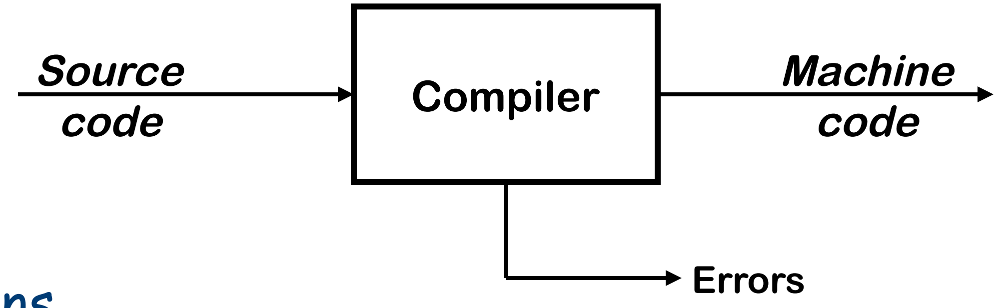
RICE

COMP 412
FALL 2009

The View from 35,000 Feet
Comp 412

High-level View of a Compiler

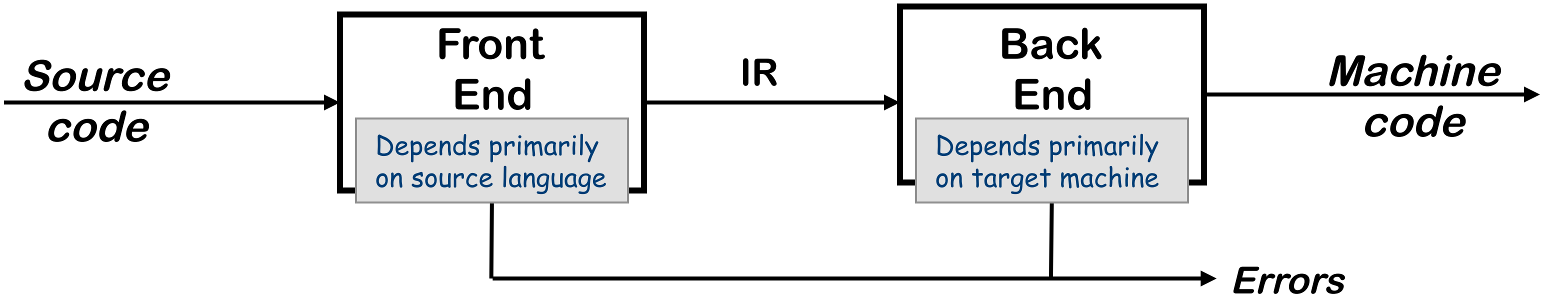


Implications

- Must recognize legal (and illegal) programs
- Must generate correct 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



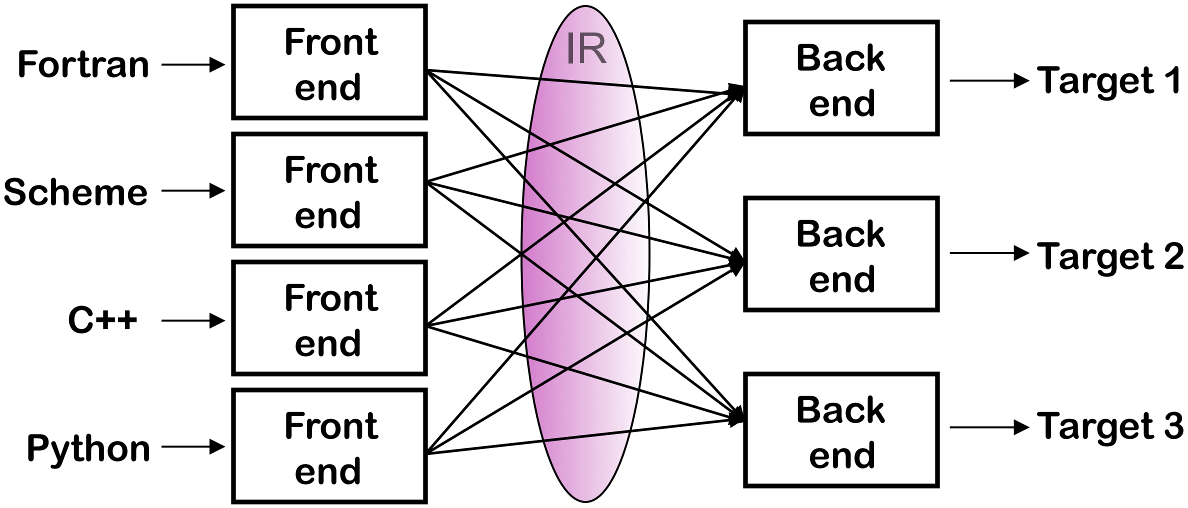
Implications

- 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 *(better code)*

Classic principle from software engineering:
Separation of concerns

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

A Common Fallacy



Fortran

Scheme

C++

Python

Front end

Front end

Front end

Front end

IR

Back end

Back end

Back end

Target 1

Target 2

Target 3

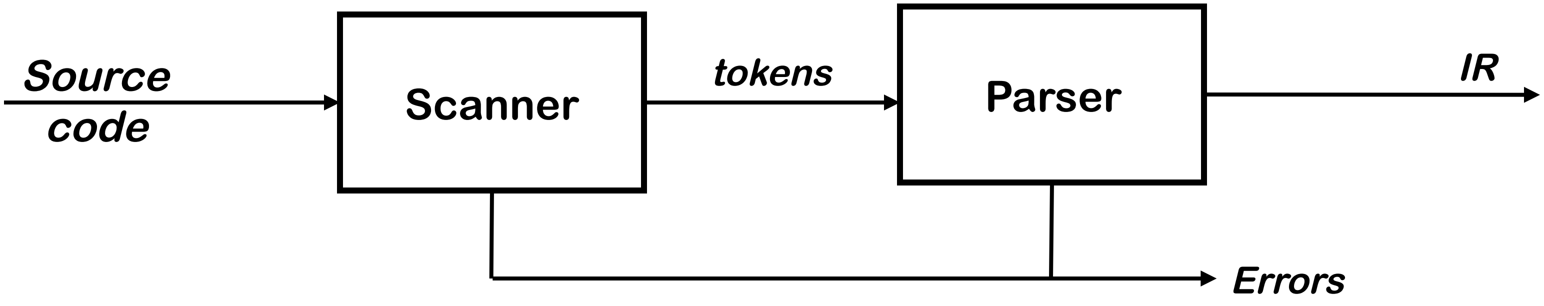
Can we build *n x m* compilers with *n+m* components?

- 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

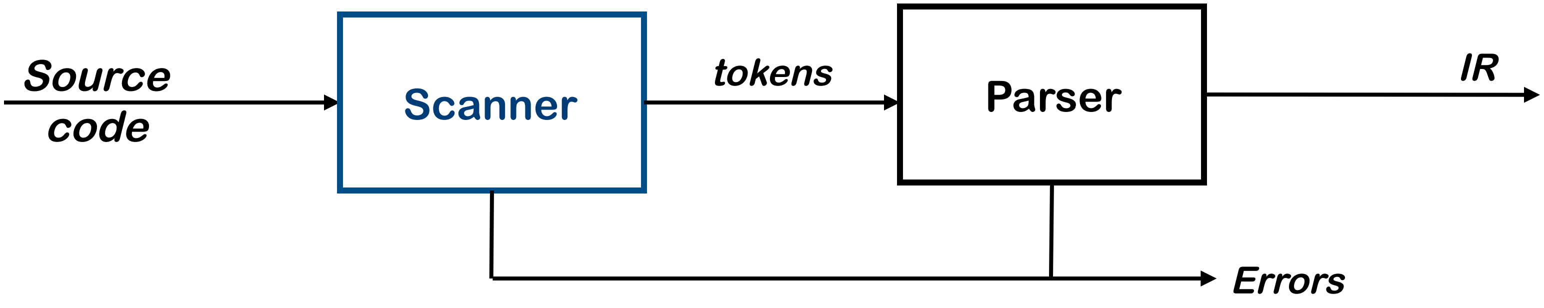
The Front End



Responsibilities

- Recognize legal (& illegal) programs
- Report errors in a useful way
- Produce IR & preliminary storage map
- Shape the code for the rest of the compiler
- Much of front end construction can be automated

The Front End



Source code

Scanner

tokens

Parser

IR

Errors

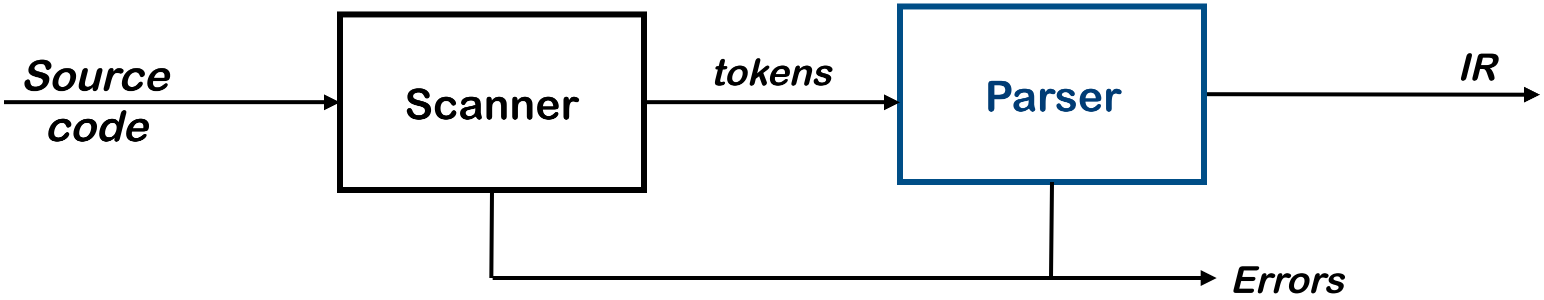
Scanner

- Maps character stream into words—the basic unit of syntax
- Produces pairs — a word & its part of speech
  - x = x + y ; *becomes* <id,x> = <id,x> + <id,y> ;
  - — *word ≅ lexeme, part of speech ≅ token type, pair ≅ a token*
- Typical tokens include *number*, *identifier*, +, –, new, *while*, *if*
- Speed is important

Textbooks advocate automatic scanner generation

Commercial practice appears to be hand-coded scanners

The Front End


Source code
Scanner
tokens
Parser
IR
Errors

Parser

- Recognizes context-free syntax & reports errors
- Guides context-sensitive (“semantic”) analysis (*type checking*)
- Builds IR for source program

See lab 2

Hand-coded parsers are fairly easy to build

Most books advocate using automatic parser generators

The Front End

Context-free syntax is specified with a grammar

$$\begin{aligned} \mathit{SheepNoise} &\rightarrow \mathit{SheepNoise}\ \underline{\text{baa}} \\ &\mid\ \underline{\text{baa}} \end{aligned}$$

This grammar defines the set of noises that a sheep makes under normal circumstances

It is written in a variant of Backus–Naur Form (BNF)

Formally, a grammar $G = (S,N,T,P)$

- S is the *start symbol*
- N is a set of *non-terminal symbols*
- T is a set of *terminal symbols* or *words*
- P is a set of *productions* or *rewrite rules* $(P : N \rightarrow N \cup T)$

(Example due to Dr. Scott K. Warren)

The Front End

Context-free syntax can be put to better use

1. Goal → Expr
2. Expr → Expr Op Term
3. | Term
4. Term → number
5. | id
6. Op → +
7. | -

S = *Goal*

T = { number, id, +, - }

N = { *Goal*, *Expr*, *Term*, *Op* }

P = { 1, 2, 3, 4, 5, 6, 7 }

- This grammar defines simple expressions with addition & subtraction over "number" and "id"
- This grammar, like many, falls in a class called "context-free grammars", abbreviated *CFG*

The Front End

Given a CFG, we can *derive* sentences by repeated substitution

Production	Result
	Goal
1	*Expr*
2	*Expr Op Term*
5	*Expr Op* y
7	*Expr* - y
2	*Expr Op term* - y
4	*Expr Op* 2 - y
6	*Expr* + 2 - y
3	*Term* + 2 - y
5	x + 2 - y

A derivation

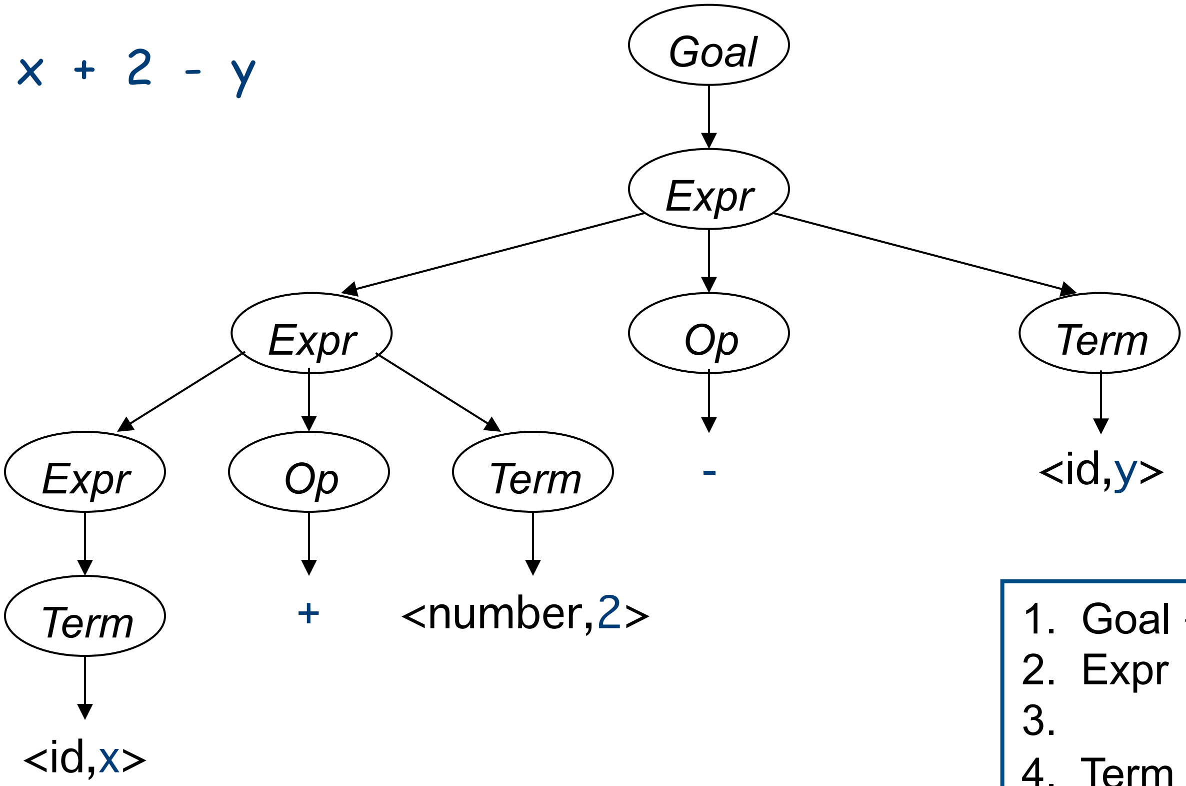
1. Goal → Expr
2. Expr → Expr Op Term
3. | Term
4. Term → number
5. | id
6. Op → +
7. | -

To recognize a valid sentence in some CFG, we reverse this process and build up a *parse*

The Front End

A parse can be represented by a tree (*parse tree* or *syntax tree*)

x + 2 - y

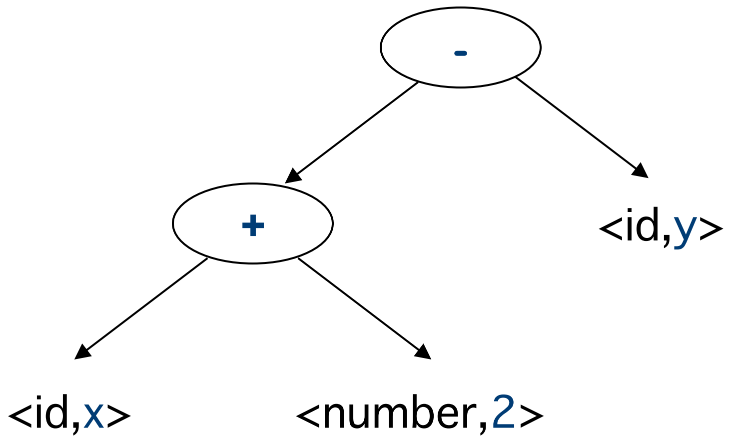


1. Goal → Expr
2. Expr → Expr Op Term
3. | Term
4. Term → number
5. | id
6. Op → +
7. | -

The parse tree contains a lot of unneeded information

The Front End

Compilers often use an *abstract syntax tree* instead of a parse tree



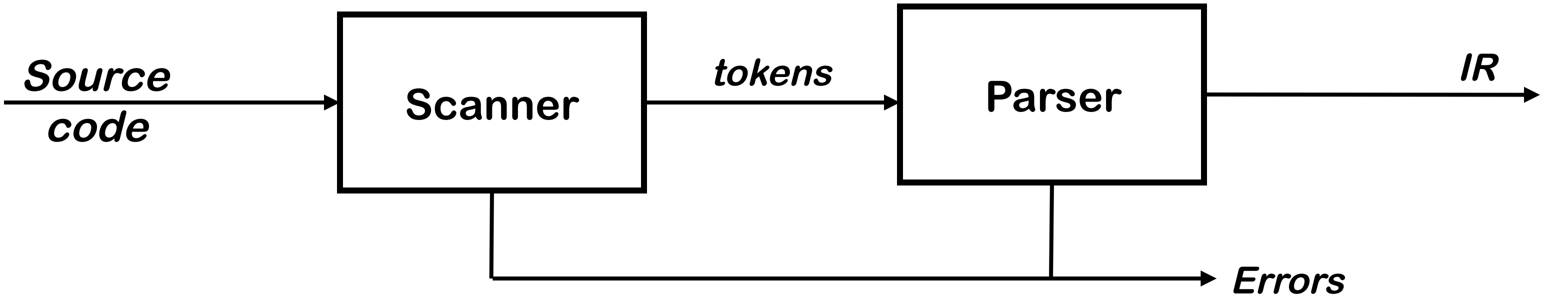
The **AST** summarizes grammatical structure, without including detail about the derivation

This is much more concise

ASTs are one kind of *intermediate representation (IR)*

Some people think that the AST is the "natural" IR.

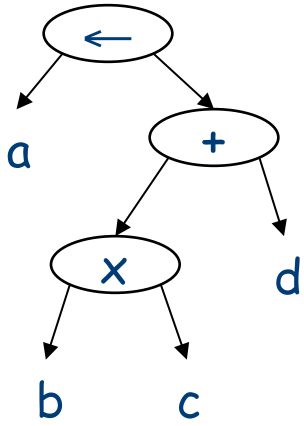
The Front End



Code shape determines many properties of resulting program

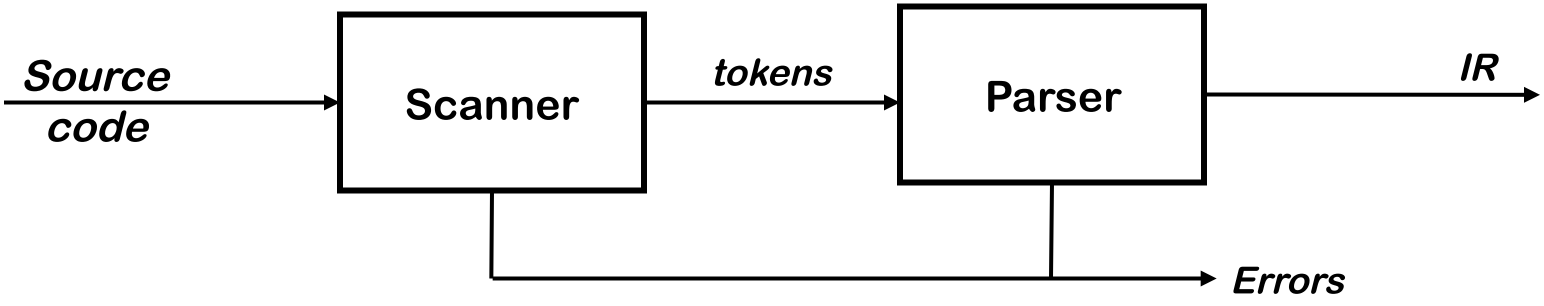
$a \leftarrow b \times c + d$

becomes



Recall the array initialization code in Lecture 1

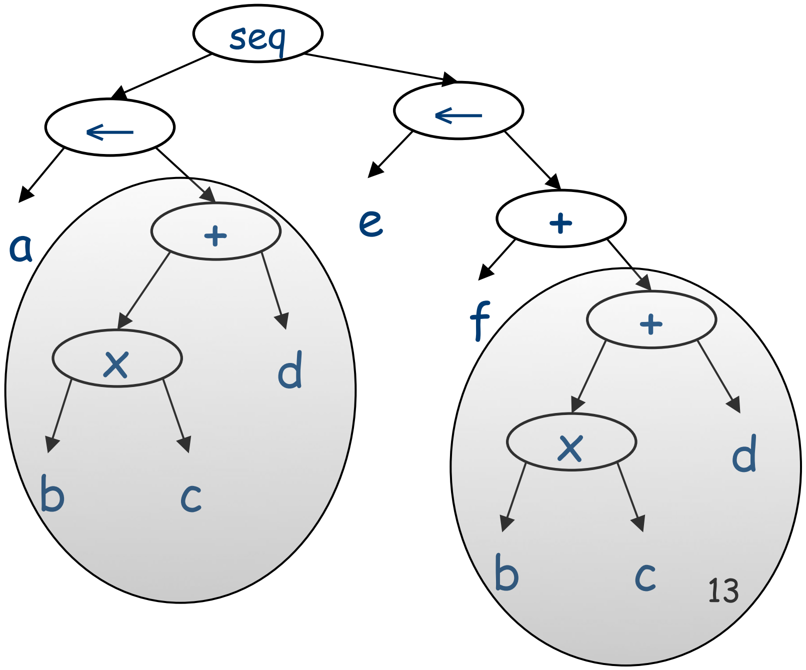
The Front End



Code shape determines many properties of resulting program

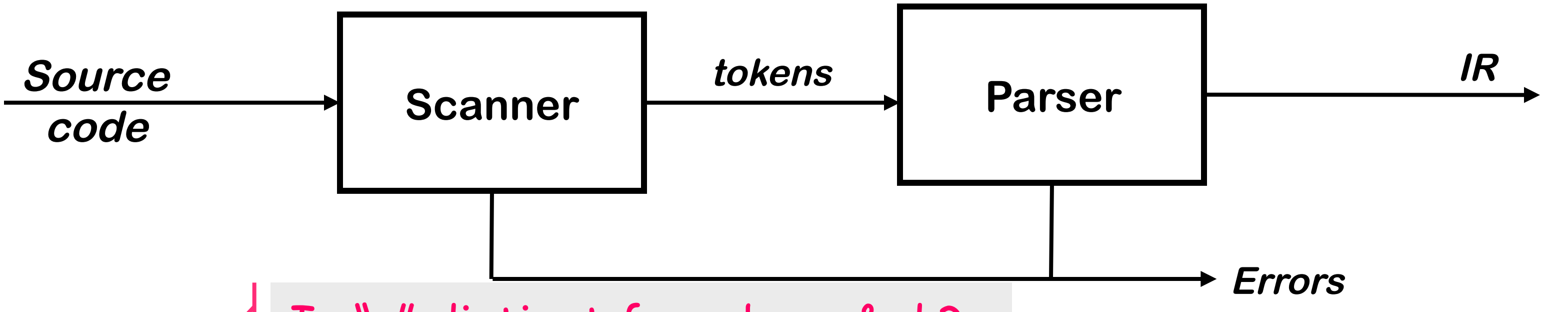
a ← b x c + d
e ← f + b x c + d

becomes



If you turn this AST into code, you will likely get duplication

The Front End



Is "a" distinct from b, c, & d ?

Code shape determines many properties of resulting program

a ← b x c + d
e ← f + b x c + d

becomes

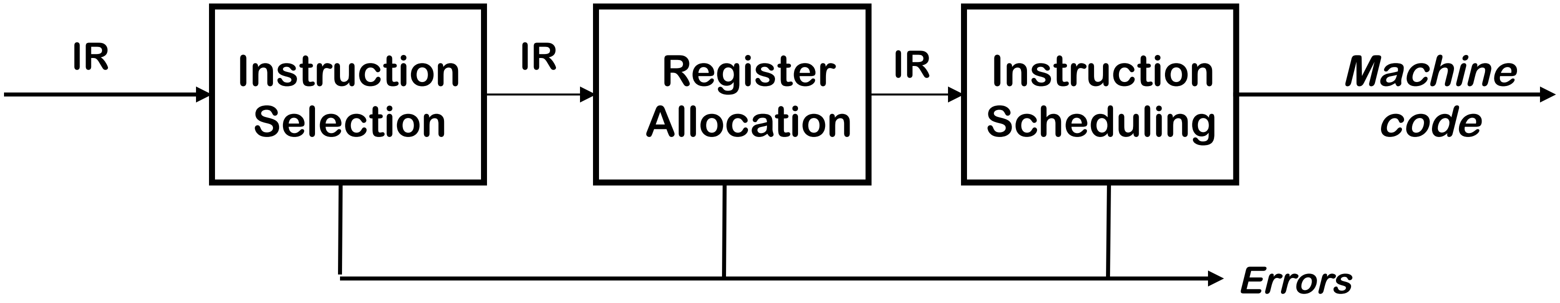
load @b ⇒ r_1
load @c ⇒ r_2
mult r_1,r_2 ⇒ r_3
load @d ⇒ r_4
add r_3,r_4 ⇒ r_5
store r_5 ⇒ @a
load @f ⇒ r_6
add r_5,r_6 ⇒ r_7
store r_7 ⇒ @e

computes b x c + d

reuses b x c + d

We would like to produce this code, but getting it right takes a fair amount of effort ….

The Back End

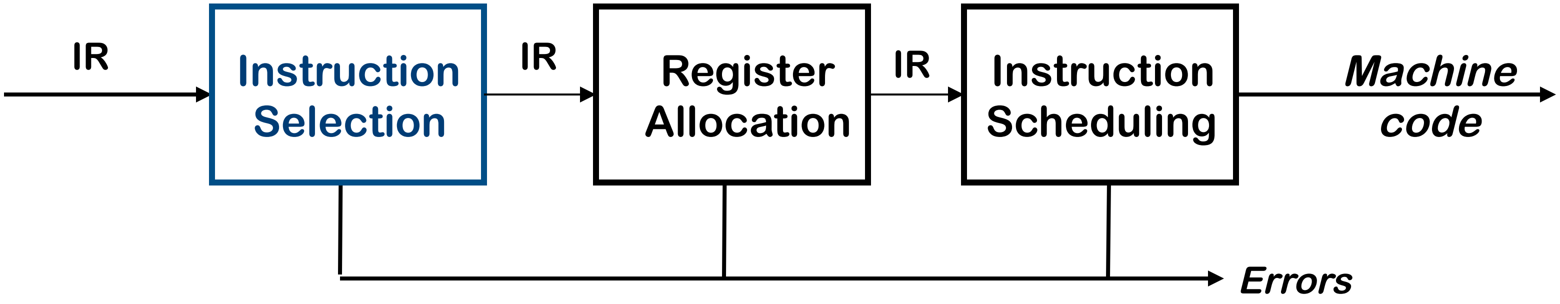


Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces

Automation has been *less* successful in the back end

The Back End



Instruction Selection

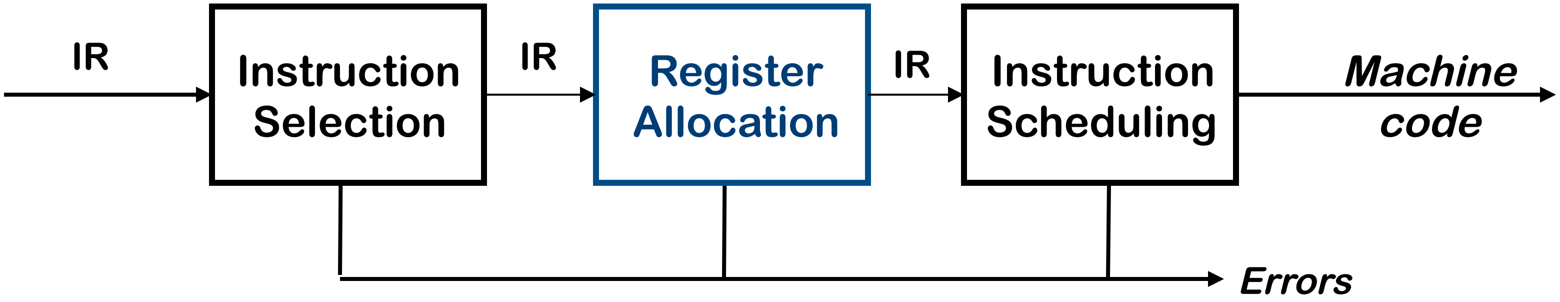
- Produce fast, compact code
- Take advantage of target features such as addressing modes
- Usually viewed as a pattern matching problem
  - *ad hoc* methods, pattern matching, dynamic programming
  - Form of the IR influences choice of technique

This was the problem of the future in 1978

- Spurred by transition from PDP-11 to VAX-11
- Orthogonality of RISC simplified this problem

Standard goal has become "locally optimal" code.

The Back End



Register Allocation

- Have each value in a register when it is used
- Manage a limited set of resources
- Can change instruction choices & insert LOADs & STOREs
- Optimal allocation is NP-Complete in most settings

Compilers approximate solutions to NP-Complete problems

You will become experts over next 3 weeks…

gcc 4.1, O3, PowerPC G4 target

The Back End

Local Register Allocation

→ An example that came to my attention after Friday's lecture

$$
\begin{aligned}
a_0 &\leftarrow a_1 + a_2 \\
a_1 &\leftarrow a_2 + a_0 \\
a_2 &\leftarrow a_0 + a_1 \\
a_3 &\leftarrow a_1 + a_2 \\
a_4 &\leftarrow a_2 + a_3 \\
a_5 &\leftarrow a_3 + a_4
\end{aligned}
$$

This block allocates into 30 registers on the PowerPC G4

$$
\begin{aligned}
a_0 &\leftarrow a_1 + a_2 \\
a_1 &\leftarrow a_2 + a_0 \\
a_2 &\leftarrow a_0 + a_1 \\
a_0 &\leftarrow a_1 + a_2 \\
a_1 &\leftarrow a_2 + a_0 \\
a_2 &\leftarrow a_0 + a_1
\end{aligned}
$$

This block allocates into 20 registers on the PowerPC G4

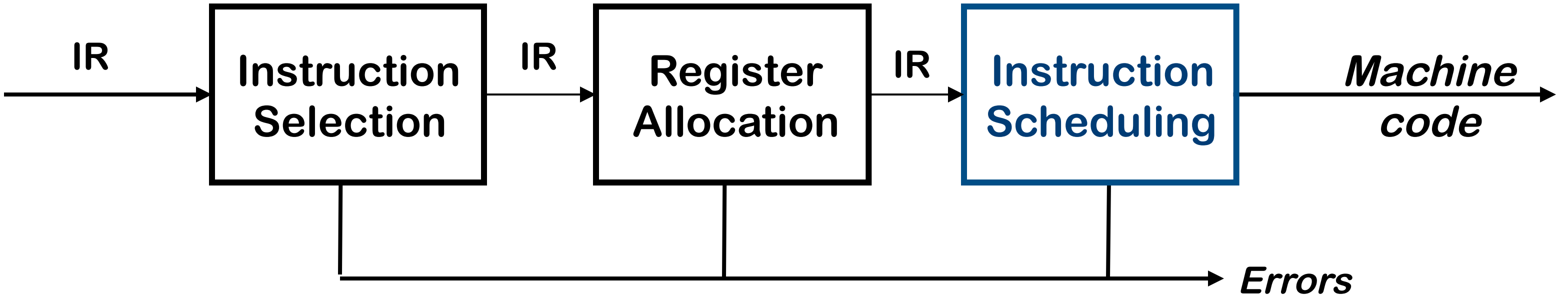
What is the difference?

- Naming
  - Block on left reuses names & confuses its notion of live range

The point of this example is that the allocator confused the distinct live ranges with the same name (e.g., a_0) with the result that it saw the problem as more constrained that it should have been. As a result it began spilling when it had at least 10 spare registers.

Renaming the "virtual registers" so that each definition point targets a unique name would avoid this problem.

The Back End



Instruction Scheduling

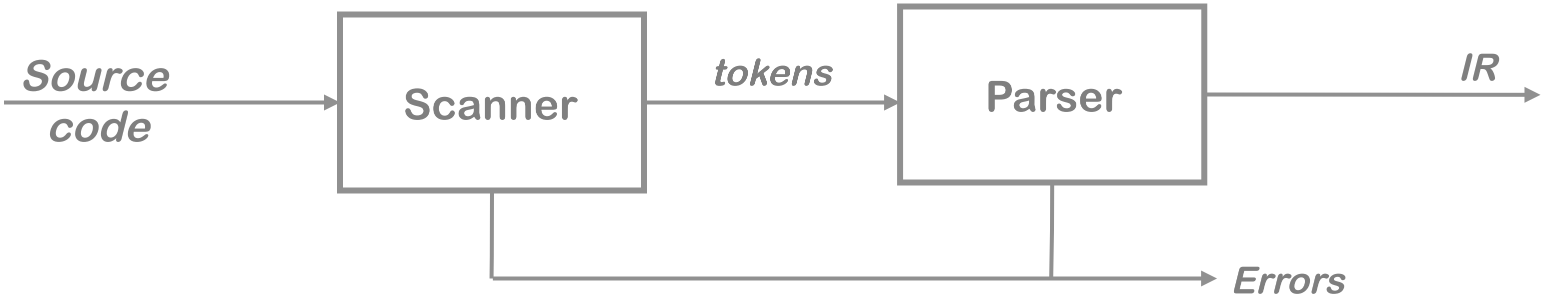
- Avoid hardware stalls and interlocks
- Use all functional units productively
- Can increase lifetime of variables (changing the allocation)

Optimal scheduling is NP-Complete in nearly all cases

Heuristic techniques are well developed

In Lab 3, you will build a pair of schedulers.

The Front End



Source code

Scanner

tokens

Parser

IR

Errors

$a \leftarrow b \times c + d$

$e \leftarrow f + b \times c + d$

becomes

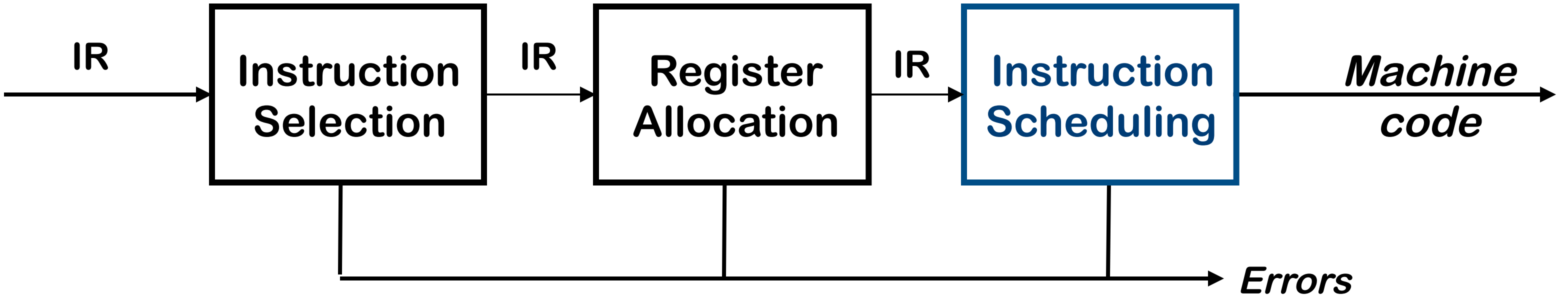
load @b $\Rightarrow r_1$
load @c $\Rightarrow r_2$
mult $r_1, r_2 \Rightarrow r_3$
load @d $\Rightarrow r_4$
add $r_3, r_4 \Rightarrow r_5$
store $r_5 \Rightarrow$ @a
load @f $\Rightarrow r_6$
add $r_5, r_6 \Rightarrow r_7$
store $r_7 \Rightarrow$ @e

computes $b \times c + d$

reuses $b \times c + d$

Recall this example ILOC program from earlier in the lecture?

The Back End


IR

Instruction Selection

IR

Register Allocation

IR

Instruction Scheduling

Machine code

Errors

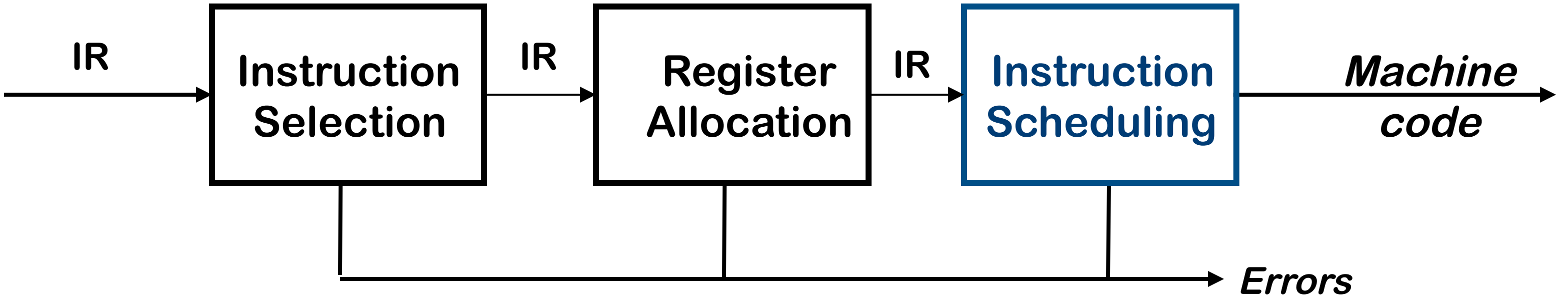
Instruction Scheduling

unit 1	unit 2
load @b $\Rightarrow r_1$	load @c $\Rightarrow r_2$
load @d $\Rightarrow r_4$	load @f $\Rightarrow r_6$
mult $r_1,r_2 \Rightarrow r_3$	nop
add $r_3,r_4 \Rightarrow r_5$	nop
store $r_5 \Rightarrow$ @a	nop
add $r_5,r_6 \Rightarrow r_7$	nop
store $r_7 \Rightarrow$ @e	nop

This schedule aggressively loads values into registers to cover the memory latency.

It finishes the computation as soon as possible (assuming 2 cycles for load & store, 1 cycle for other operations.

The Back End



Instr load)

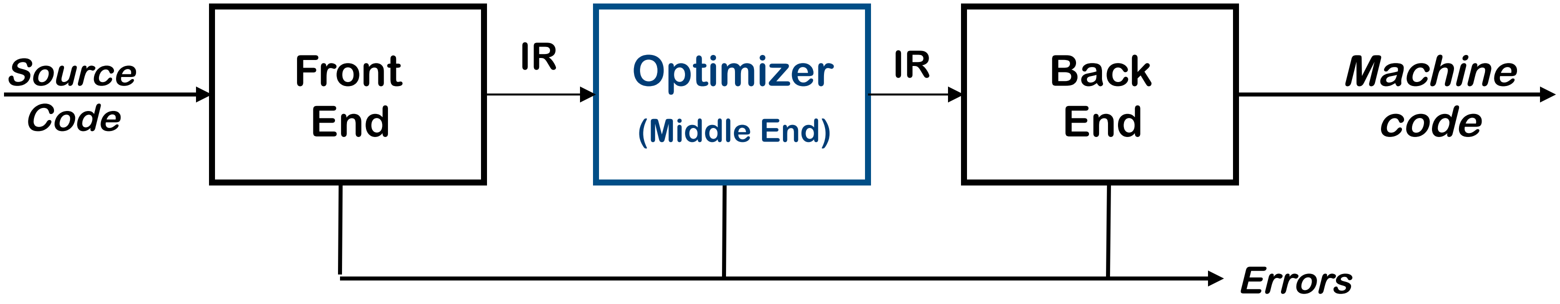
This schedule needs fewer registers but the code is no slower

unit 1	unit 2
load @b ⇒ r_1	load @c ⇒ r_2
load @d ⇒ r_4	load @f ⇒ r_6
mult r_1,r_2 ⇒ r_3	nop
add r_3,r_4 ⇒ r_5	nop
store r_5 ⇒ @a	nop
add r_5,r_6 ⇒ r_7	nop
store r_7 ⇒ @e	nop

unit 1	unit 2
load @b ⇒ r_1	load @c ⇒ r_2
load @d ⇒ r_4	nop
mult r_1,r_2 ⇒ r_3	nop
add r_3,r_4 ⇒ r_5	load @f ⇒ r_6
store r_5 ⇒ @a	nop
add r_5,r_6 ⇒ r_7	nop
store r_7 ⇒ @e	nop

Same set of names, fewer of them are simultaneously live

Traditional Three-part Compiler

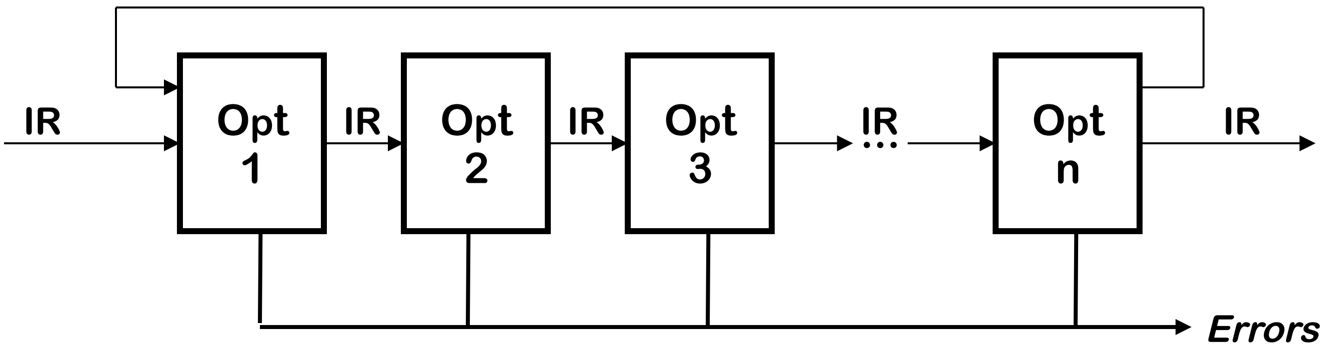


Code Improvement (or *Optimization*)

- 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

Subject of COMP 512, 515, maybe final weeks of 412

The Optimizer (or Middle End)



Modern optimizers are structured as a series of passes

Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

Example

- Optimization of Subscript Expressions

Address(A(I,J)) = address(A(0,0)) + J * (column size) + I

Does the user realize that a multiplication is generated here?

```
DO I = 1, M
      A(I,J) = A(I,J) + C
ENDDO
```

Strength reduction →

```
compute addr(A(0,J))
DO I = 1, M
      add k to get addr(A(I,J))
      A(I,J) = A(I,J) + C
ENDDO
```

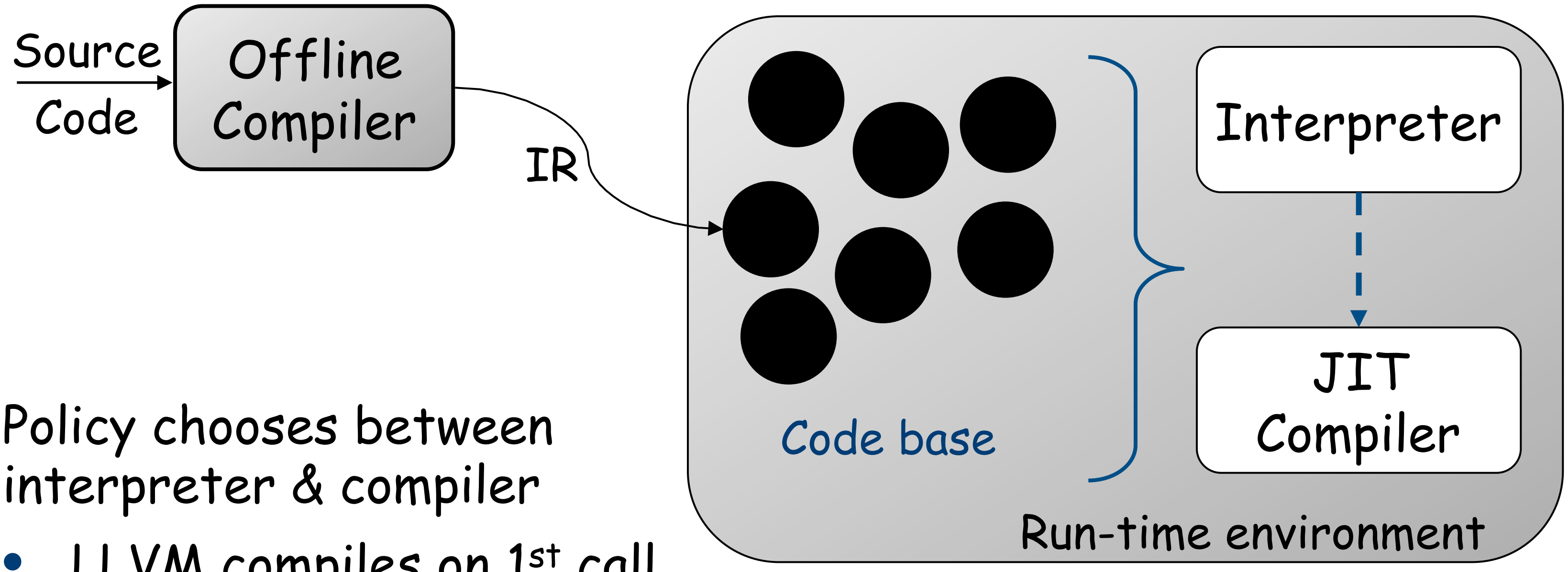
Wrong IR abstraction can hide an opportunity like this one (LLVM)

Role of the Run-time System

- Memory management services
  - Allocate
    - → In the heap or in an activation record (*stack frame*)
  - Deallocate
  - Collect garbage
- Run-time type checking
- Error processing
- Interface to the operating system
  - Input and output
- Support of parallelism
  - Parallel thread initiation
  - Communication and synchronization

Run-time Compilation

Systems such as HotSpot, Jalapeno, and Dynamo deploy compiler and optimization techniques *at run-time*



Source Code

Offline Compiler

IR

Code base

Interpreter

JIT Compiler

Run-time environment

Policy chooses between interpreter & compiler

- LLVM compiles on 1st call
- Dynamo optimizes on 50th execution

Next Class

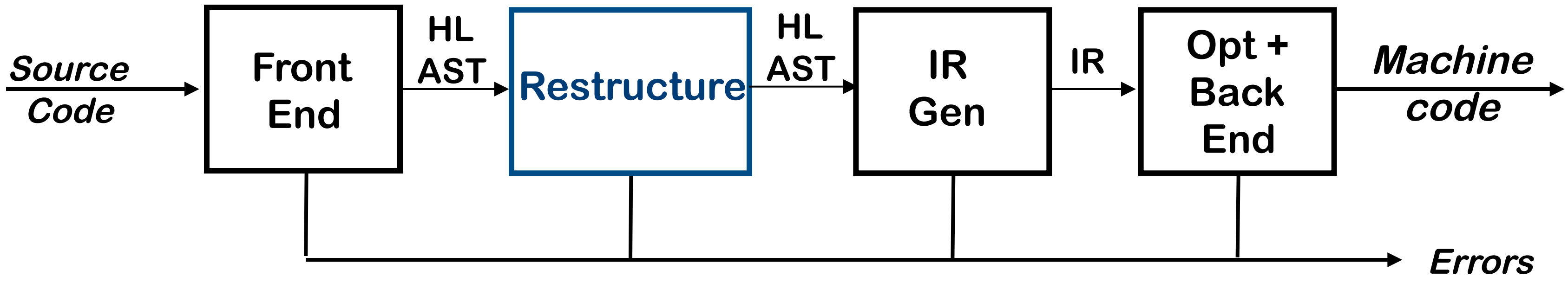
- Introduction to Local Register Allocation
- Announcements:
  - Specs for Lab 1 available Friday (8/28/2009)
    - → Due Sept 16 (documentation 1 day later)
    - → Practice blocks and simulator will be available
    - → Grading blocks will be hidden from you

NOT THIS YEAR –

Next Lecture is

An Introduction to Lexical Analysis (Scanning)

Modern Restructuring Compiler



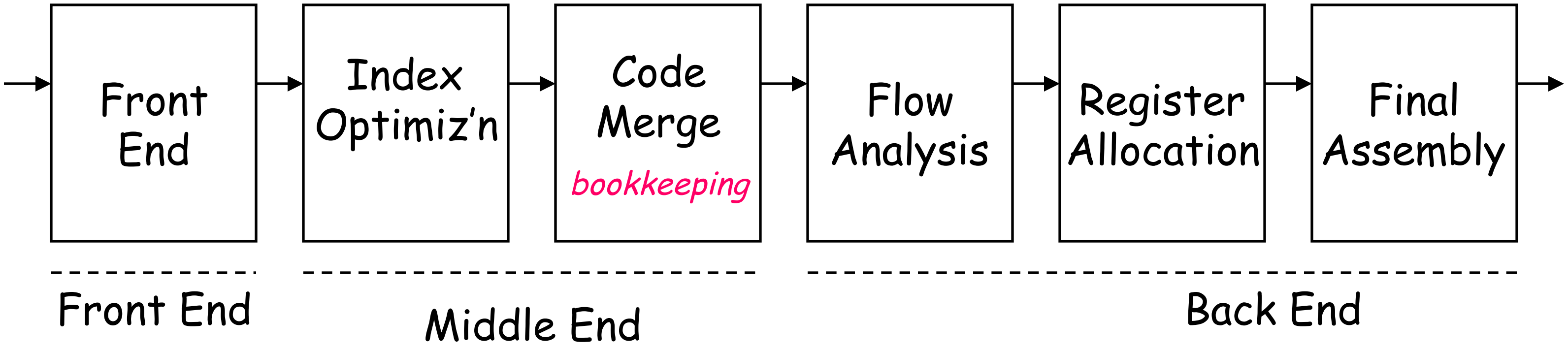
Typical Restructuring Transformations:

- Blocking for memory hierarchy and register reuse
- Vectorization
- Parallelization
- All based on dependence
- Also full and partial inlining

Subject of COMP 515

Classic Compilers

1957: The FORTRAN Automatic Coding System



Front End → Index Optimiz'n → Code Merge *bookkeeping* → Flow Analysis → Register Allocation → Final Assembly

Front End

Middle End

Back End

- Six passes in a fixed order
- Generated good code

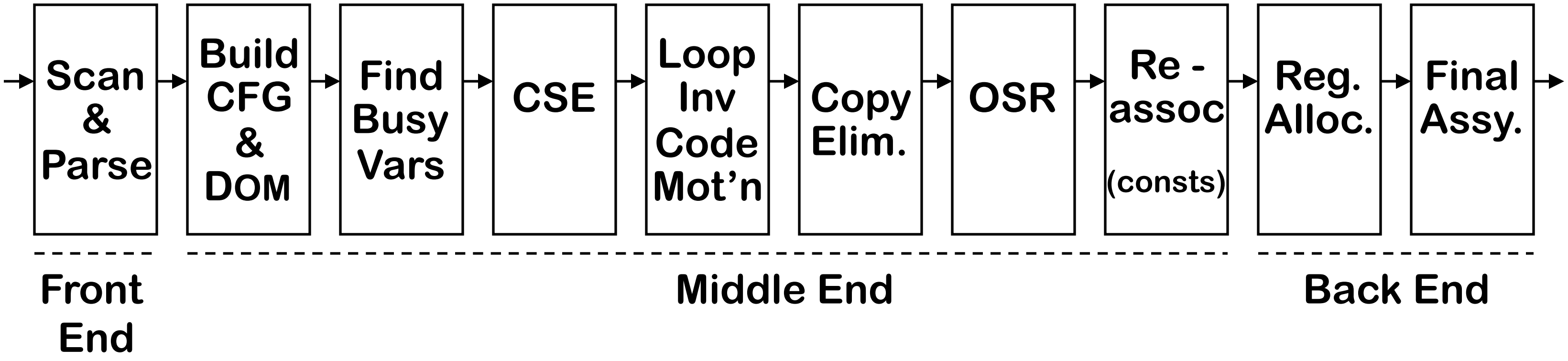
  Assumed unlimited index registers

  Code motion out of loops, with ifs and gotos

  Did flow analysis & register allocation

Classic Compilers

1969: IBM's FORTRAN H Compiler



Scan & Parse → Build CFG & DOM → Find Busy Vars → CSE → Loop Inv Code Mot'n → Copy Elim. → OSR → Re - assoc (consts) → Reg. Alloc. → Final Assy.

Front End | Middle End | Back End

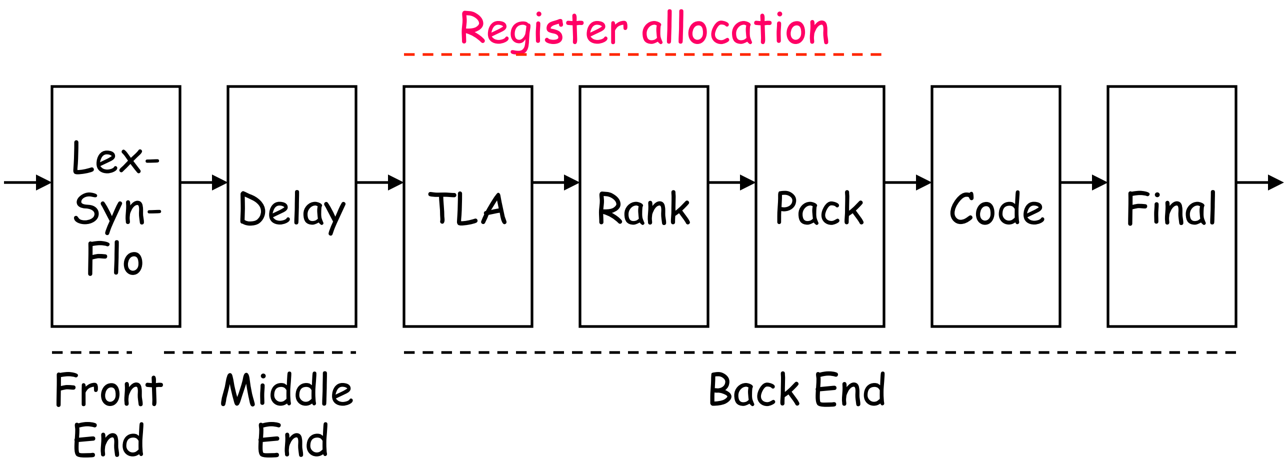
- Used low-level IR (quads), identified loops with dominators
- Focused on optimizing loops ("inside out" order)

  Passes are familiar today

- Simple front end, simple back end for IBM 370

Classic Compilers

1975: BLISS-11 compiler (Wulf *et al.*, CMU)



- The great compiler for the PDP-11
- Seven passes in a fixed order
- Focused on code shape & instruction selection

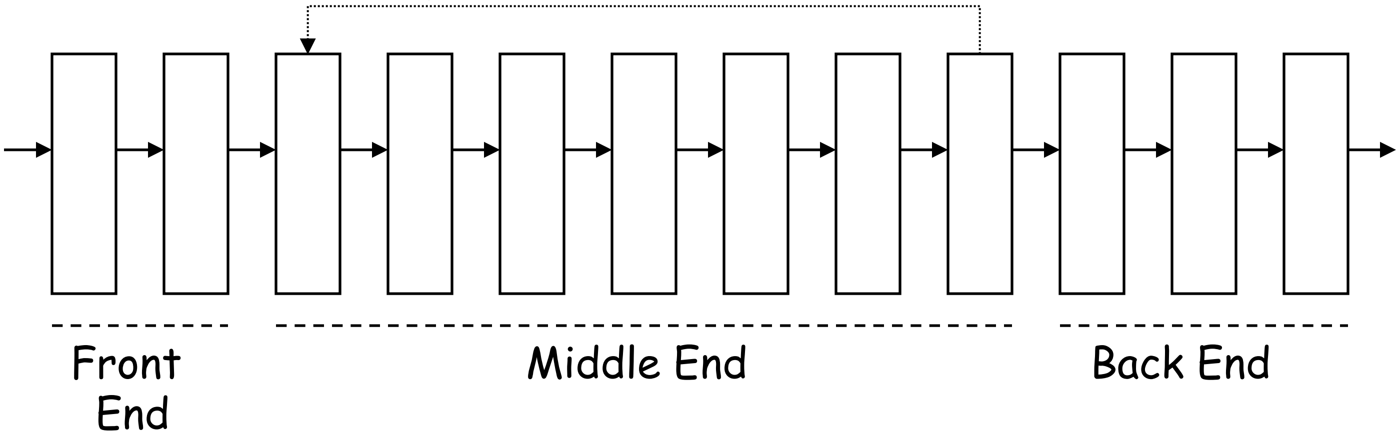
  LexSynFlo did preliminary flow analysis

  Final included a grab-bag of peephole optimizations

Basis for early VAX & Tartan Labs compilers

Classic Compilers

1980: IBM's PL.8 Compiler



- Many passes, one front end, several back ends
- Collection of 10 or more passes

  Repeat some passes and analyses

  Represent complex operations at 2 levels

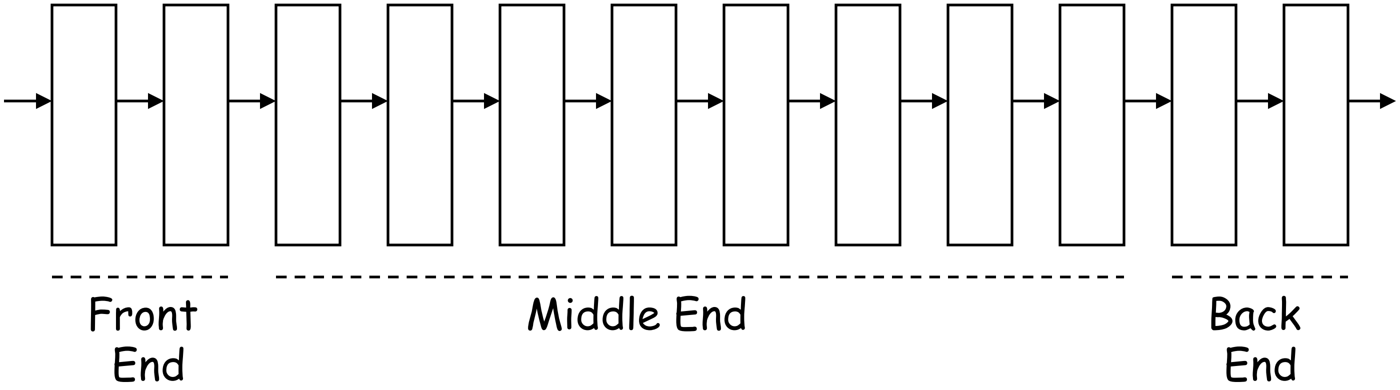
  Below machine-level IR

Dead code elimination
Global cse
Code motion
Constant folding
Strength reduction
Value numbering
Dead store elimination
Code straightening
Trap elimination
Algebraic reassociation

Multi-level IR has become common wisdom

Classic Compilers

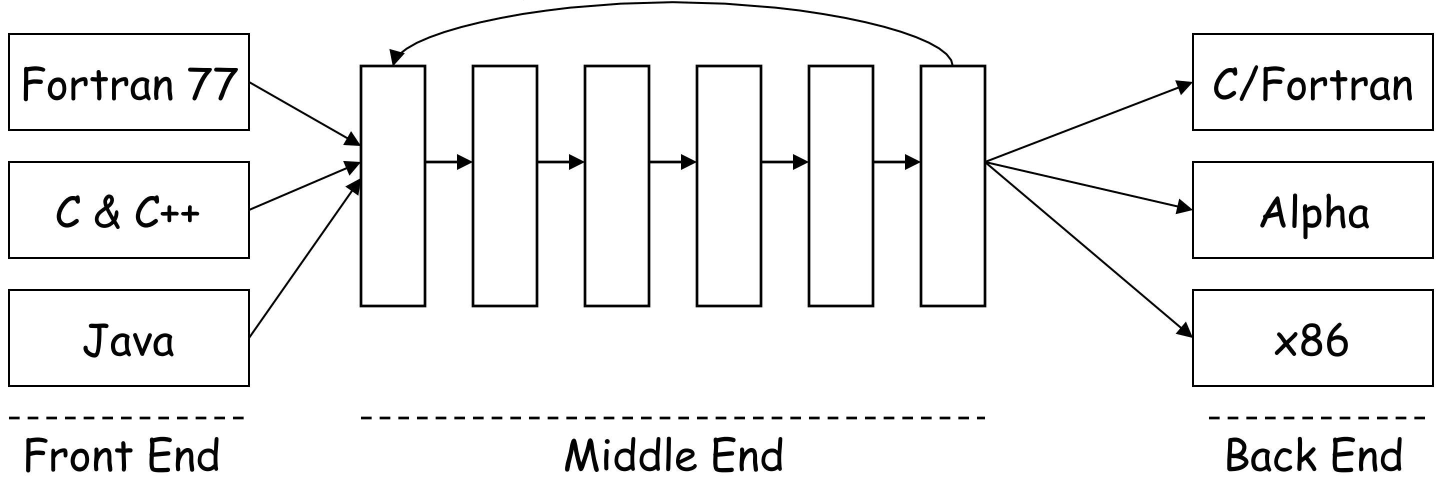
1986: HP's PA-RISC Compiler



- Several front ends, an optimizer, and a back end
- Four fixed-order choices for optimization (9 passes)
- Coloring allocator, instruction scheduler, peephole optimizer

Classic Compilers

1999: The SUIF Compiler System



Another classically-built compiler

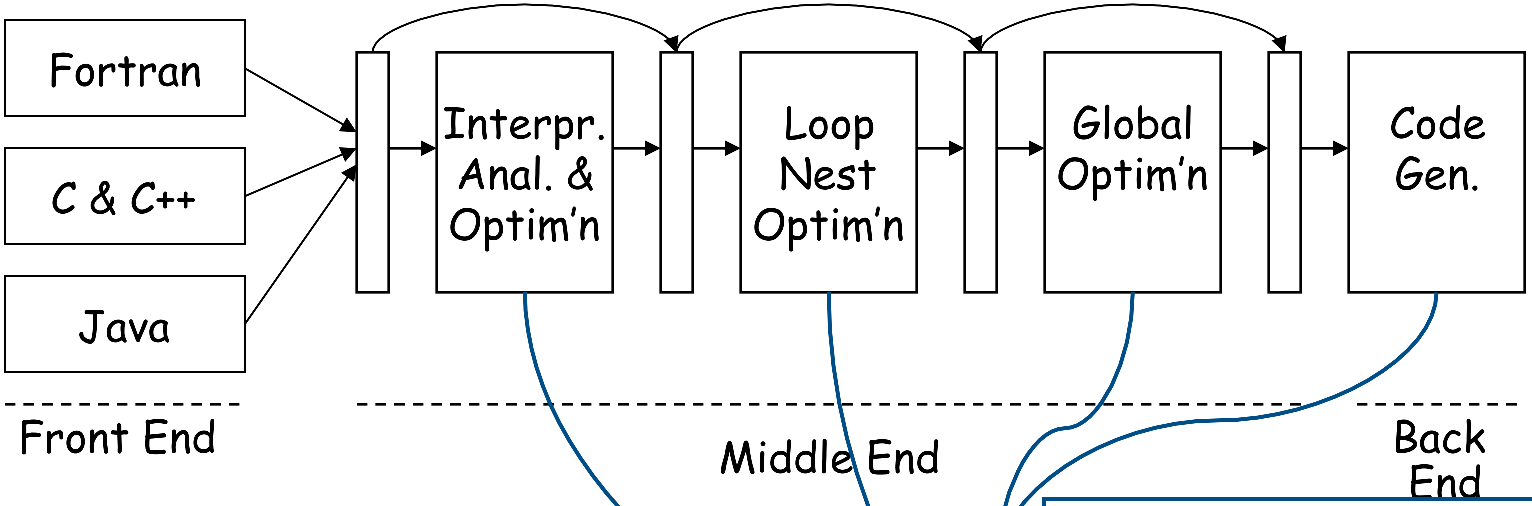
- 3 front ends, 3 back ends
- 18 passes, configurable order
- Two-level IR (High SUIF, Low SUIF)
- Intended as research infrastructure

Data dependence analysis
Scalar & array privitization
Reduction recognition
Pointer analysis
Affine loop transformations
Blocking
Capturing object definitions
Virtual function call elimination
Garbage collection

SSA construction
Dead code elimination
Partial redundancy elimination
Constant propagation
Global value numbering
Strength reduction
Reassociation
Instruction scheduling
Register allocation

Classic Compilers

2000: The SGI Pro64 Compiler (now Open64)



Open source compiler for IA 64

- 3 front ends, 1 back end
- Five-levels of IR
- Gradual lowering of abstraction level

Interprocedural

Classic analysis

Inlining (user & library code)

Cloning (constants & locality)

Dead function elimination

Dead variable elimination

Loop Nest Optimization

Dependence analysis

Loop transformations (fission, fusion, interchange, peeling, tiling, unroll & jam)

Array privitization

Global Optimization

SSA-based analysis & opt'n

Constant propagation, PRE, OSR+LFTR, DVNT, DCE

(also used by other phases)

Code Generation

If conversion & predication

Code motion

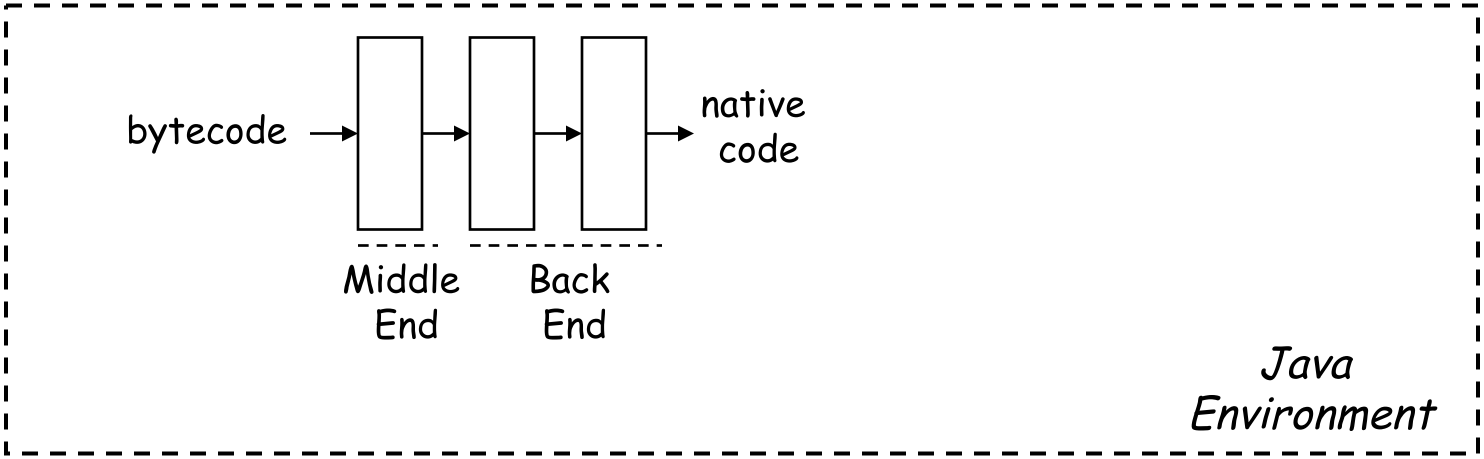
Scheduling (inc. sw pipelining)

Allocation

Peephole optimization

Classic Compilers

Even a modern JIT fits the mold, albeit with fewer passes



- Front end tasks are handled elsewhere
- Few (if any) optimizations
  - Avoid expensive analysis
  - Emphasis on generating native code
  - Compilation must be a priori profitable