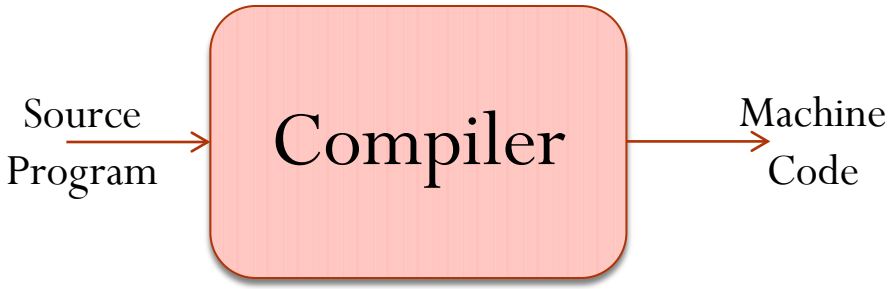
Compilers

Topic: The View from 35,000 Feet

Monsoon 2011, IIIT-H, Suresh Purini

ACK: Some slides are based on Keith Cooper's CS412 at Rice University

High Level View of a Compiler

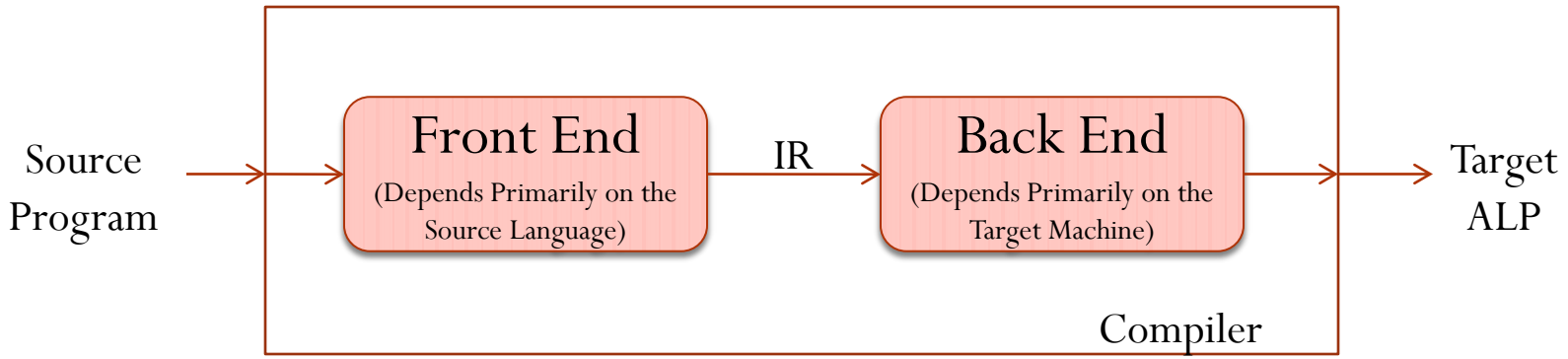


Implications

- Must recognize legal (and illegal) programs
- Must generate correct and efficient code
- Must manage storage of all variables (and code)
- Must agree with OS & linker on format for object code

Big step up from assembly language—use higher level notations

Traditional Two-pass Compiler



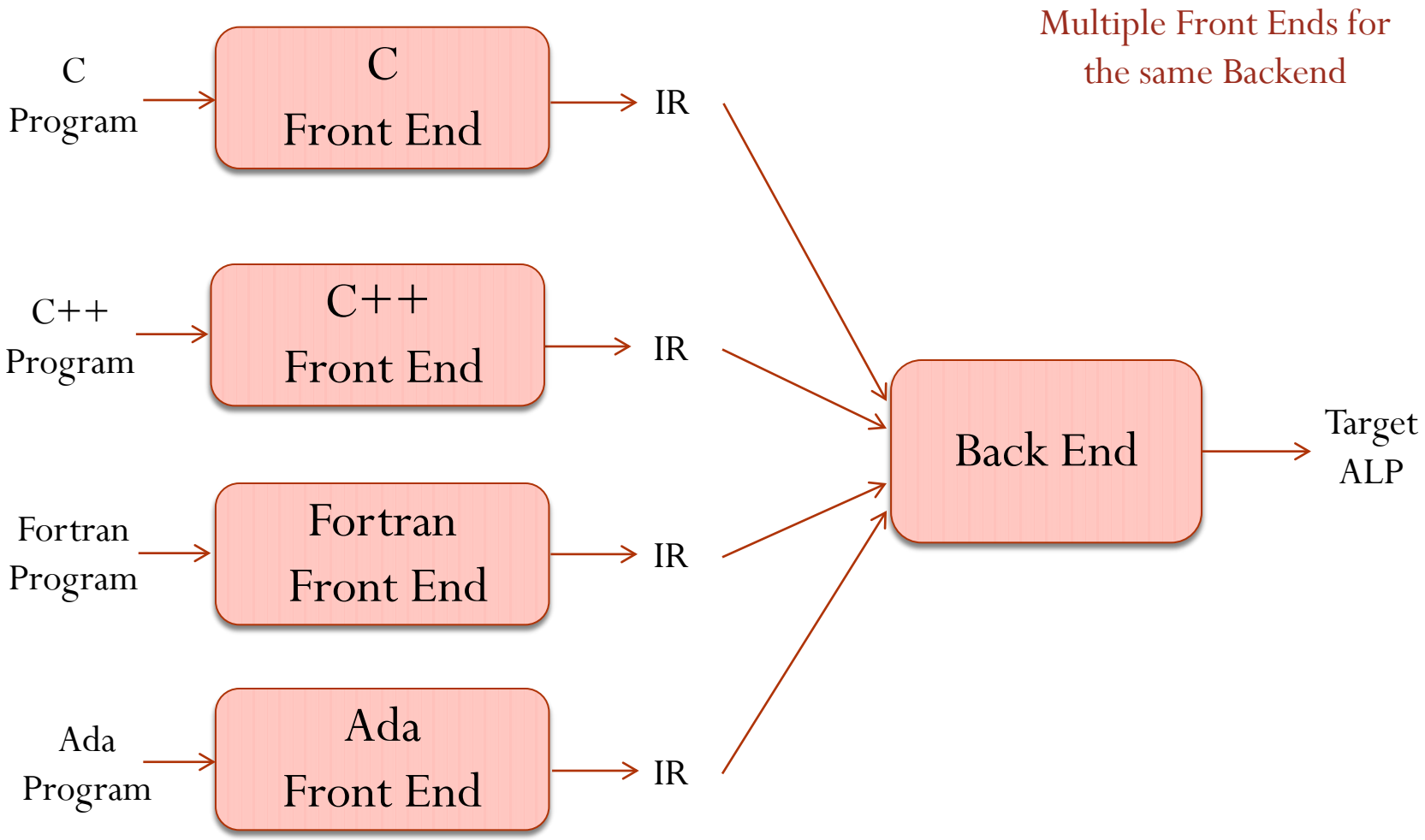
Implications

- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Admits multiple front ends & multiple passes

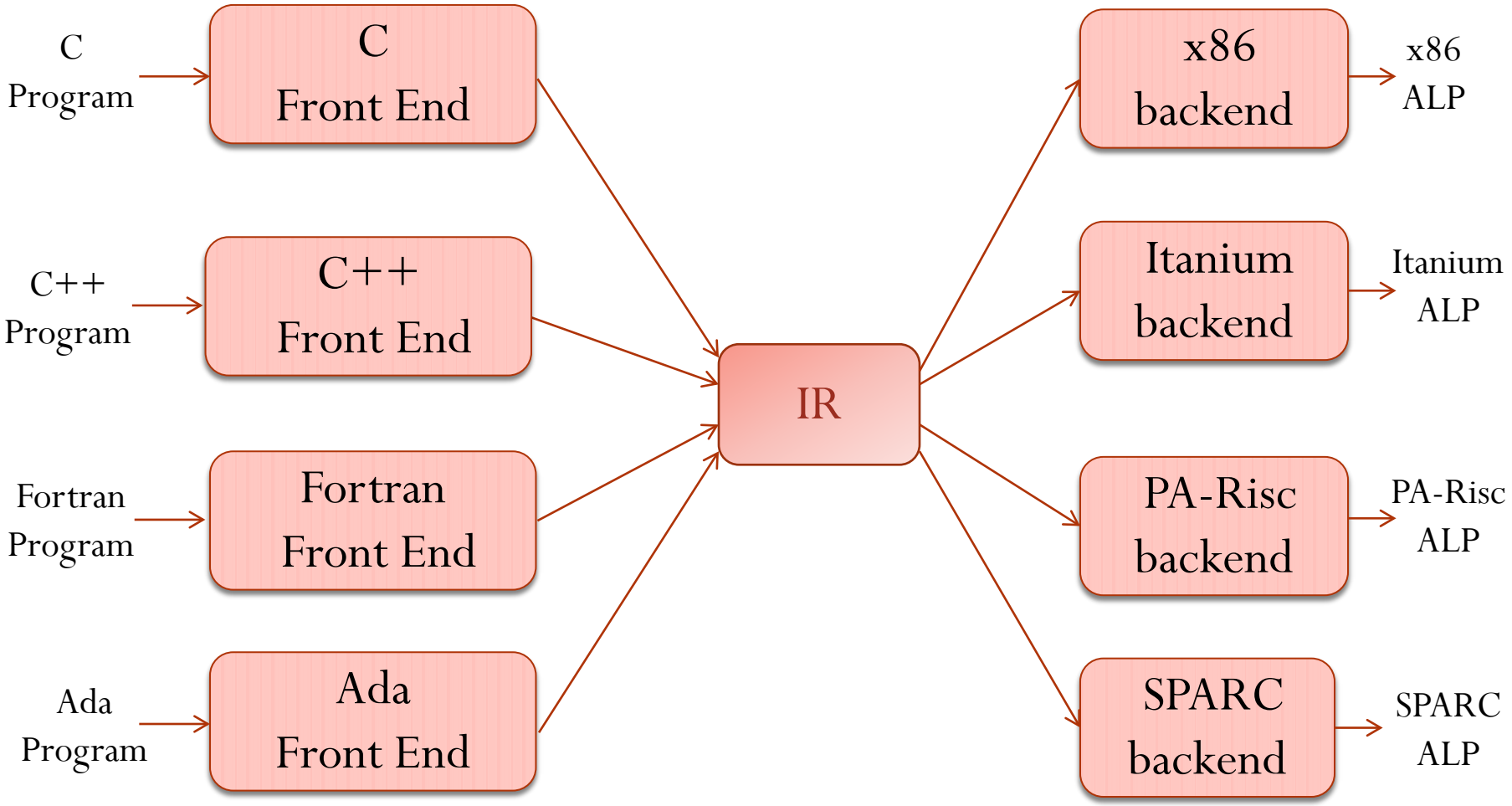
Classic principle from software engineering: Separation of concerns

Typically, front end is O(n) or O(n log n), while back end is NPC

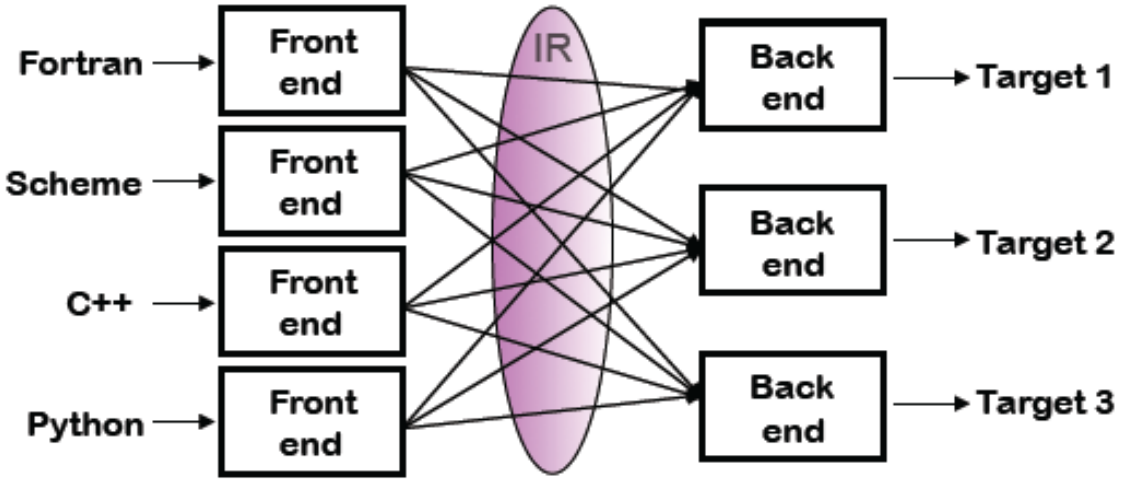
Why IR?


Multiple Front Ends for the same Backend
C Program
C Front End
IR
C++ Program
C++ Front End
IR
Fortran Program
Fortran Front End
IR
Ada Program
Ada Front End
IR
Back End
Target ALP

Why IR?


C
Program
C
Front End
C++
Program
C++
Front End
Fortran
Program
Fortran
Front End
Ada
Program
Ada
Front End
IR
x86
backend
x86
ALP
Itanium
backend
Itanium
ALP
PA-Risc
backend
PA-Risc
ALP
SPARC
backend
SPARC
ALP

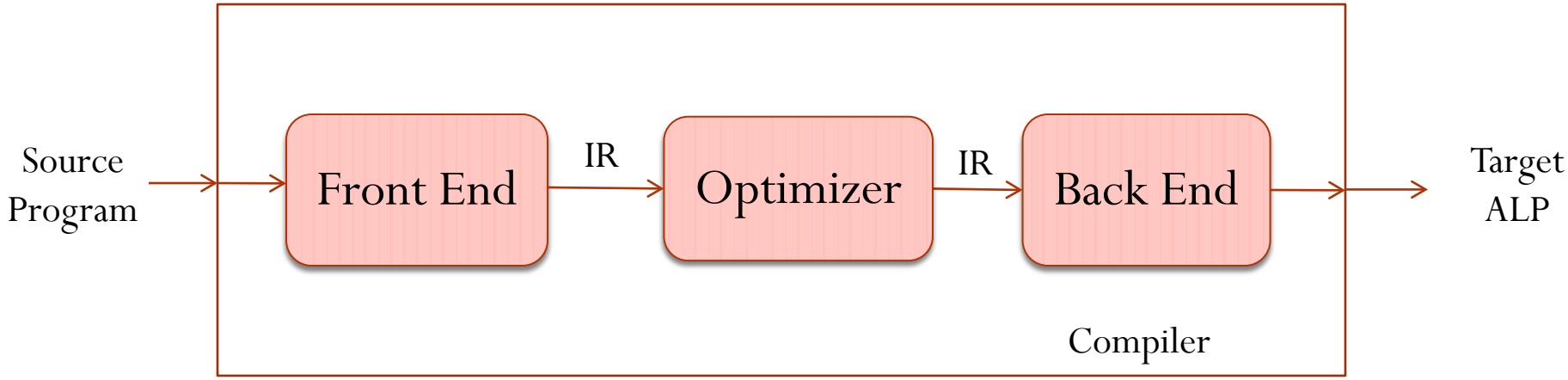
A Common Fallacy



Can we build n x m compilers with n+m components?

- Must encode all language specific knowledge in each front end
- Must encode all features in a single IR
- Must encode all target specific knowledge in each back end
- Successful in systems with assembly level (or lower) IRs
  - e.g. gcc or llvm

Structure of a Three Phase Compiler



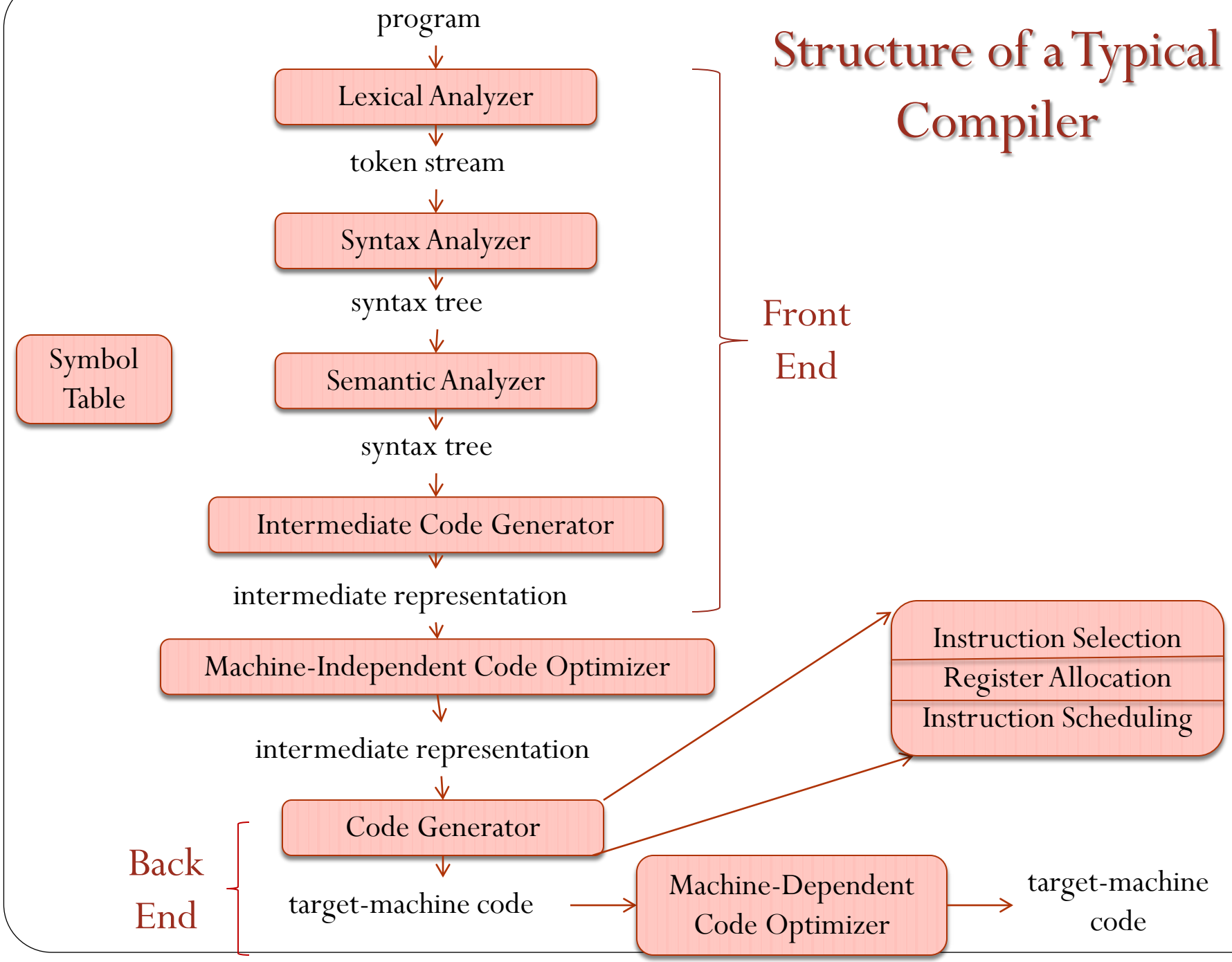
An Optimizer

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …
- Must preserve "meaning" of the code
  - Measured by values of named variables

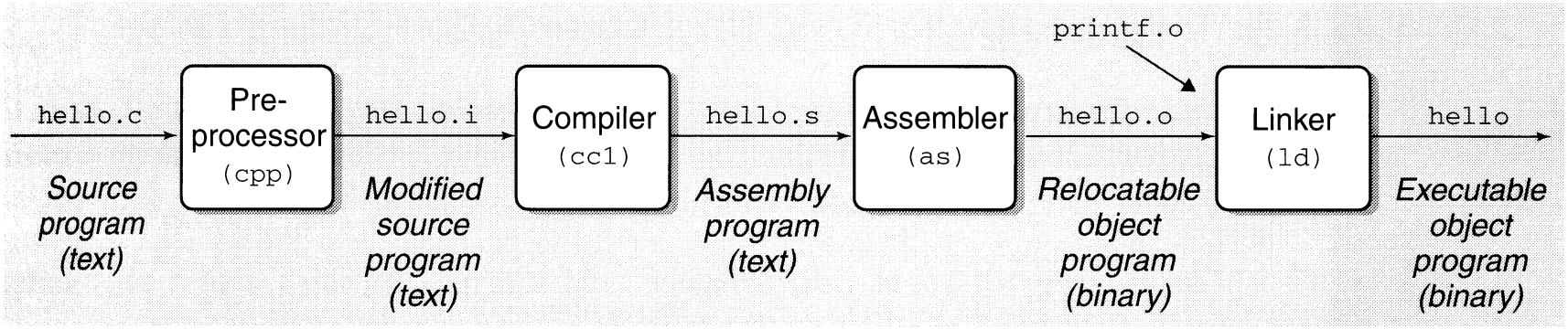
Levels of IRs

- Multiple optimization passes can be made over the IR improving the final program performance.
- Typically multiple IRs at various phases of compilation
  - High level IRs
    - Abstract Syntax Trees – Useful for loop transformations, procedure inlining etc.
  - Intermediate level IRs
    - Three address code, Static Single Assignment form – Useful for optimizations like constant propagation etc.
  - Low level IRs – Good for Low level machine dependent code optimizations

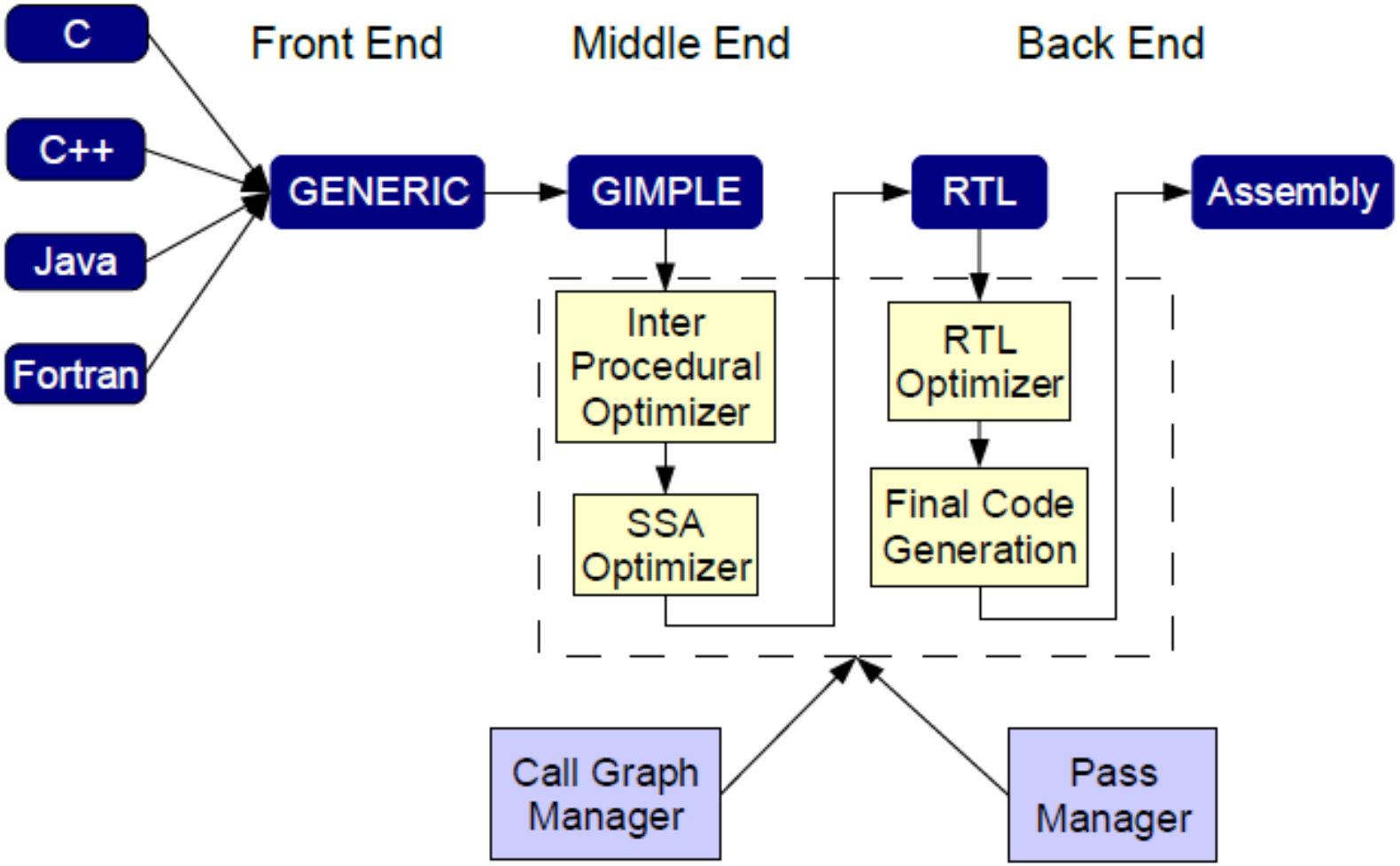
Structure of a Typical Compiler



GCC


printf.o
hello.c
Pre-
processor
(cpp)
hello.i
Compiler
(cc1)
hello.s
Assembler
(as)
hello.o
Linker
(ld)
hello
Source
program
(text)
Modified
source
program
(text)
Assembly
program
(text)
Relocatable
object
program
(binary)
Executable
object
program
(binary)

GCC Compiler Pipeline



GCC – SSA Optimizers

- Operates on GIMPLE
- Around 100 Compiler Passes
  - Vectorization
  - Various loop optimizations
  - Traditional Scalar Optimizations: CCP, DCE, DSE, FRE, PRE, VRP, SRA, jump threading, forward propagation
  - Field-sensitive Points-to alias analysis
  - Pointer checking instrumentation for C/C++
  - Interprocedural analysis and optimizations: CCP, inlining, points-to analysis, pure/const and type escape analysis

GCC: RTL Optimizations

- Around 70 passes
- Operates closer to the target
  - Register Allocation
  - Scheduling
  - Software Pipelining
  - Common Subexpression Elimination
  - Instruction Recombination
  - Mode Switching Reduction
  - Peephole optimizations
  - Machine specific reorgnization

GCC: GENERIC and GIMPLE IRs

GENERIC

```
if (foo (a + b,c))
  c = b++ / a
endif
return c
```

High GIMPLE

```
t1 = a + b
t2 = foo (t1, c)
if (t2 != 0)
  t3 = b
  b = b + 1
  c = t3 / a
endif
return c
```

Low GIMPLE

```
t1 = a + b
t2 = foo (t1, c)
if (t2 != 0) <L1,L2>
L1:
t3 = b
b = b + 1
c = t3 / a
goto L3
L2:
L3:
return c
```

Open 64 Compiler Structure