantics
  - Light-weight first-class data structure
- Examples

```
var i3: (int, int, int) = (1, 2, 3);
var i3_2: 3*int = (4, 5, 6);
var triple: (int, string, real) = (7, “eight”, 9.0);
```

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, ...
```

Range Examples

- examples/primers/ranges.chpl

Array Types

- Syntax

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

- Semantics
  - Stores an element of *elt-type* for each index
- 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

Array Examples

- examples/primers/arrays.chpl

For Loops

- Syntax

```
for-loop:
  for index-expr in iteratable-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));         // DO RE MI
for a in A { a += “LA“; write(a); } // DOLA RELA MILA
```

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

- Syntax

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

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

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

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

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

Traditional Control

- Conditional statements

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

- While loops

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

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

- 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
  - Other basic language features

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 (generates) values for consumption
  - Otherwise, like a function
- Example

```
def string_chars(s: string) {
  for i in 1..length(s) do
    yield s.substring(i);
}

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

Iterator Examples

- examples/primers/iterators.chpl

Records

- Value-based objects
  - Value-semantics (assignment copies fields)
  - Contain variable definitions (fields)
  - Contain function definitions (methods)
  - 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 objects
  - Reference-semantics (assignment aliases)
  - Dynamic allocation
  - Dynamic dispatch
  - 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
delete c1;
```

Classes Examples

- examples/primers/classes.chpl

Method Examples

Methods are functions associated with types.

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

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

Methods can be defined for any type.

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

writeln(5.square());
```

Generic Functions

Generic functions 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 functions are replicated for each unique instantiation:

```
foo(int, x);    // copy of foo() with t==int
foo(string, x); // copy of foo() with t==string
goo(4, 2.2);    // copy of goo() with int and real args
```

Generic Types

Generic types 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; }
```

Generic types are replicated for each unique instantiation:

```
// copy of Table with 10 fields
var myT: Table(10);
// copy of Triple with x:int, y:int, z:real
var my3: Triple(int,int,real) = new Triple(1,2,3.0);
```

Generics Examples

- examples/primers/genericClasses.chpl

Other Basic Language Features

- Unions
- Enumerated types
- Range and domain by and # operators
- Expression forms of conditionals and loops
- Type select statements
- Function instantiation constraints (where clauses)
- Formal argument intents (in, out, inout, const)
- User-defined compiler warnings and errors

Future Directions

- Fixed length strings
- Binary I/O
- Parallel I/O
- Interoperability with other languages
- More advanced OO features

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
  - Other basic language features