Compilers

The graduate version - Spring 2020

Goals

- To become knowledgeable of the foundational concepts underlying modern compiler optimization
- To explore and understand the tradeoffs required when implementing scalable program analyses
- To become familiar with a production-quality compiler system that you can use in your own research

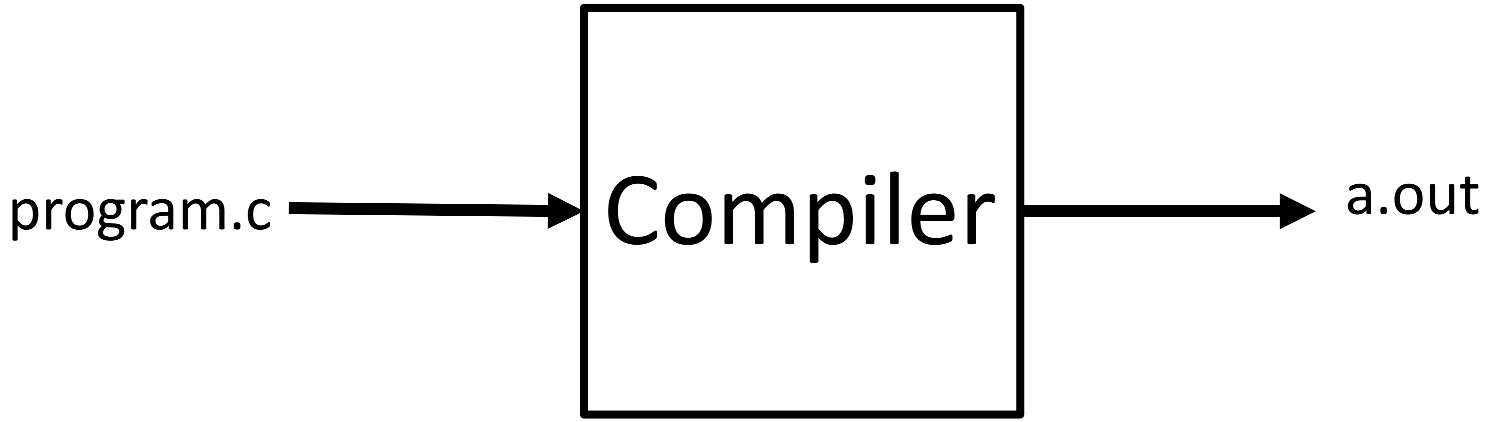
A bit about me …

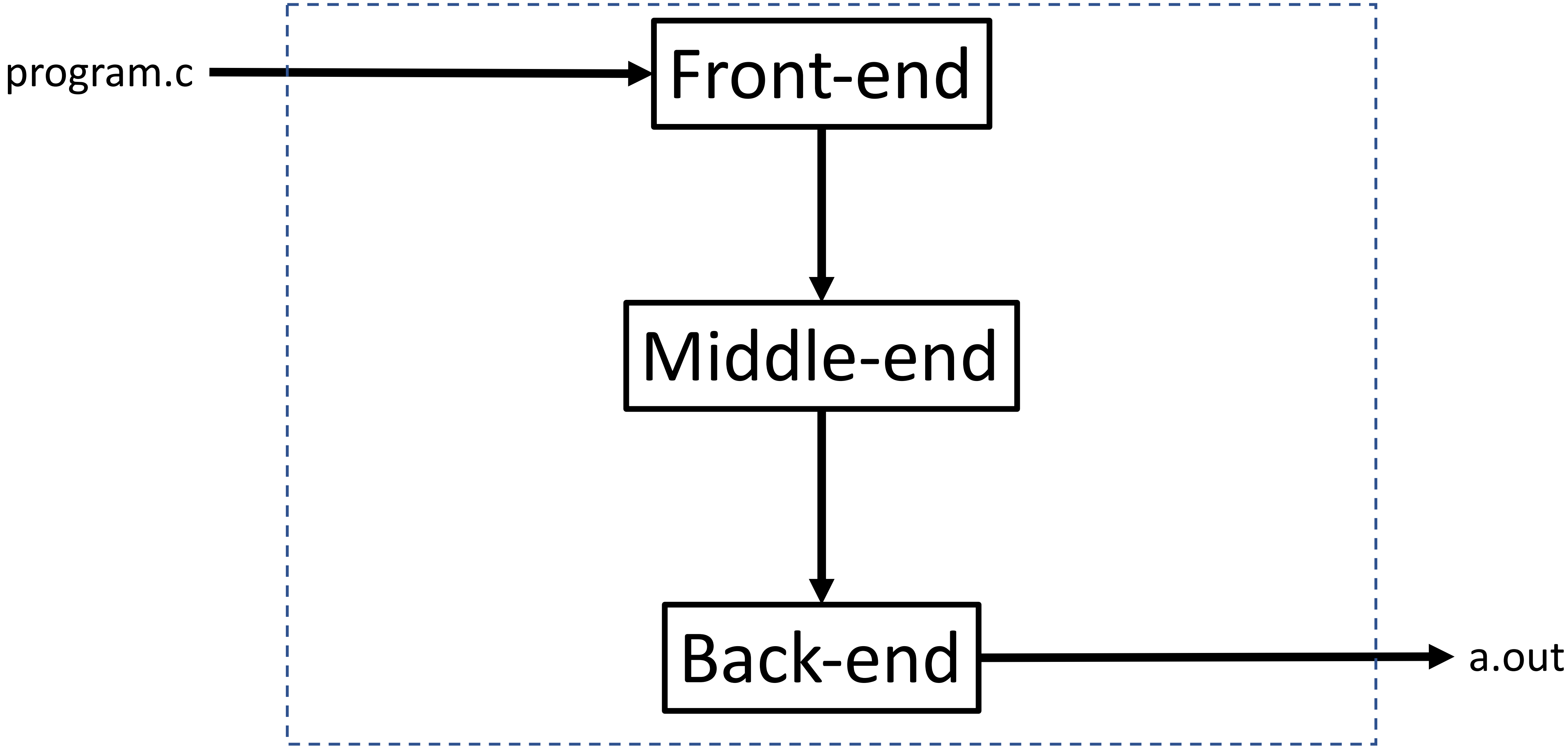
- Worked as a compiler developer in industry from 1986-1990
- Doctoral work on data flow analysis
- Have taken three courses in compilers (all grad courses)
- Have taught undergrad and graduate compilers 20 times
  - 5 different instantiations of the course
- Have implemented significant parts of 7 compilers
  - Most recently this summer (as you will see)
- Lead research on topics that are closely related to compilation

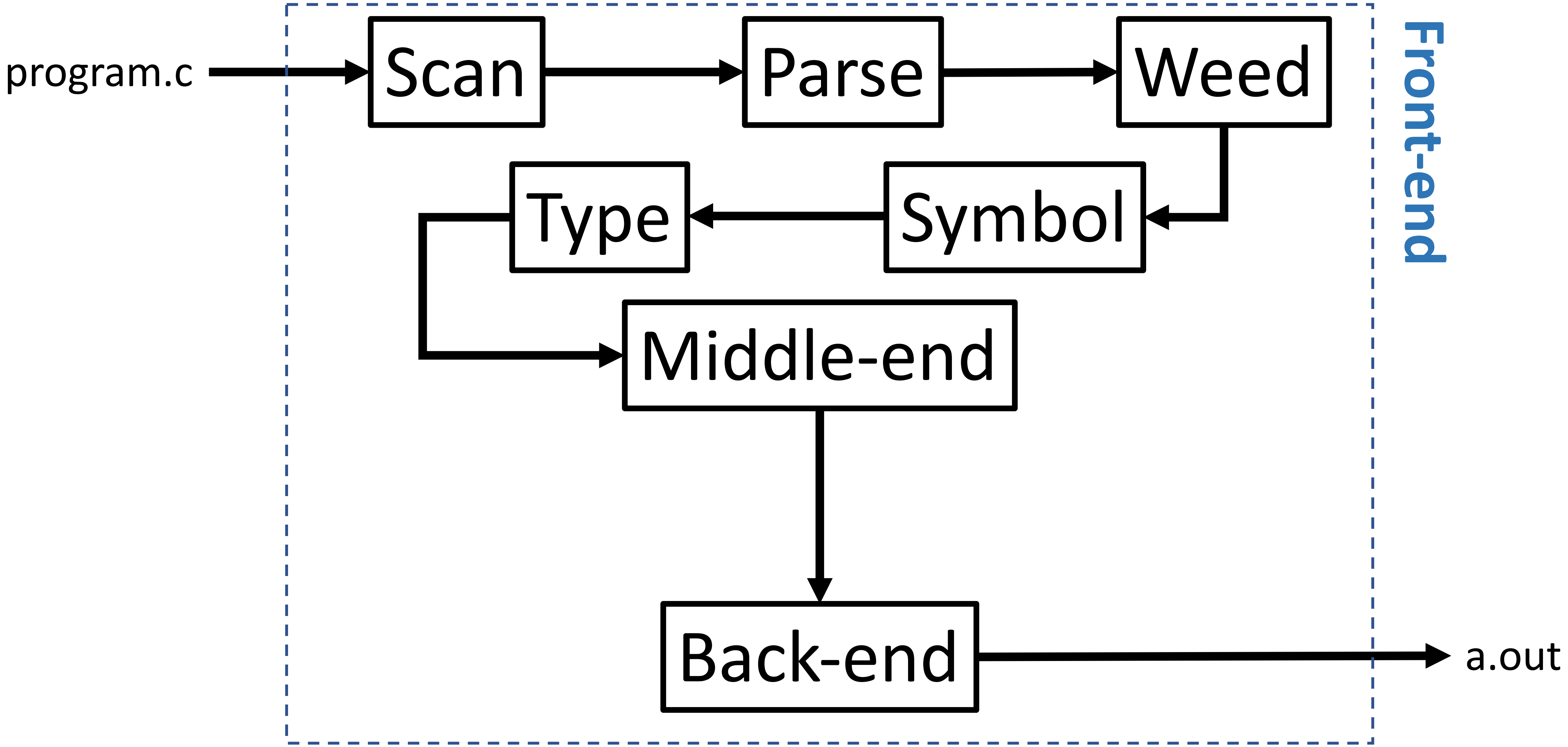
From theory to normal engineering

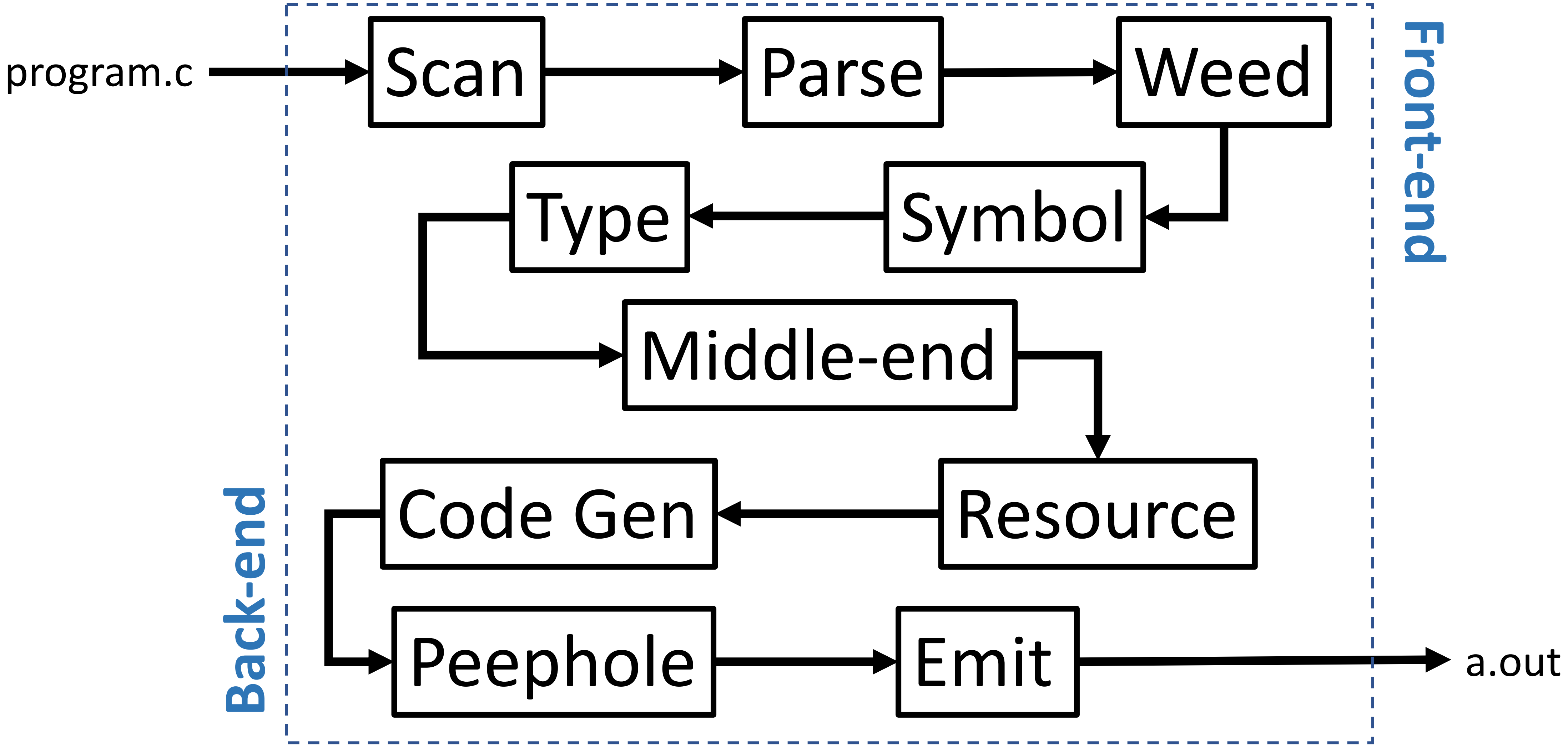
- in the 1960s compilation was *art*
- in the 1970s compilation was *theory*, i.e., studied by theoreticians
- in the 1980s and 90s compilation was *engineering*, i.e., studied as a software product line, supported by reusable programming frameworks and DSLs
- in the 2000s those frameworks became more powerful
- in the 2010s we finally figured out how to test them
- it is one of the most mature software domains you will ever encounter

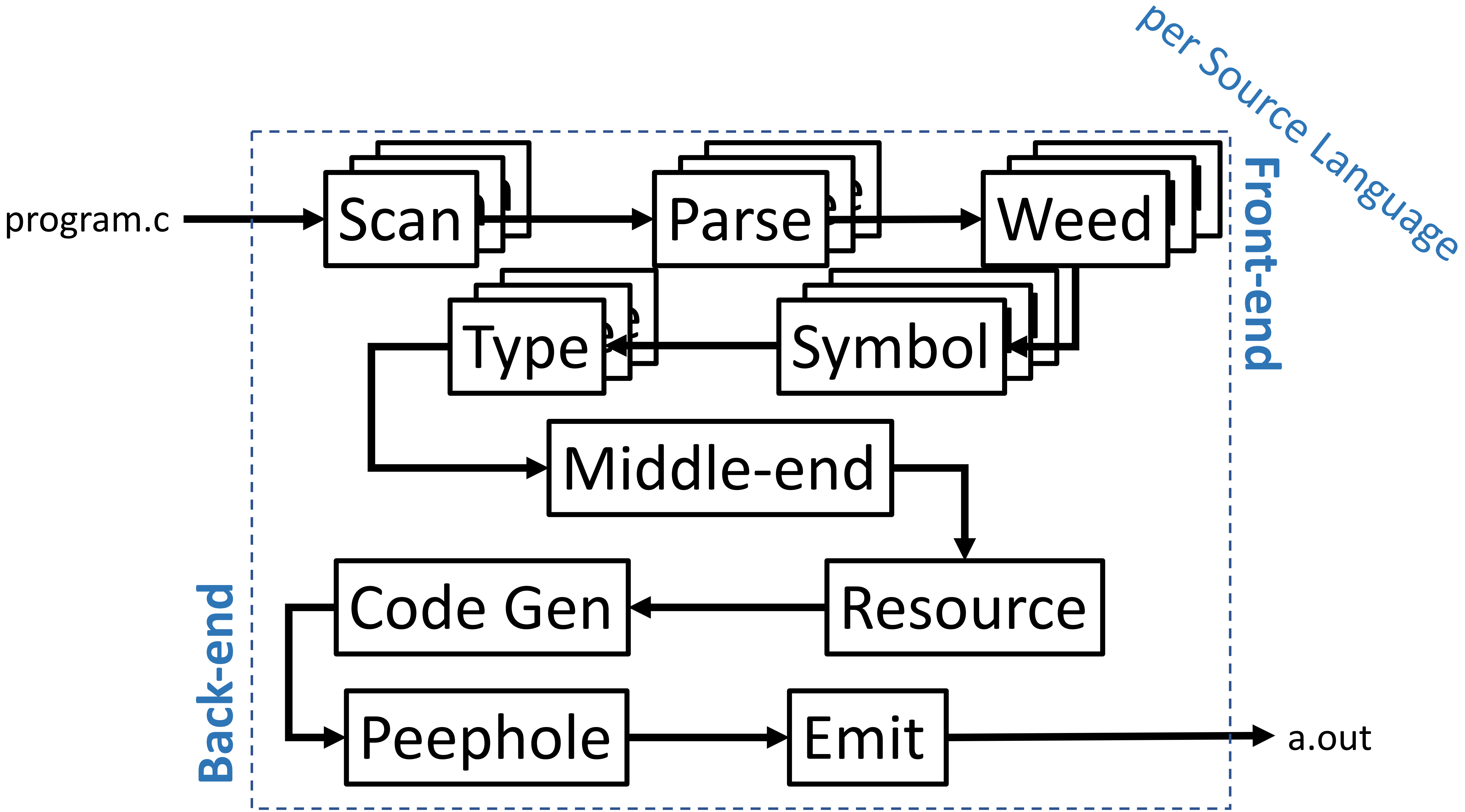
What is a compiler?

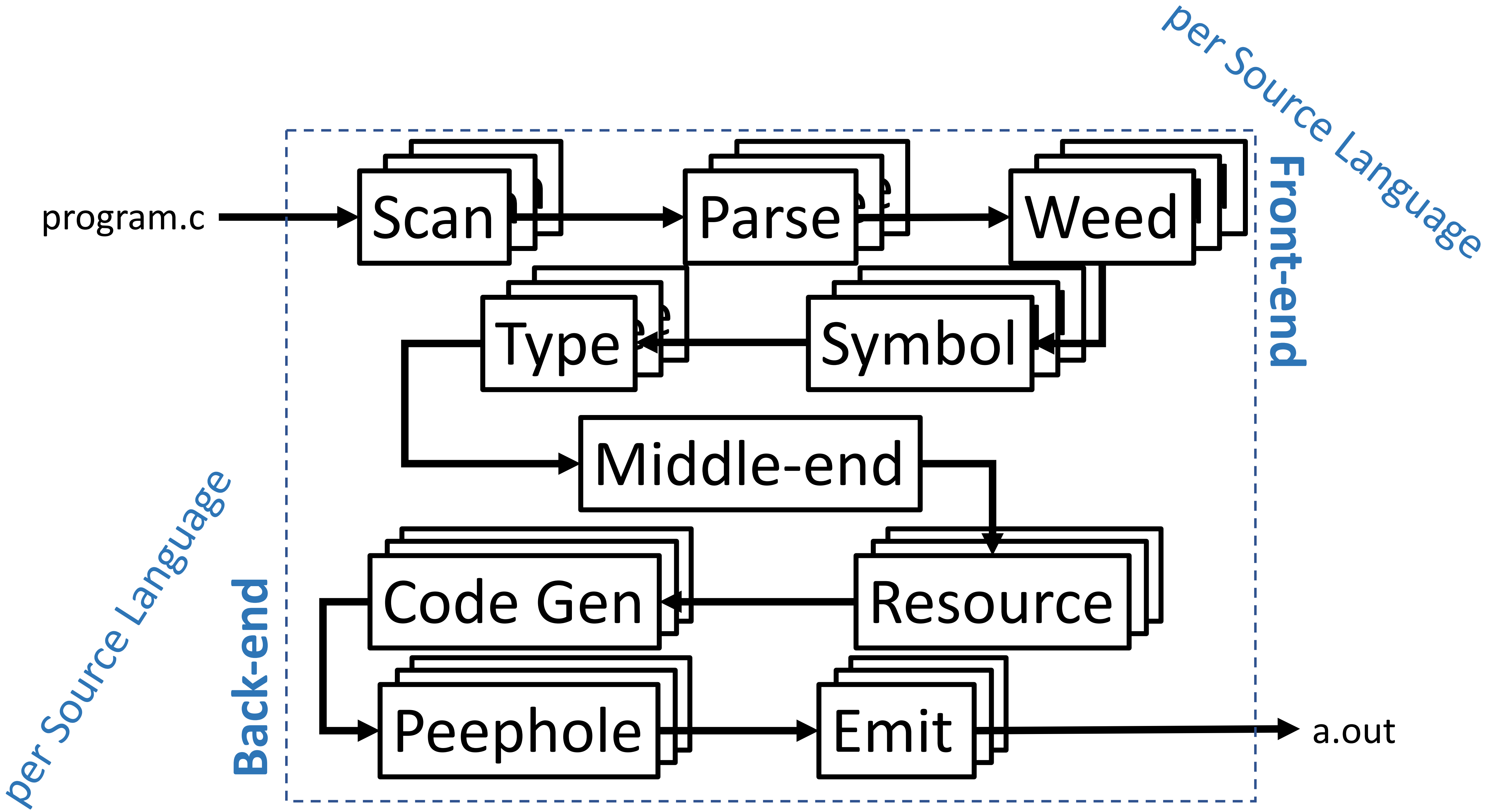



program.c
Front-end
Middle-end
Back-end
a.out


program.c
Scan
Parse
Weed
Type
Symbol
Middle-end
Back-end
a.out
Front-end


program.c
Scan
Parse
Weed
Symbol
Type
Middle-end
Resource
Code Gen
Peephole
Emit
a.out
Front-end
Back-end

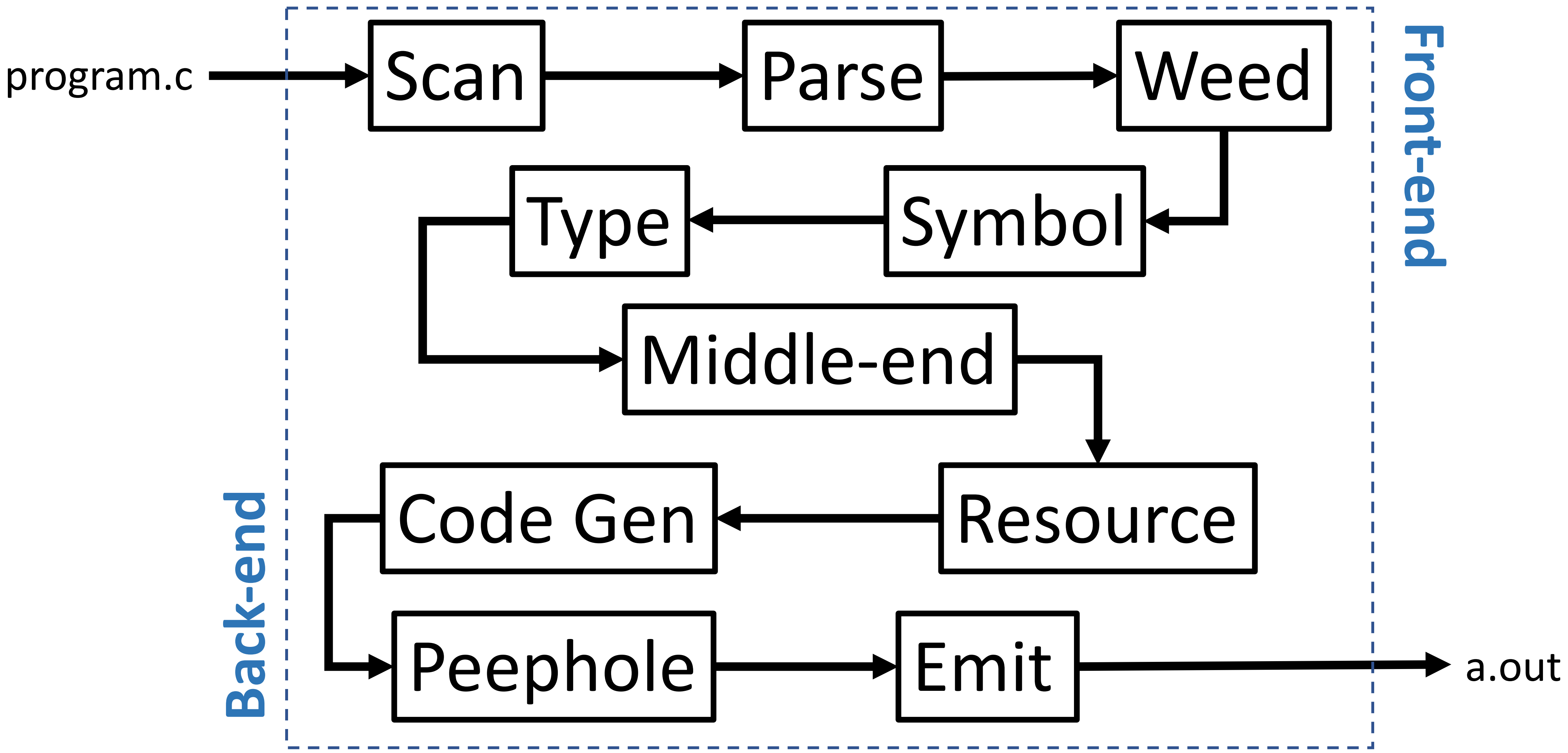

per Source Language
Front-end
Back-end
program.c
Scan
Parse
Weed
Type
Symbol
Middle-end
Code Gen
Resource
Peephole
Emit
a.out



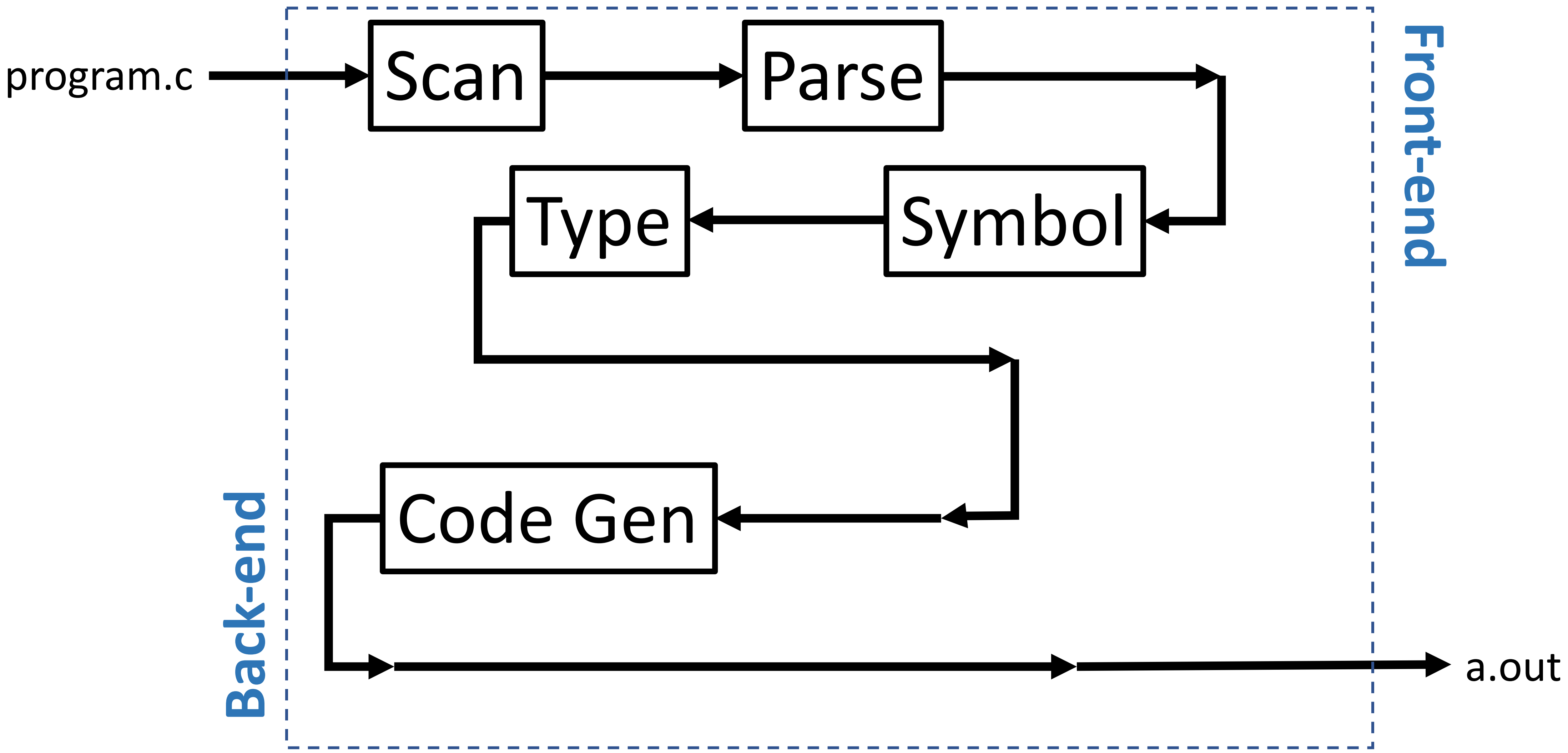
program.c
Scan
Parse
Weed
Type
Symbol
Middle-end
Code Gen
Resource
Peephole
Emit
a.out
Front-end
Back-end
per Source Language
per Source Language

Compilers are …

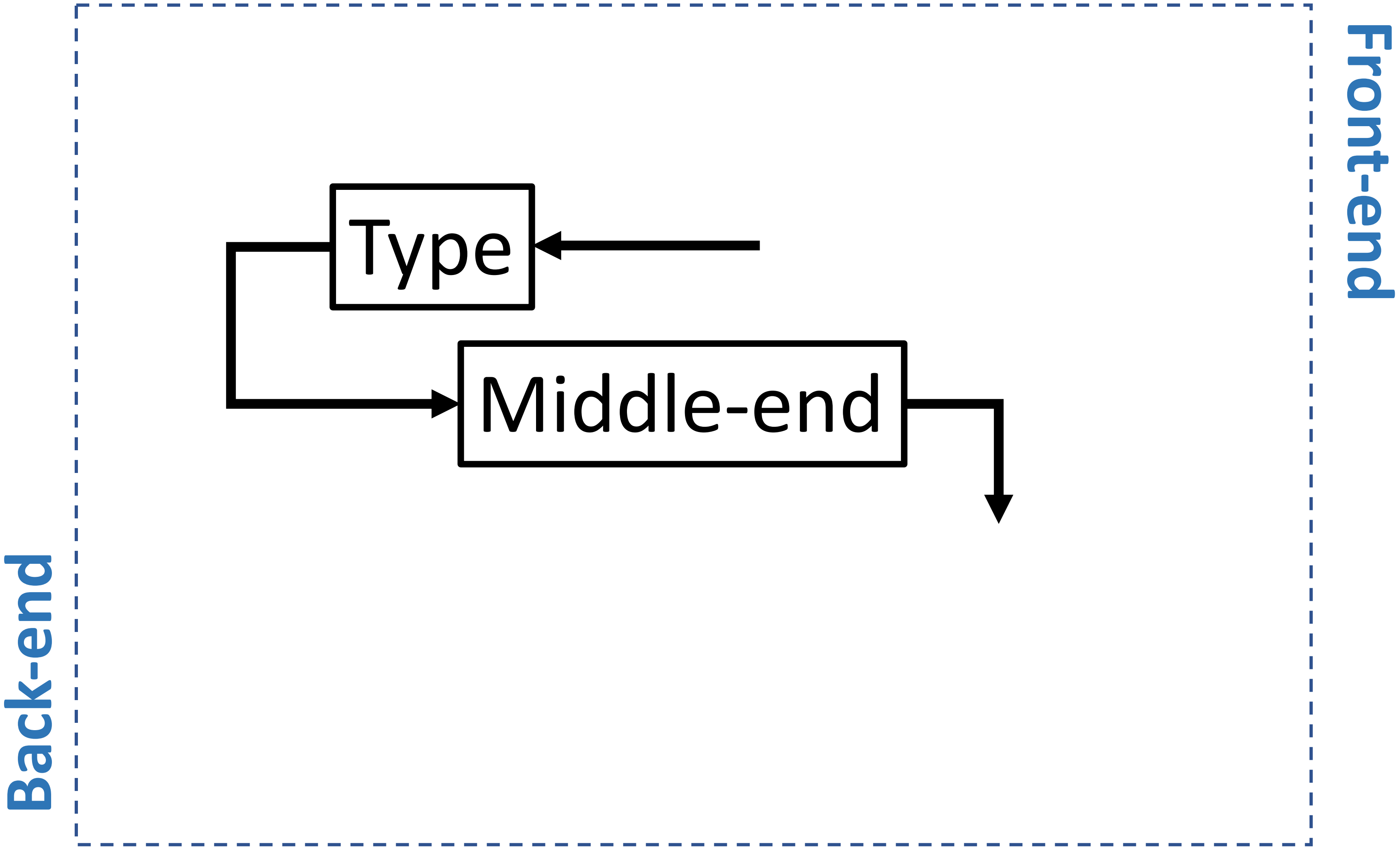
- Large complex software systems
  - GCC >7MSLOC
  - CLANG+LLVM >4MSLOC
- Highly-structured software architectures
  - Well-defined interfaces
  - Components modularized and plug compatible
- Focused on the input and output languages, e.g., for GCC
  - C, C++, Objective C, Ada, Fortran, Go, D, Cobol, Modula-2/3, …
  - arm, alpha, i386, mips, rs6000, sparc, … (51 currently)
- **We are going to side-step a lot of that complexity**


program.c
Scan
Parse
Weed
Type
Symbol
Middle-end
Resource
Code Gen
Peephole
Emit
a.out
Front-end
Back-end

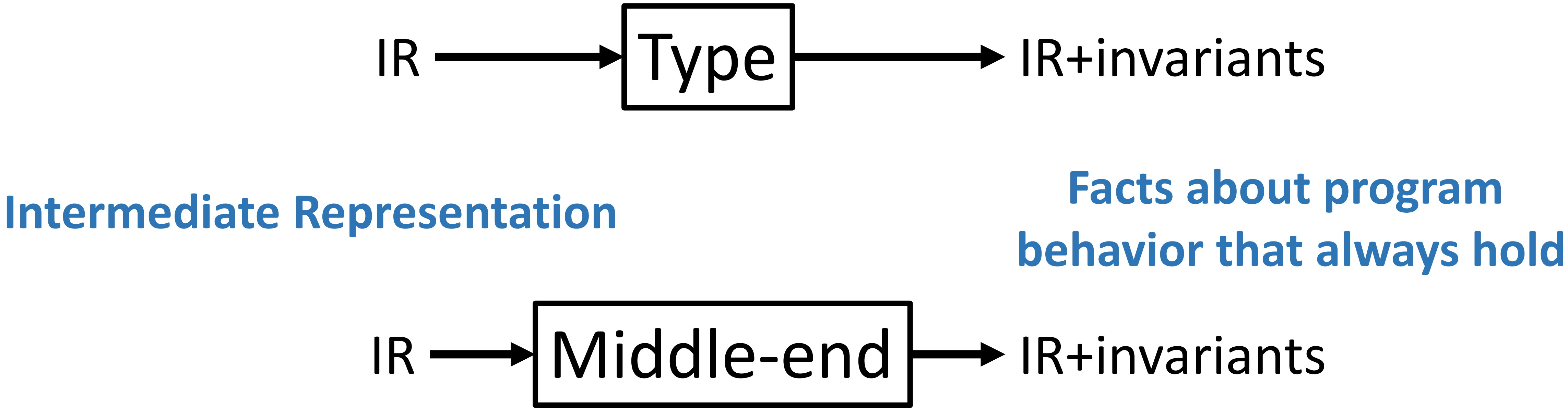
Undergraduate Compilers



This Class

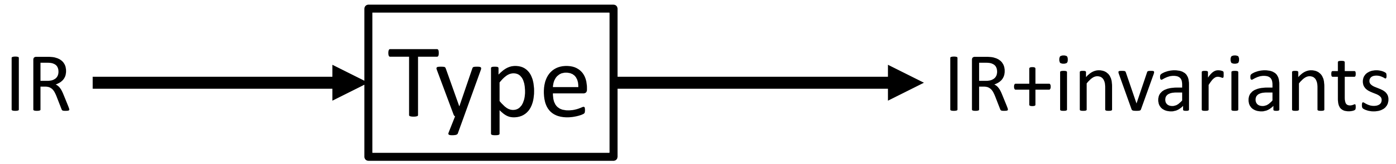


Static Program Analysis



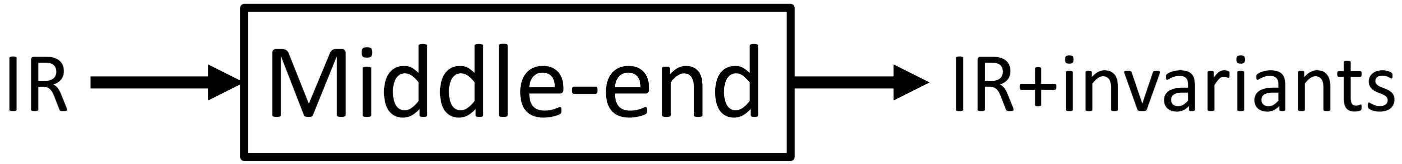
Static Program Analysis

abstract syntax tree,
symbol table, …



x is an "integer"
foo(x) returns an "integer"

control flow graph,
dependence graph,
call graph, …



x+y is always z-10
p and q never point to the same memory
foo() is always called with positive args

Compilers in three parts

- Theory in a controlled environment
  - TIP – Tiny Imperative Language
  - Scala implementation of interpreter and analyses (with holdbacks)
- Practice in the wild
  - `tipc` a compiler from (a subset of) TIP to LLVM bitcode
  - Yours to extend in a class project
- Prompts to drive your exploration and learning
  - Analysis passes in LLVM

A degree of independence will be required

- Theory in a controlled environment
  - TIP is 4500 SLOC of Scala
  - Much of it you will not need to touch or even look at
  - 46 lines marked “`??? //<--- Complete here`”
- Practice in the wild
  - `tipc` is about 1000 SLOC of C++
  - Makes heavy use of LLVM APIs and coding idioms (smart pointers)
  - Uses ANTLR4 grammar and custom visitors for AST construction and code-gen
- There is no TA
  - I can be of help for many issues (I implemented tipc)
  - I don’t use IDEs, so I can’t help with that, but I hear they are great