Hewlett Packard Enterprise

PLANNED IMPROVEMENTS TO THE CHAPEL COMPILER

Michael Ferguson, HPE

June 4, 2021

COMPILER IMPROVEMENTS: OUTLINE

- Where We Are
- Missing Features
- Proposed Direction
- Progress Report


WHERE WE ARE

PROBLEMS WITH THE CURRENT COMPILER

- **Speed**
  - The current compiler is generally slow, and extremely so for large programs (~7s to 15 minutes)

- **Structure and Program Representation**
  - The compiler is structured only for whole-program analysis, preventing separate/incremental compilation
  - Unclear how to integrate an interpreter, provide IDE support, or ‘eval’ Chapel snippets

- **Development**
  - The modularity of the compiler implementation needs improvement
  - There is a steep learning curve to become familiar with the compiler implementation

WHAT IS WORKING WELL

- The current compiler code base has enabled the design and evolution of Chapel to date
- The compiler includes key optimizations that support program performance
- The compiler is relatively fast to build and very portable
- Chapel's internal and library modules are extensive and largely independent of the compiler
- The runtime libraries are well-architected and not in need of major changes

SUMMARY OF CURRENT COMPILER PASSES

pass	time for Hello World	time for Arkouda	approx lines of .cpp
parse+	0.5s	0.8s	10,000 lines
scopeResolve	0.4s	0.7s	4,500 lines
normalize+	0.9s	2.2s	9,000 lines
resolve	2.0s	165s	35,000 lines
post-resolve	0.3s	16s	16,000 lines
lowerIterators	0.1s	7.1s	6,000 lines
parallel	0.1s	17s	2,000 lines
optimization	0.5s	46s	7,000 lines
insertWideRefs+	0.1s	15s	6,000 lines
codegen	0.5s	48s	20,000 lines
makeBinary	1.1s	422s	-
TOTAL OF ABOVE	6.0s	724s	114,000 lines
TOTAL	6.4s	743s	170,000 lines in ‘compiler/*’


MISSING FEATURES

MISSING FEATURES

- Some commonly requested features:
  - IDE support (beyond syntax highlighting)
  - Separate and/or incremental compilation

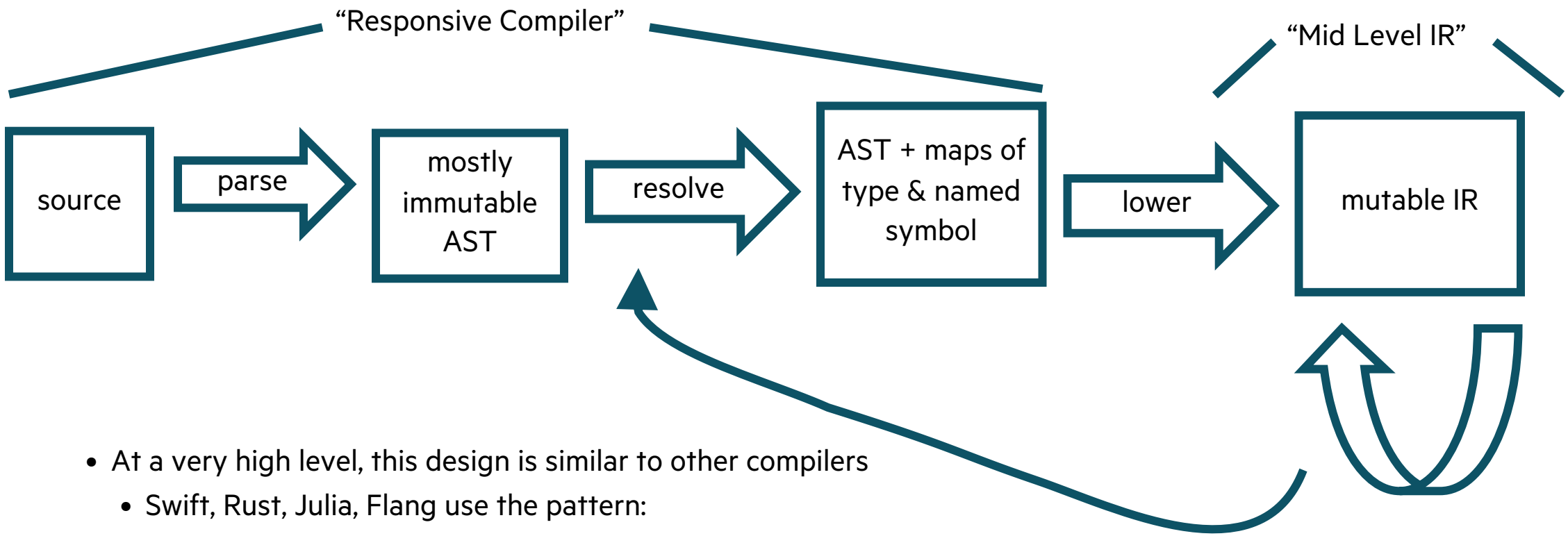
- Good IDE integration requires the compiler to
  - behave more like a server
  - respond quickly to limited queries (e.g. code completion or mouse-over)

- Incremental compilation and separate compilation require
  - a more flexible compiler that can instantiate some generics but re-use other instantiations

- Compiler architecture improvements can make these problems easier to solve


PROPOSED DIRECTION

IMPROVED COMPILER ARCHITECTURE



- At a very high level, this design is similar to other compilers
  - Swift, Rust, Julia, Flang use the pattern:
    - parse ⟶ AST ⟶ mid-level IR (and progressively lower)
- Earliest part of compilation is most relevant for IDE integration
  - so is designed to be very incremental

CHANGES TO GET TO IMPROVED ARCHITECTURE

- Create a new AST more faithful to source code for early passes
- Develop a new pass architecture with less rigid ordering
  - make passes typically run per-function rather than whole-program and otherwise be idempotent
- Convert the new AST into the old AST to enable incremental development
- Gradually port later passes over to a new IR more suited for optimization

RESPONSIVE COMPILERS

- Matsakis' talk, *Responsive Compilers*[1], presents a vision for good IDE support:
  - highly incremental and demand-driven—just process enough to answer a query
    - e.g., how to complete newBlock<tab>
    - fast response times are key for a satisfying experience

- The strategy relies on:
  - structuring compilation in terms of many fine-grained queries
    - e.g., what is the type of this variable?
  - framework uses these queries to manage dependencies among results
  - each query saves its result and is re-run when necessary
  - query results are represented separately from the input—which tends to mean a lot of maps
  - AST elements are given IDs to support these maps

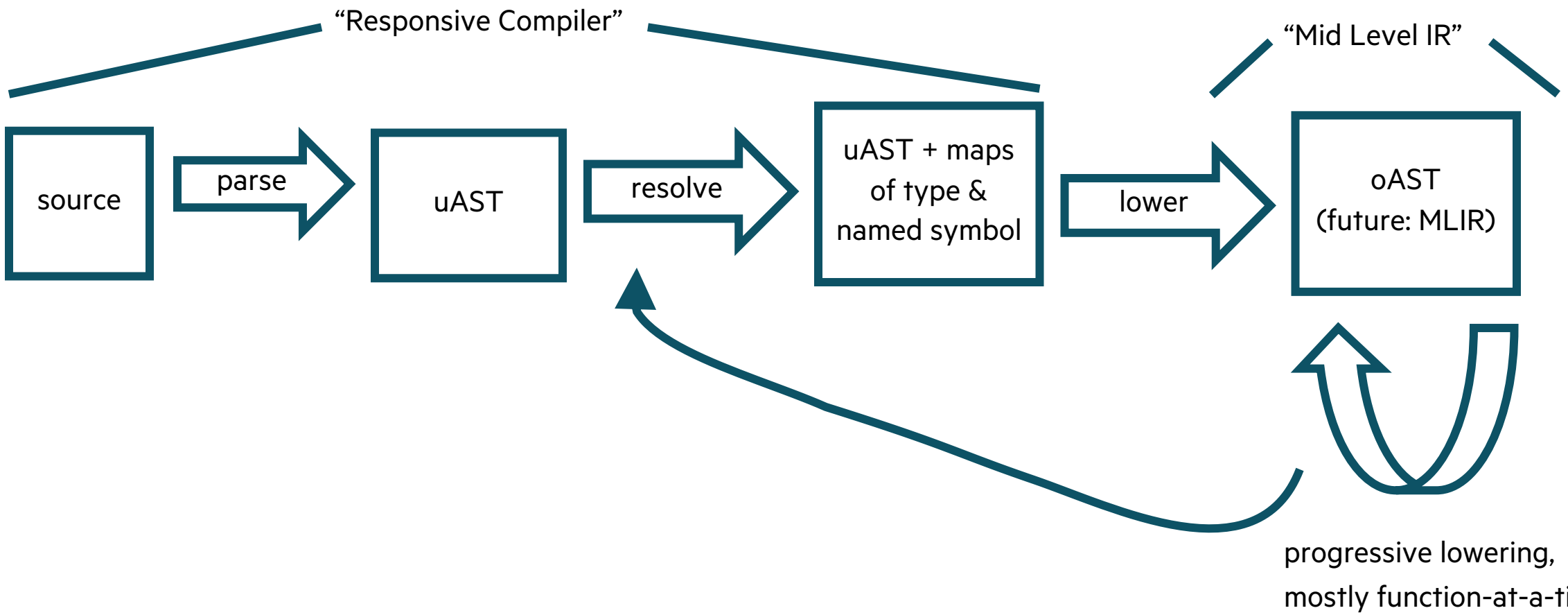
1 – *Responsive Compilers*, Nicholas Matsakis, PLISS 2019

RESPONSIVE COMPILER QUERIES

- parse(filePath) ⟶ AST for file (which also establishes IDs)
- locate(AST) ⟶ (line number, column number)

- getDefinedIn(Expr, name) ⟶ Symbols defined in ‘Expr’ named ‘name’
- getVisible(Expr, name) ⟶ Symbols visible from ‘Expr’ named ‘name’
- types(Expr) ⟶ map from Symbols to Types for Symbols defined in ‘Expr’
- resolve(Expr) ⟶ map from Identifiers to Symbols they refer to

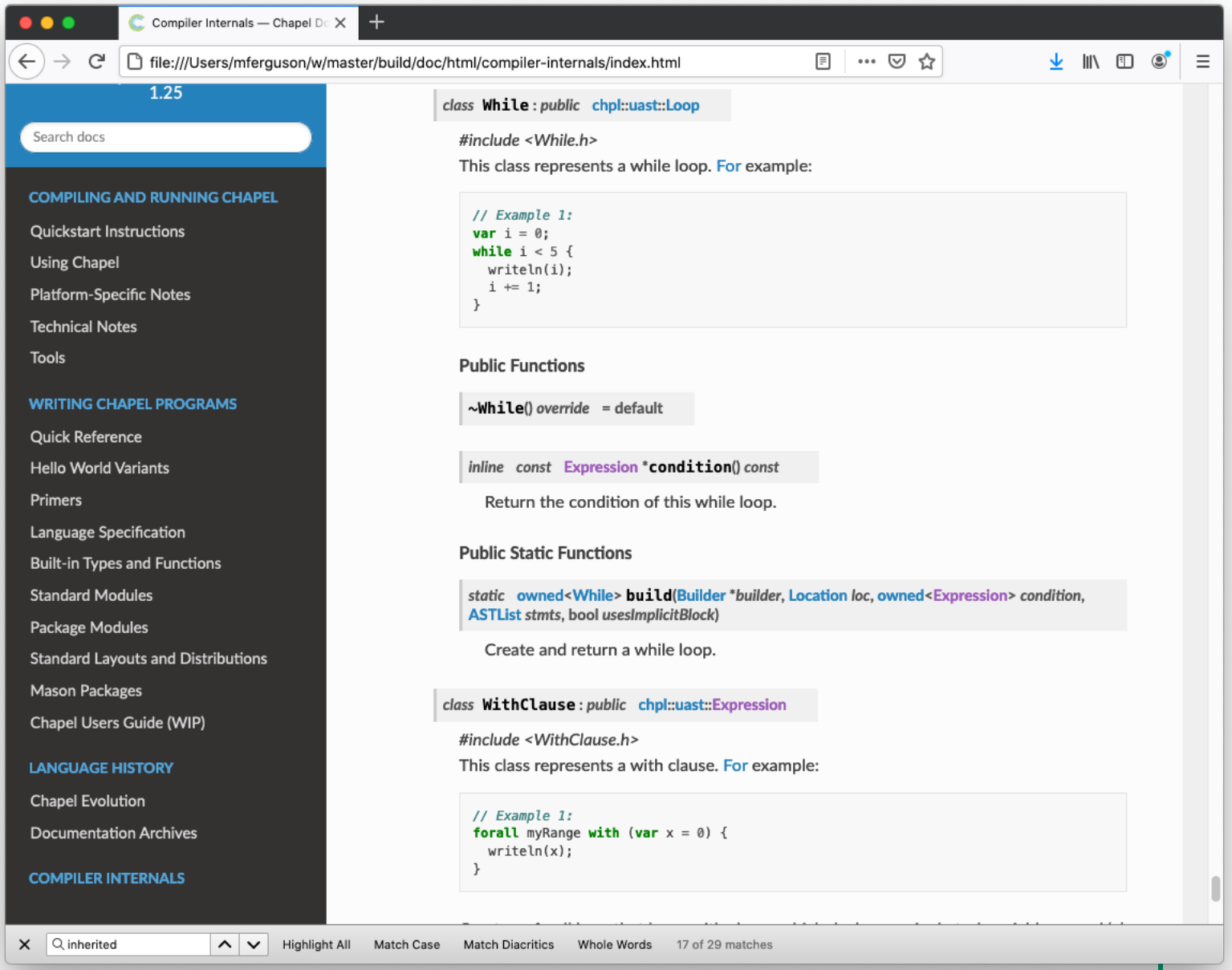
FUTURE COMPILER ARCHITECTURE




PROGRESS REPORT

COMPILER LIBRARY

- Developing new code as a library
- New uAST nodes have documentation!
- These features will enable community members to contribute Chapel tools
- Library Use Cases:
  - Linter
  - Documentation tools
  - IDE integrations



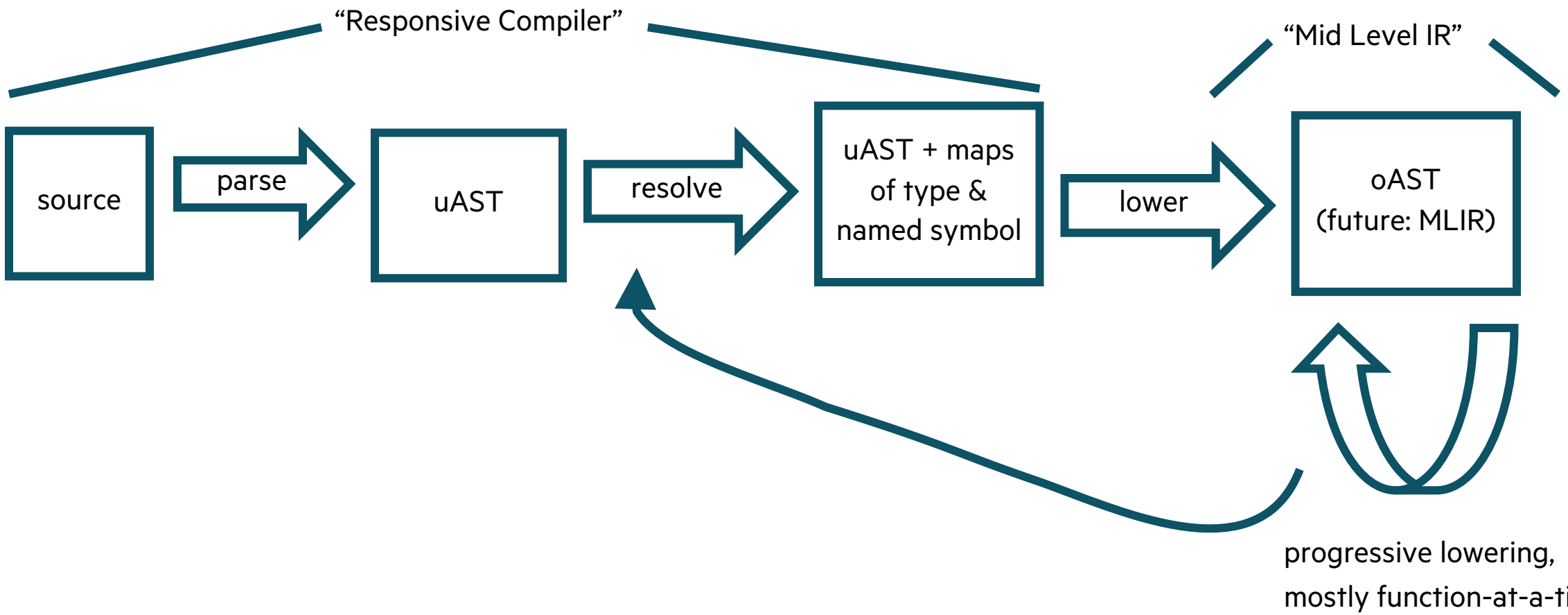
IMPLEMENTATION PROGRESS

- Part-way through implementing the new uAST, parsing it, and translating it into the old AST
- Have demonstrated incremental re-compilation with simple examples

```
// mymodule.chpl

module M {
  proc f() {
    writeln();
  }
}
```

```
prompt % testInteractive mymodule.chpl

mymodule.chpl:3: error: 'writeln' undeclared (first use t
Module M:
Module  0x7fc962406250 M
  Function M 0x7fc962406140 f
    FnCall M.f@1 0x7fc9624060a0
      Identifier M.f@0 0x7fc962406020 writeln


Would you like to incrementally parse again? [Y]:
```

QUESTIONS?


“Responsive Compiler”
“Mid Level IR”
source
parse
uAST
resolve
uAST + maps of type & named symbol
lower
oAST (future: MLIR)
progressive lowering, mostly function-at-a-time


THANK YOU
https://chapel-lang.org
@ChapelLanguage