Tool Improvements

Chapel Team, Cray Inc.

Chapel version 1.11

April 2, 2015

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Tools Overview

- Prototype Chapel Interactive Programming Environment
- chpltags: source code navigation aid
- chpldoc: source-to-documentation tool
- Tool Priorities and Next Steps

Prototype Chapel IPE
(Interactive Programming Environment)

Executive Summary

- **Foundation for interpreter developed**
  - Taking a depth-first approach in implementing features
    - Initially support narrow feature set, sprinting toward parallelism/locality
    - Then broaden iteratively
- **Recent focus**
  - Managing scopes / environments
  - Scope/Function resolution
  - Incremental application development
- **Primary challenges**
  - Sequencing of transformations
    - Currently applied on a program-wide basis
    - IPE must operate on single statements
  - Current implementation of resolve is complex
    - Implemented in multiple passes
    - Interleaved with other passes esp. normalization

Status of Prototype

What Can It Do?

```
1> proc PowerOfTwo(n : int) : int
   {
     var i   = 0;
     var res = 1;

     while (i < n)
     {
       i   = i   + 1;
       res = res * 2;
     }

     return res;
   }

2> writeln('2**10 = ', PowerOfTwo(10));
   2**10 = 1024
```

Simple Operations

- **Arithmetic operations on int, real**
  - Unary +,-
  - Binary +, -, *, /
- **Equality comparison on int, real, bool**
  - ==, !=
- **Ordered comparison on int, real**
  - <, >, <=, >=
- **Misc**
  - abs
  - _cond_test: implicit conversion to bool for conditionals
  - primitive version of writeln
- **Expressed as module code in ChapelBase**

Simple Control Flow

- **if-then and if-then-else statements**
- **while-do statement**
  - no break or continue
- **return statement**
  - If present, must be final statement of procedure

User-defined Functions

- **Fixed number of formals**
  - Types must be specified
- **Return type must be specified**
  - Can be void
- **At most 1 return statement**
  - Must be last statement

Interactive Console

- **Enter statements interactively**
  - Variable definitions
  - Procedure definitions
  - If statements
  - While-Do statements
  - Function calls
- **Output using limited version of writeln()**

```
1> var x = 8;
2> writeln('x = ', x);
  x = 8
3> x = 12 + 4 * x;
4> writeln('x = ', x);
  x = 44
```

Development Strategy

- **Initially narrow and deep**
  - ✓ Manage modules
  - ✓ Simple expressions on default integers, reals, bools, cstring
  - ✓ Simple control flow and function calls
  - ✓ Simple interactive console
  - Tasks within a single locale e.g. implement **begin** statement
  - Multi-locale e.g. implement **on** statement
- **Then widen iteratively**
  - Remaining primitive types
  - Remaining sequential control flow
  - Generic functions
  - Iterators
  - Classes, records, enums, etc
  - Remaining parallel control flow

Compatibility with Static Chapel

- **Static Chapel programs are valid interactive programs**
  - Currently
    - Very small fraction of static Chapel supported
  - Nearer-term
    - Multi-tasking within a single locale
    - Increasing support for breadth of static Chapel
    - Investigation to support multi-locale programs
  - Longer-term
    - Full support for sequential static Chapel
    - Multi-locale programs
    - Scalable support for extern C procedures
- **Strive to make interactive programs valid static programs**
  - Some challenges exist due to nature of interactivity
    - e.g. incremental definition and redefinition of functions

Common Codebase

Mostly Positive for Compiler and Interpreter

- **Pros**
  - Compiler provides a mature infrastructure for Chapel
    - Existing parser, AST, transformations
  - Development for IPE benefits compiler
    - Improved abstraction in AST e.g. introduction of Loop statements
    - Interactive console => pure parser
  - Chapel language and implementation still maturing
    - Interpreter and compiler grow together
    - Additional changes to AST
    - Refinement of scope/function resolution
- **Cons**
  - Existing pipeline tailored to compiler
    - Limited abstraction
    - Largely fixed-order program-wide transformations
    - Refactoring reduces initial rate of progress for IPE

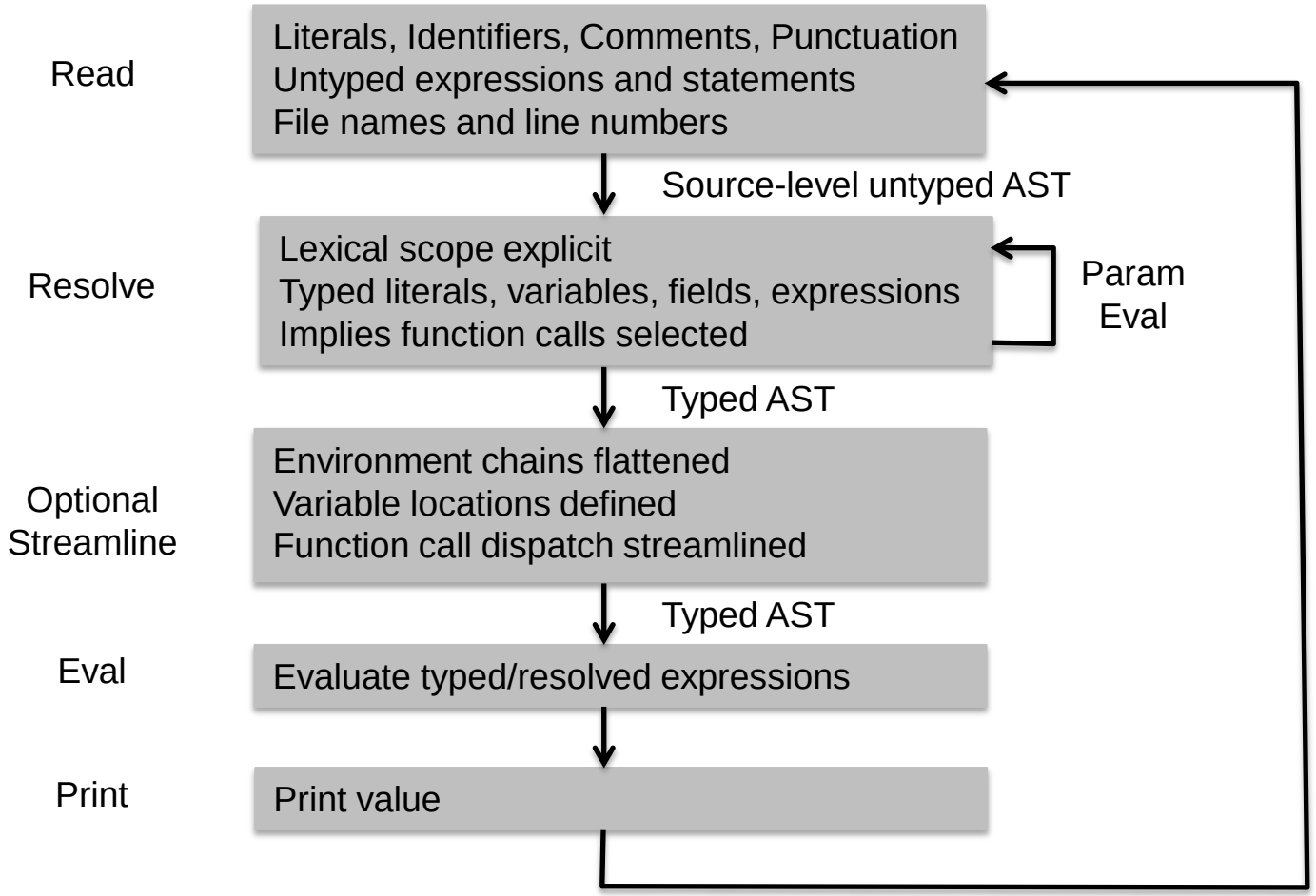
Refactoring Driven By IPE

- **Introduced new AST nodes derived from LoopStmt**
  - Conceived to simplify interpreter's use of While stmts
    - These had been expanded as nested BlockStmts during parsing
    - Minor work remains for compiler
  - Reduced reliance on BlockStmt::blockInfo for loop management
    - Reported to have benefited vectorization effort
  - LoopStmt currently derives from BlockStmt
    - Should derive from Stmt
- **Parser is reentrant (almost)**
  - Bison/Flex updated to use pure, push API
    - No changes to grammar
  - Basis to develop a Chapel-rific source include feature
    - Will need to complete the removal of global state

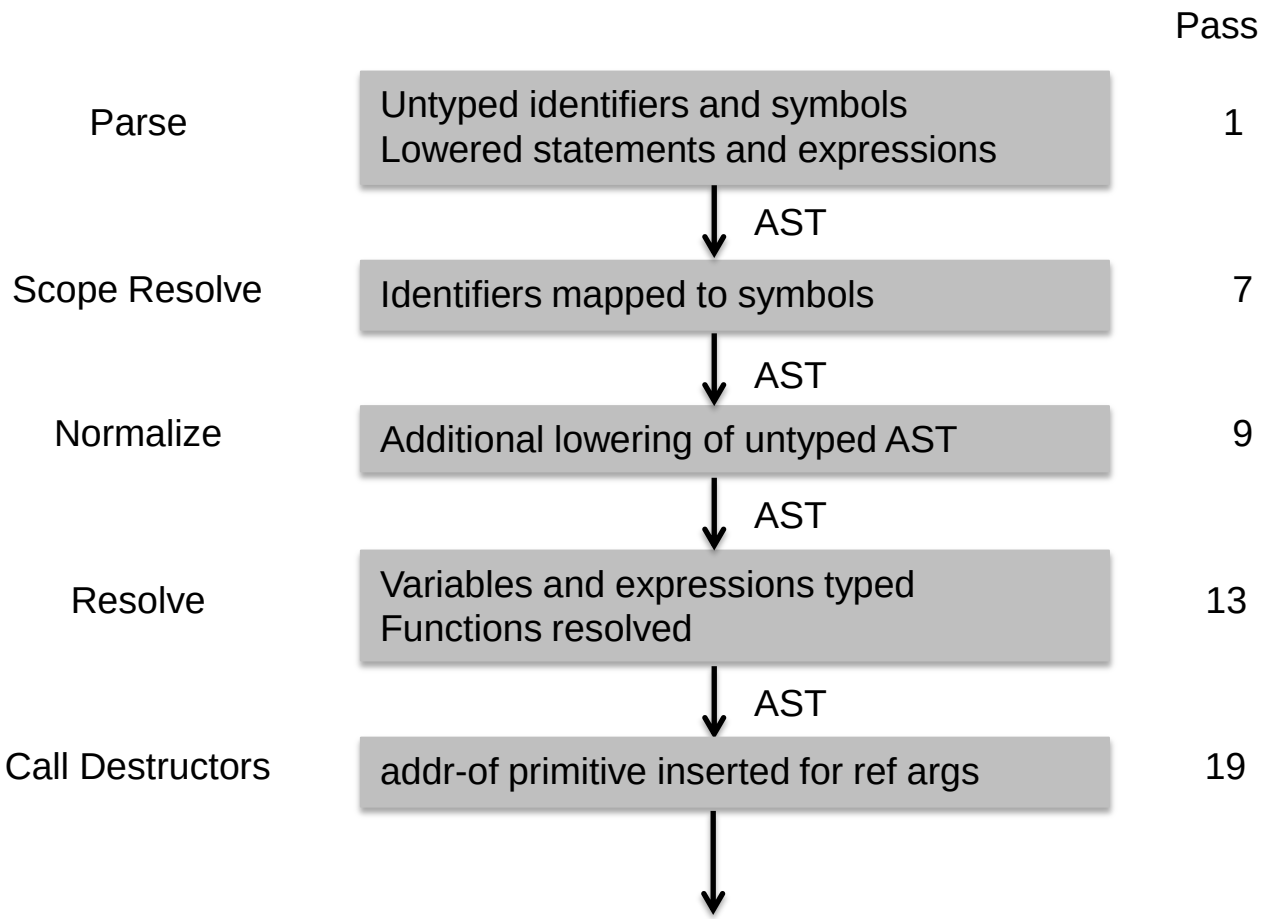
Future Contributions to Compiler Front-end

- **Interpreter should drive enhancements to AST**
  - Interpreters should provide source-level error messages and tracing
    - Challenging with current AST
    - Compiler lowers many statements during parse pass
- **Interpreter should drive improvements to resolve**
  - Interpreter requires statement-oriented version of resolve
    - Efficiency argues for performing this once per program statement
- **Interpreter and Compiler should share a front-end**
  - Parser generates source-level untyped AST
    - Literals, identifiers, source-level statements, line numbers
    - Input for chpldoc and resolution
  - Resolution generates typed AST
    - Types, variables, fields, expressions, environments
    - Can be executed with acceptable efficiency by interpreter
    - Compiler can lower/transform as appropriate

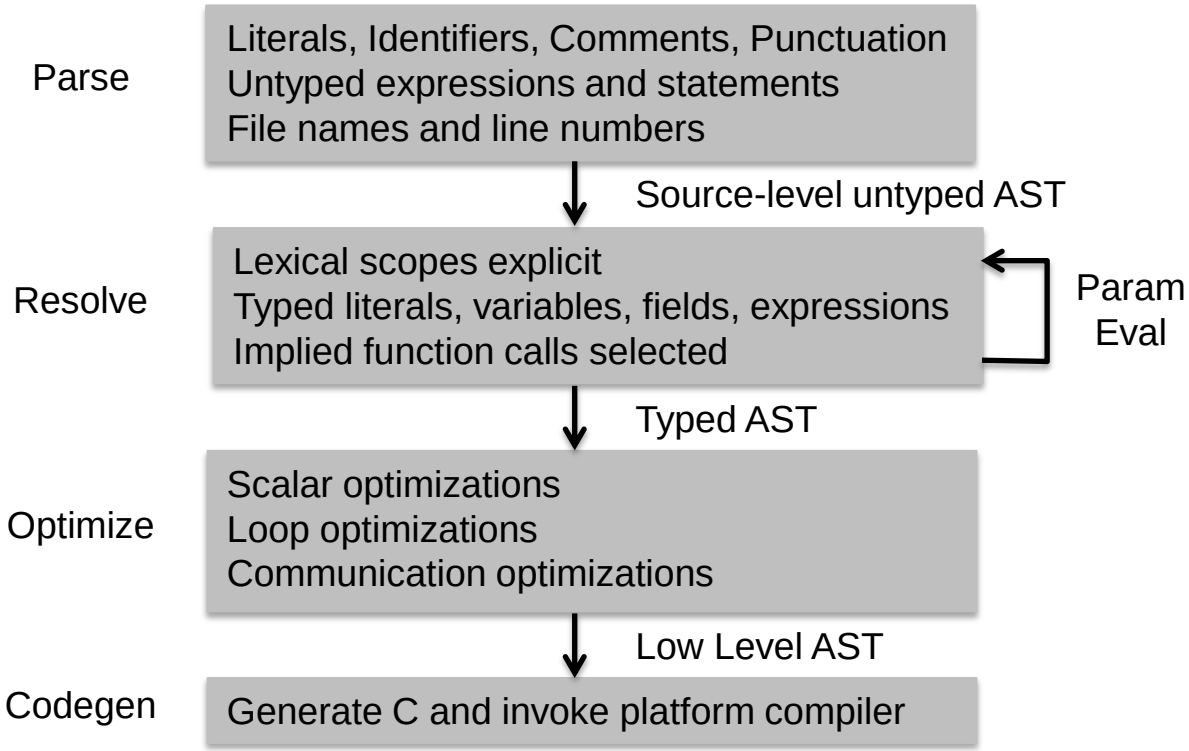
What the Interpreter Wants



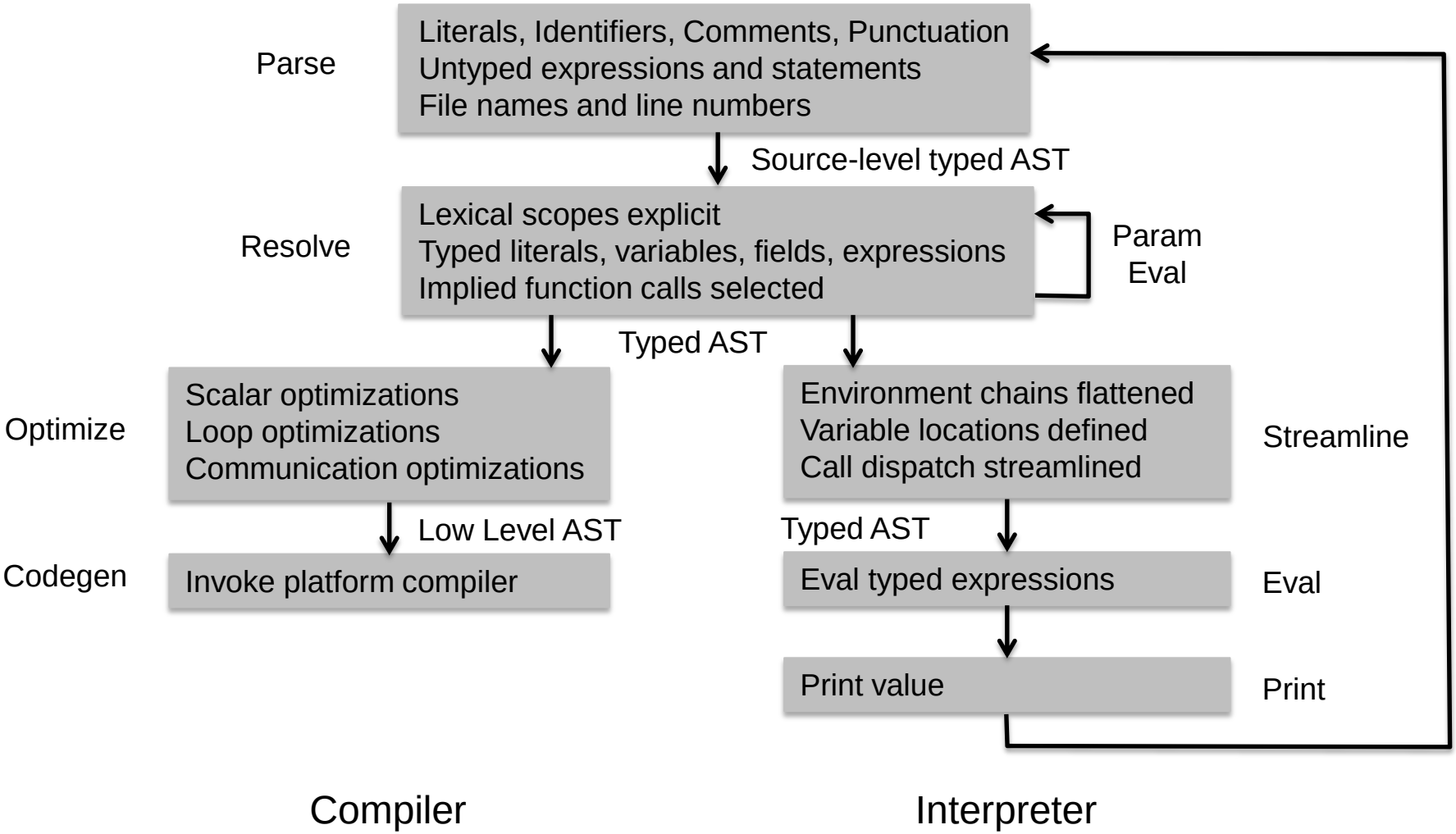
What the Interpreter Gets



What the Compiler Wants



A Shared Future



30,000’ View of Interpreter

Focus is on Expressions and Environments

- **Expressions**
  - Literals
  - Identifiers
  - Definitions
  - Control flow (if, while, for, ...)
  - Calls
- **Environments**
  - A sequence of frames
    - A frame is a table of bindings from identifiers to types and values
  - Frames are created by
    - parsing a module declaration
    - calling a function
    - entering a block statement
  - Frames related by lexical nesting and use statements

Syntactic Dispatch on Expression Type

- **Literals**
  - Self-evaluating
- **Identifiers**
  - Walk the environment chain to find the current binding
- **Definitions**
  - Extend the current environment frame
- **Control flow (if, while, for, ...)**
  - Handful of special cases to implement
- **Calls**
  - Determine the type of every actual and select required definition
  - Construct a new environment and bind formals to actuals
  - Evaluate body in new environment
  - Drive most of the effort

Efficiency

- **Operating on source level statements is inefficient**
  - walking static environment chains to find binding for an identifier
  - resolving functions
  - walking dynamic environment chains to find value for a binding
- **Opportunities to streamline execution**
  - Adopt scope resolve to map identifiers to symbols
  - Adopt function resolution to select function statically
    - Must be able to handle redefinition of functions
  - Flatten environments
    - Module level variables in common store at depth 0; respect lexical scoping
    - Flatten nested block-statement environments within a function
  - Inline calls to primitive operations
- **Be lazy e.g. do not resolve functions until/unless called**
- **“Try not to be stupid about things” cuts both ways**

Redefine a Function

```
1> proc square(x : int) : int return x + x;          // typo
2> proc sumOfSquares(x : int, y : int) : int
     return square(x) + square(y);
3> writeln('sos(3, 4) => ', sumOfSquares(3, 4));
   sos(3, 4) => 14                                   // Oh no!
4> proc square(x : int) : int return x * x;
5> writeln('sos(3, 4) => ', sumOfSquares(3, 4));
                                                     // 14 or 25?
```

The call to sumOfSquares()
  resolved the formals and body of sumOfSquares()

The execution of the body of sumOfSquares()
  resolved the formals and body of square()

Natural representation of the first call to square() in sumOfSquares() is

```
#<CallExpr #<SymExpr var: #<FnSymbol  name: "square">>
           #<SymExpr var: #<ArgSymbol name: "x">>>
```

How Do We Get 25?

- **Bookkeeping and back-patching call-sites**
  - Feels complex and error prone
  - Some code updates might result in an observable pause
- **Indirection**
  - Treat a function definition “as if” a variable definition i.e.
    - **var** square **= lambda**(x : **int**) : **int return** x ***** x;
    - Redefinition becomes a variable assignment
    - Sets stage for better support of first-class functions
  - At call-site resolve baseExpr to VarSymbol rather than FnSymbol
    - Calls fetch value of function variable and apply the value to the actuals
    - Indirection could be optimized away for *sealed* modules

Add an Override

```
1> proc square(x : real) : real return x * x;
2> proc sumOfSquares(x : int, y : int) : real
     return square(x) + square(y);
3> writeln('sos(3, 4) => ', sumOfSquares(3, 4));
   sos(3, 4) => 25.0
4> proc square(x : int) : int return x + x;        // Deliberate
5> writeln('sos(3, 4) => ', sumOfSquares(3, 4));
```

How do we get 14.0?

- Treat functions as collections of typed methods
  - Inspired by CLOS's view of generic functions
- Functions are versioned
  - Version id updated if set of methods is altered
- Call sites encode the version id and method offset
  - Version id inspected for every call
  - Call-site updated if version-id obsolete

IPE Next Steps

- **Expand feature set:**
  - Support for single-locale parallelism (e.g., `begin`)
  - Support for multi-locale execution (e.g., locales and `on`-clauses)
- **Improve compiler/interpreter architecture:**
  - Additional AST refactorings
  - Investigate shared, revamped, early resolution pass---also desired for:
    - concepts/constrained generics/interfaces work
    - compiler optimization efforts
    - reducing generated code size / compile time
    - richer support for param computations
    - developer community

chpltags: source code navigation aid

chpltags

Background: Navigating the Chapel modules is challenging

- ~47,000 lines of Chapel source as of 1.11 release

This Effort: Create a new utility – `chpltags`

- Generates 'tags' files for Chapel source code
- Passes regular expressions off to `ctags` or `etags`

Impact: We now build 'tags' for all module code

- Try out the tag support in your favorite editor!

Next Steps: Write a parser-based `chpltags`

- Regular expressions alone cannot parse all valid Chapel source:

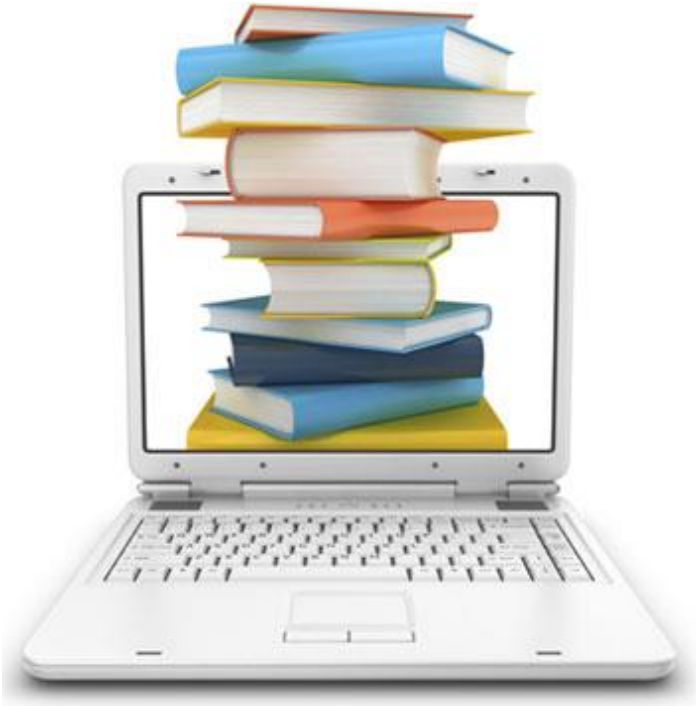
```
var x, y, z = 5.0; // Only the definition of x will be found
```

- Not a priority – the current state gives us most of what we need

chpldoc: source-to-documentation tool

chpldoc: Background

- Modern users expect online library docs
- Ideally, generate docs from source code and comments
- chpldoc prototype present since Chapel 1.6



chpldoc: chpldoc prototype

chpldoc prototype…

…Parsed code and comments

…Generated text or HTML

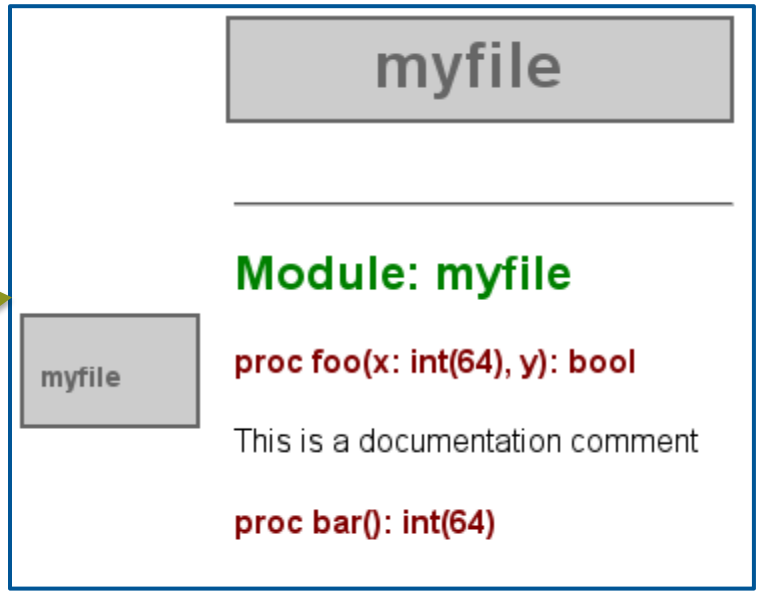
myfile.chpl:

```
/* This is a documentation
   comment */
proc foo (x: int, y): bool {
   …
}

// This is not documentation
proc bar (): int {
   …
}
```

myfile.txt:

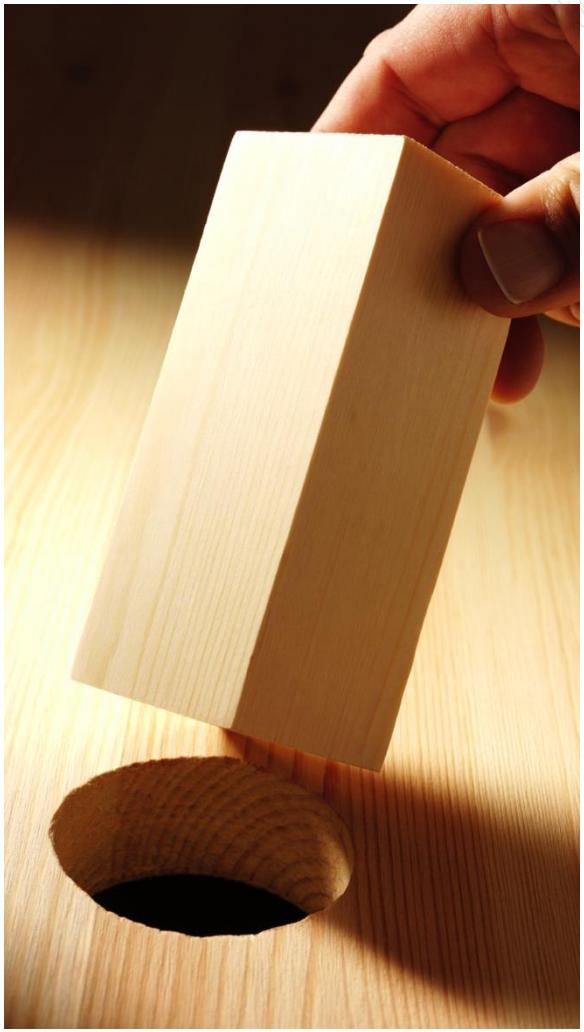
```
Module: myfile
   proc foo(x: int(64), y): bool
       This is a documentation
   comment

   proc bar(): int(64)
```

myfile.html:



chpldoc: chpldoc prototype (cont)

- **HTML output had minimal features**
  - Minimal UI
  - Missing search and cross references (links)
- **Limited language support**
  - Missing some language features
  - Bugs in output
- **Tightly coupled to compiler**
  - chpldoc and chpl shared same flags
  - chpldoc info hard to find in man page
  - Extra build step to get html



chpldoc: This Effort

- separate chpldoc from chpl
- Rich HTML output
- Support standard library
- New features
- Fix bugs in existing chpldoc

chpldoc: separate chpldoc from chpl

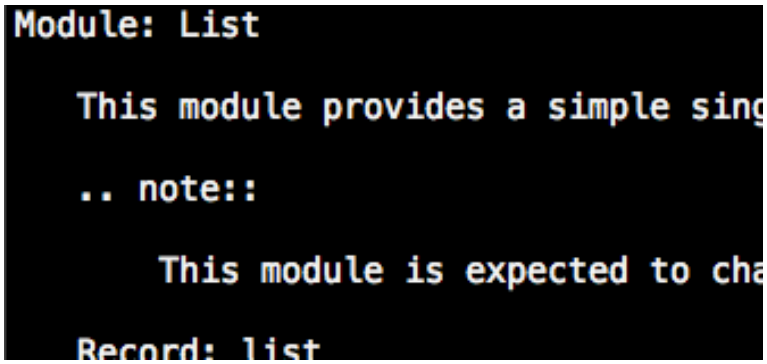
- **Built separately: make chpldoc**
- **Separate flags**
  - No longer need --doc prefixes
- **Separate man pages**

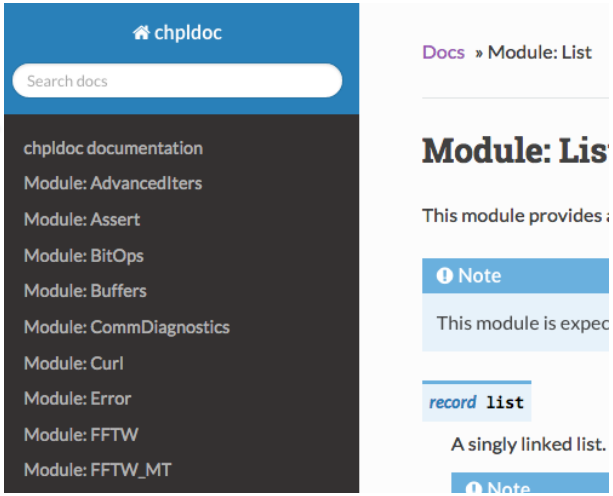
```
$ chpldoc --help
Usage: chpldoc [flags] [source files]

Documentation Options:
  -o, --output-dir <dirname>          Sets the
                                      <dirname>
      --save-sphinx <directory>       Save gene
                                      directory
```

```
$ man chpldoc
chpldoc(1)



NAME
       chpldoc - the Chap

SYNOPSIS
       chpldoc [-o direct
               [--save-sp
               [options]
```

chpldoc: Rich HTML output

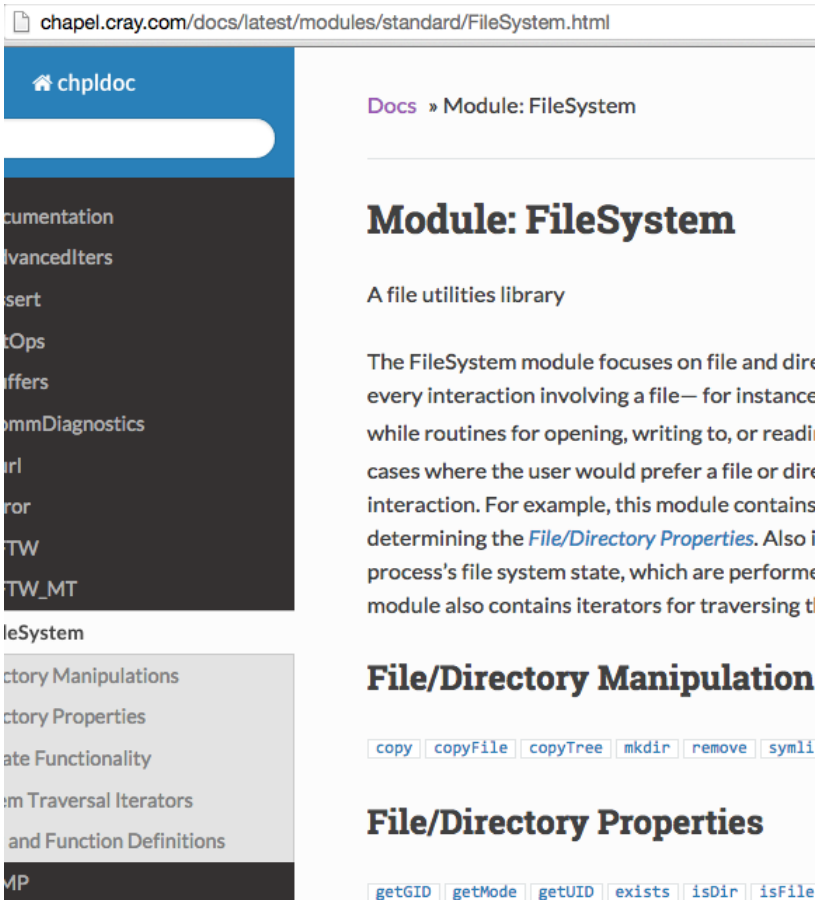
- **Comments use reStructuredText formatting**
- **Sphinx used under the covers**
  - reStructuredText parsing and formatting
  - Rich HTML UI
  - Provides cross references (links), search, etc.
  - Uses new Chapel domain for Sphinx





chpldoc: Support standard library

- Module docs online:
  - http://chapel.cray.com/docs/latest/
- See “Chapel Documentation” slides for more detail



chpldoc: New features

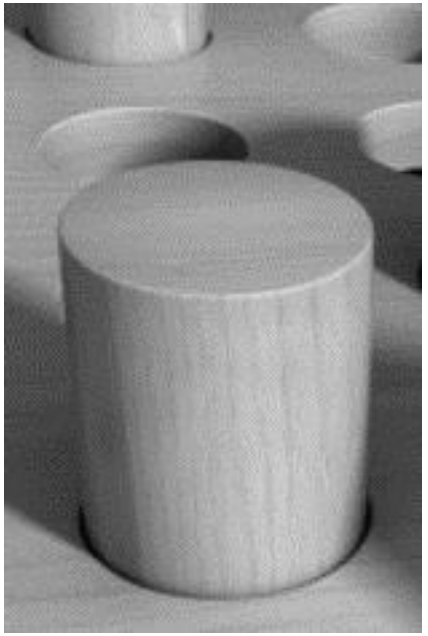
- Support for enums and globals
- pragma “no doc” to stifle symbols
- Source-based output ordering
- Hide inline/extern for procs
- “Module Index” listing all modules
- …and many incremental features!

chpldoc: Fixed bugs

- **Addressed several output bugs with:**
  - Function argument and return types
  - Class and record members
  - Missing default values in certain places
- **No-Paren procs output with parens**
- **type symbols output as var**
- **Secondary methods output twice**
  - E.g. proc MyClass.myProc() {}
- **…and many more!**

chpldoc: Impact

- Hardened standalone chpldoc program
- Used chpldoc to create online module documentation
- Added many new features
- And fixed existing bugs



chpldoc: Status and Next Steps

Status:

- chpldoc is first class citizen in project
- Usable by 3rd party library developers

Next Steps:

- Continue fixing bugs
- Support inline tests in docs
  - Similar to python doctests

chpldoc: Next steps (cont)

- **Support documentation-only files, e.g. READMEs**
  - And interaction with module docs
- **Add class and record index**
- **Improve error handling for reStructuredText formatting**
- **Support class inheritance**
- **Additional output orderings**
  - Like alphabetical and logical groupings
- **Link to source code from docs**

Tool Priorities and Next Steps

Tool Priorities and Next Steps

- **IPE:**
  - support for dynamic execution
  - support for multi-locale execution
  - architect and implement shared resolution pass with compiler

- **chpldoc:**
  - prioritize additional features based on developer/user needs

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA, and YARCDATA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

CRAY
THE SUPERCOMPUTER COMPANY

http://chapel.cray.com chapel_info@cray.com http://sourceforge.net/projects/chapel/