Chapel: Base Language

"Hello World" in Chapel: Two Versions

- Fast prototyping

```
writeln("Hello, world!");
```

- "Production-grade"

```
module Hello {
  proc main() {
    writeln("Hello, world!");
  }
}
```

Characteristics of Chapel

- **Design points**
  - Identifying parallelism & locality is user's job, not compiler's
  - No compiler-inserted array temporaries
  - No pointers and limited aliases
  - Intentionally not an extension of an existing language

Chapel Influences

C, Modula: basic syntax

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

CRAY MTA C/Fortran: task parallelism, synchronization

CLU (see also Ruby, Python, C#)**:** iterators

Scala (see also ML, Matlab, Perl, Python, C#)**:** type inference

Java, C#: OOP, type safety

C++: generic programming/templates
(but with a different syntax)

Outline

- Introductory Notes
- Elementary Concepts
- Data Types and Control Flow
- Program Structure

Lexical Structure

- Comments

```
/* standard
   C style
   multi-line */

// standard C++ style single-line
```

- Identifiers:
  - Composed of A-Z, a-z, _, $, 0-9
  - Cannot start with 0-9
- Case-sensitive

Primitive Types

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

- Basic syntax

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

- Semantics
  - **var/const**: execution-time variable/constant
  - **param**: compile-time constant
  - No *init-expr* ⇒ initial value is the type's default
  - No *type* ⇒ type is taken from *init-expr*
- Examples

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

Config Declarations

- Syntax

```
config-declaration:
  config type-alias-declaration
  config declaration
```

- Semantics
  - Like normal, but supports command-line overrides
  - Must be declared at module/file scope
- Examples

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

```
% chpl myProgram.chpl -sintSize=64 -selementType=real
% a.out --start=2 --epsilon=0.00001
```

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

<table>
<tr><th>Kind</th><th>Description</th></tr>
<tr><td><code>=</code></td><td>simple assignment</td></tr>
<tr><td><code>+= -= *= /= %=</code>
<code>**= &= |= ^=</code>
<code>&&= ||= <<= >>=</code></td><td>compound assignment
(e.g., <code>x += y;</code> is equivalent to <code>x = x + y;</code>)</td></tr>
<tr><td><code><=></code></td><td>swap assignment</td></tr>
</table>

- Note: assignments are only supported at the statement level

Console Input/Output

- Output
  - **write**(expr-list): writes the argument expressions
  - **writeln**(…) variant: writes a linefeed after the arguments
- Input
  - **read**(*expr-list*): reads values into the argument expressions
  - **read**(*type-list*): reads values of given types, returns as tuple
  - **readln**(…) variant: same, but reads through next linefeed
- Example:

```
var first, last: string;
write("what is your name? ");
read(first);
last = read(string);
writeln("Hi ", first, " ", last);
```

```
What is your name?
Chapel User
Hi Chapel User
```

- I/O to files and strings also supported

Outline

- Introductory Notes
- Elementary Concepts
- Data Types and Control Flow
- Program Structure

Tuples

- Syntax

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

homogenous-tuple-type:
  param-int-expr * type

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

- Purpose
  - supports lightweight grouping of values
    (e.g., when passing or returning procedure arguments)

- Examples

```
var coord: (int, int, int) = (1, 2, 3);
var coordCopy: 3*int = i3;
var (i1, i2, i3) = coord;
var triple: (int, string, real) = (7, "eight", 9.0);
```

Range Values

- Syntax

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

- Semantics
  - Regular sequence of integers

    low <= high: *low*, *low*+1, *low*+2, …, *high*

    low > *high*: degenerate (an empty range)

    low or *high* unspecified: unbounded in that direction

- Examples

```
1..6            // 1, 2, 3, 4, 5, 6
6..1            // empty
3..             // 3, 4, 5, 6, 7, …
```

Range Operators

- Syntax

```
range-op-expr:
  range-expr by stride
  range-expr # count
  range-expr[range-expr]
```

- Semantics
  - **by**: strides range; negative *stride* ⇒ start from *high*
  - **#**: selects initial *count* elements of range
  - **()** or **[]**: intersects the two ranges
- Examples

```
1..6 by 2     // 1, 3, 5
1..6 by -1    // 6, 5, 4, …, 1
1..6 #4       // 1, 2, 3, 4
1..6[3..]     // 3, 4, 5, 6
```

```
1.. by 2       // 1, 3, 5, …
1.. by 2 #3    // 1, 3, 5
1.. #3 by 2    // 1, 3
0..#n          // 0, …, n-1
```

Array Types

- Syntax

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

- Semantics
  - Stores an element of *elt-type* for each index
  - May be initialized using tuple expressions
- Examples

```
var A: [1..3] int = (5, 3, 9), // 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 the data parallelism talk…

For Loops

- Syntax

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

- Semantics
  - Executes loop body serially, once per loop iteration
  - Declares new variables for identifiers in *index-expr*
    - type and const-ness determined by *iteratable-expr*
    - *iteratable-expr* could be a range, array, or iterator
- Examples

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

for i in 1..3 { write(A(i)); }          // DO RE MI
for a in A { a += "LA"; } write(A);   // DOLA RELA MILA
```

Zipper Iteration

- Syntax

```
zipper-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..3, 0..5 by 2) { … }  // (1,0), (2,2), (3,4)
```

Outline

- Introductory Notes
- Elementary Concepts
- Data Types and Control Flow
- Program Structure

Procedures, by example

- Example to compute the area of a circle

```
proc area(radius: real): real {
  return 3.14 * radius**2;
}

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

```
proc area(radius = 0.0) {
  return 3.14 * radius**2;
}
```

Argument and return types can be omitted

- Example of argument default values, naming

```
proc 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)
writeCorrd(y=2.0, 3.0);  // (3.0, 2.0)
```

Iterators

- ***Iterator:*** a procedure that generates values/variables
  - Used to drive loops or populate data structures
  - Like a procedure, but yields values back to invocation site
  - Control flow logically continues from that point
- Example

```
iter fibonacci(n) {
  var current = 0,
      next = 1;
  for 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
for f in fibonacci(7) do
  writeln(f);
```

```
0
1
1
2
3
5
8
```

Generic Procedures

- Generic procedures can be defined using type and param arguments:

```
proc foo(type t, x: t) { … }
proc bar(param bitWidth, x: int(bitWidth)) { … }
```

- Or by simply omitting an argument type (or type part):

```
proc goo(x, y) { … }
proc sort(A: []) { … }
```

- Generic procedures are instantiated for each unique argument signature:

```
foo(int, 3);         // creates foo(x:int)
foo(string, "hi");   // creates foo(x:string)
goo(4, 2.2);         // creates goo(x:int, y:real)
```

Other Base Language Features not covered today

- Records and Classes for OOP
- Modules for managing namespaces
- Argument Intents
- Enumerated types
- Type select statements, argument type queries
- Compile-time features for meta-programming
  - type/param procedures
  - folded conditionals
  - unrolled for loops
  - user-defined compile-time warnings and errors

Status: Base Language Features

- Most features are in reasonably good shape
- Performance is currently lacking in some cases
- Some semantic checks are incomplete
  - e.g., constness-checking for members, arrays
- Error messages could use improvement at times
- OOP features are limited in certain respects
  - user constructors for generic classes, subclasses
- Some memory is leaked (e.g., strings)

Future Directions

- I/O improvements
  - Binary I/O
  - Parallel I/O
  - General serialization capability
- Fixed-length strings
- Error handling/Exceptions
- Interfaces
- Improved namespace control
  - private fields/methods in classes and records
  - module symbol privacy, filtering, renaming
- Interoperability with other languages

Questions?