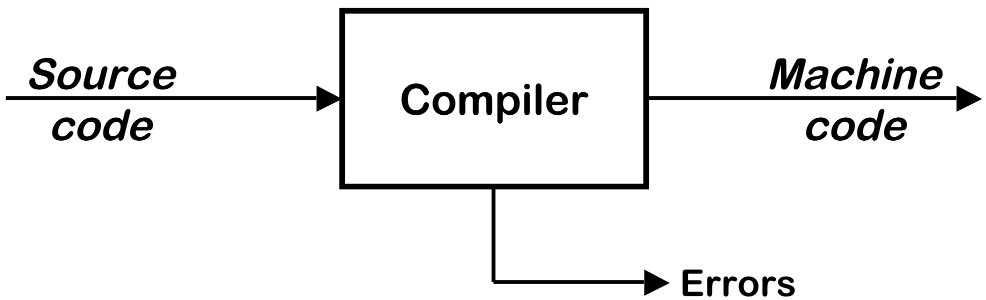
The View from 35,000 Feet

COMP 412
Rice University
Fall 2004

High-level View of a Compiler

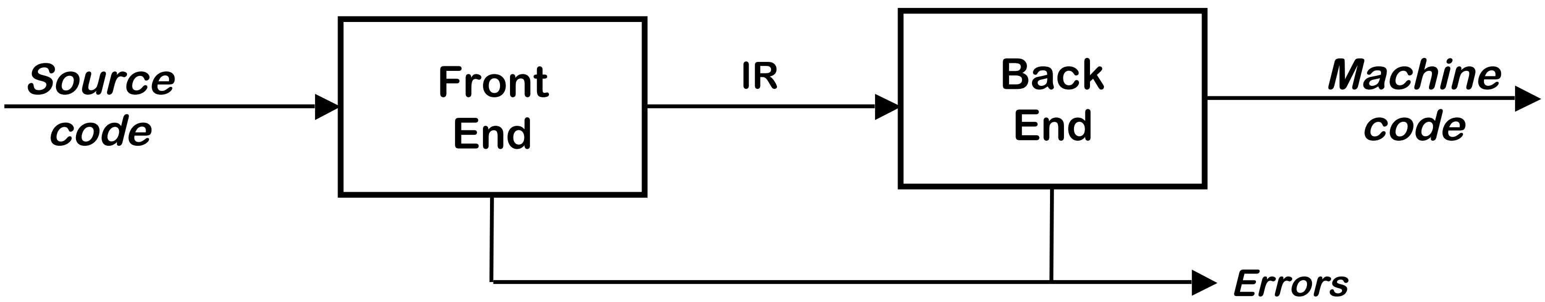


Implications

- 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

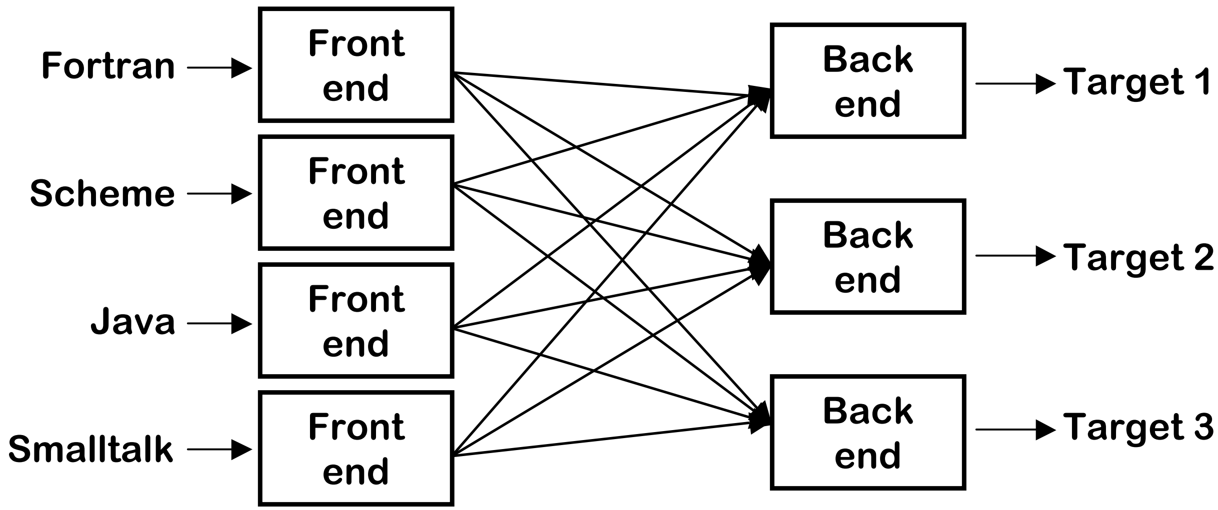


Implications

- 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)*

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

A Common Fallacy

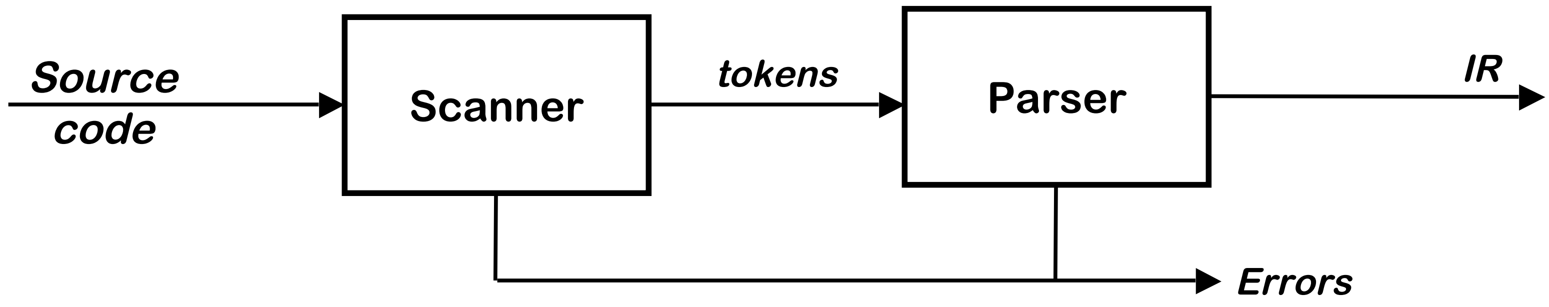


Can we build *n x m* compilers with *n+m* components?

- 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

Limited success in systems with very low-level IRs

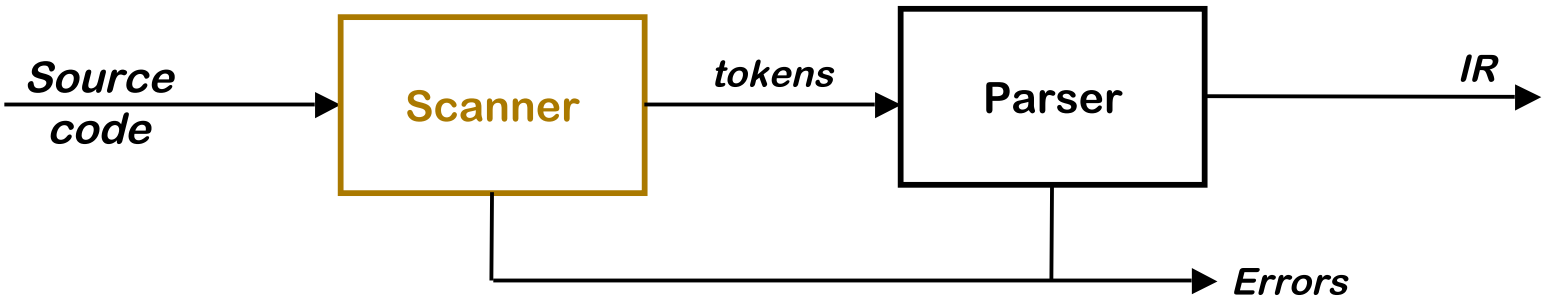
The Front End



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

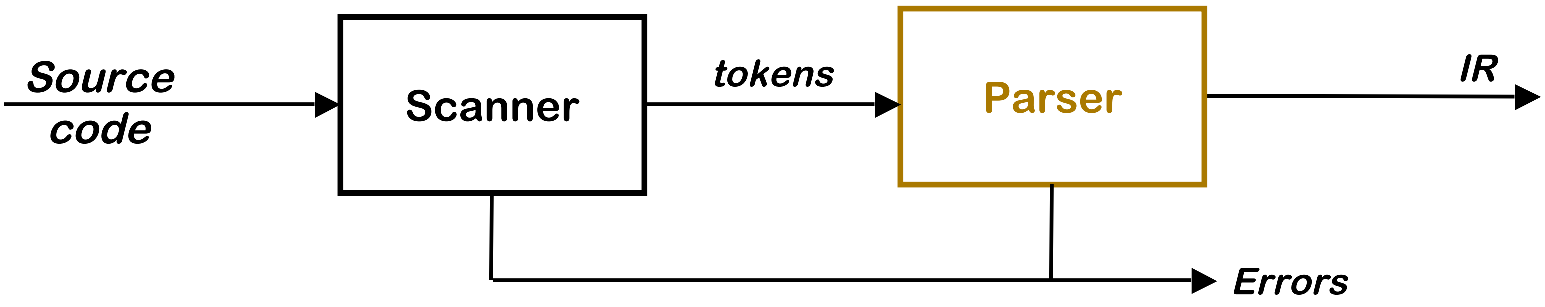


Scanner

- 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*
  - In casual speech, we call the pair a *token*
- Typical tokens include *number*, *identifier*, +, –, new, *while*, *if*
- Scanner eliminates white space (*including comments*)
- Speed is important

The Front End



Parser

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

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

$$\begin{aligned} \textit{SheepNoise} \rightarrow\ & \textit{SheepNoise}\ \underline{\text{baa}} \\ |\ & \underline{\text{baa}} \end{aligned}$$

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

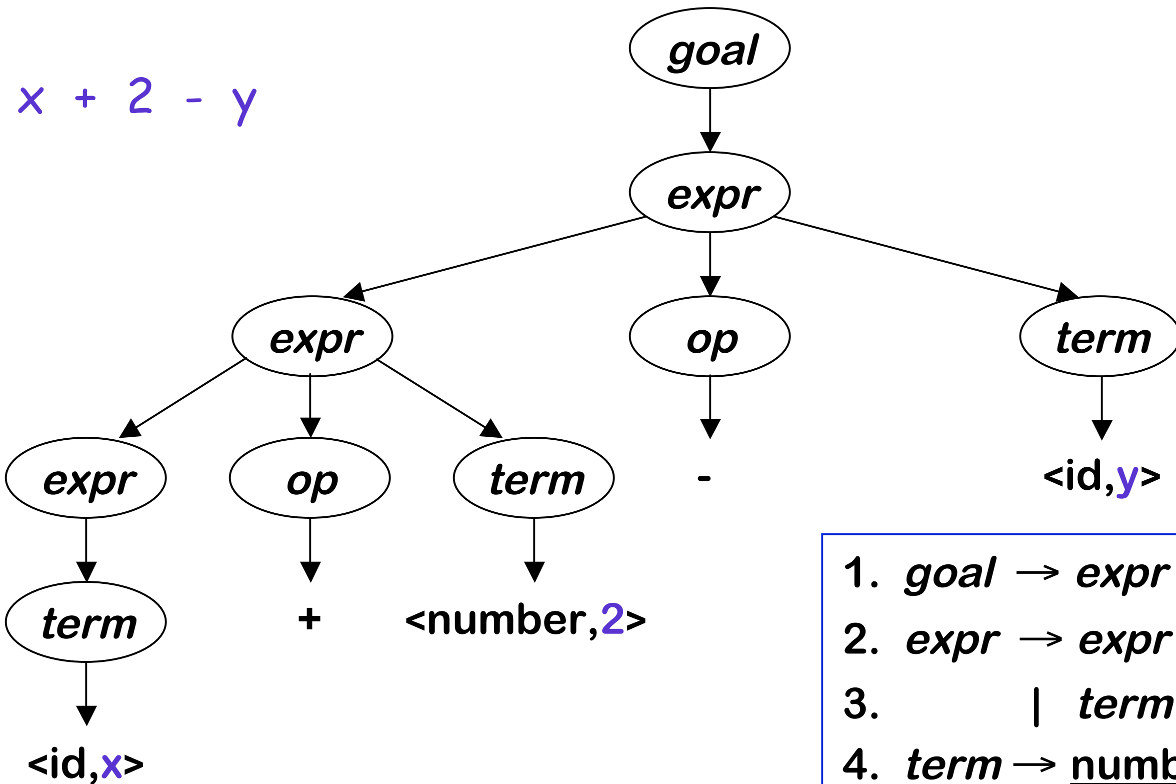
1.	*goal*	→	*expr*
2.	*expr*	→	*expr op term*
3.		|	*term*
4.	*term*	→	number
5.		|	id
6.	*op*	→	+
7.		|	-

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

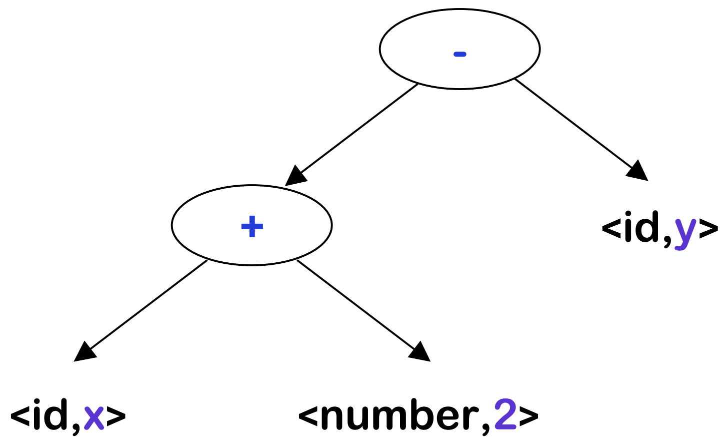


1. *goal* → *expr*
2. *expr* → *expr op term*
3. | *term*
4. *term* → number
5. | id
6. *op* → +
7. | -

This contains a lot of unneeded information.

The Front End

Compilers often use an *abstract syntax tree*

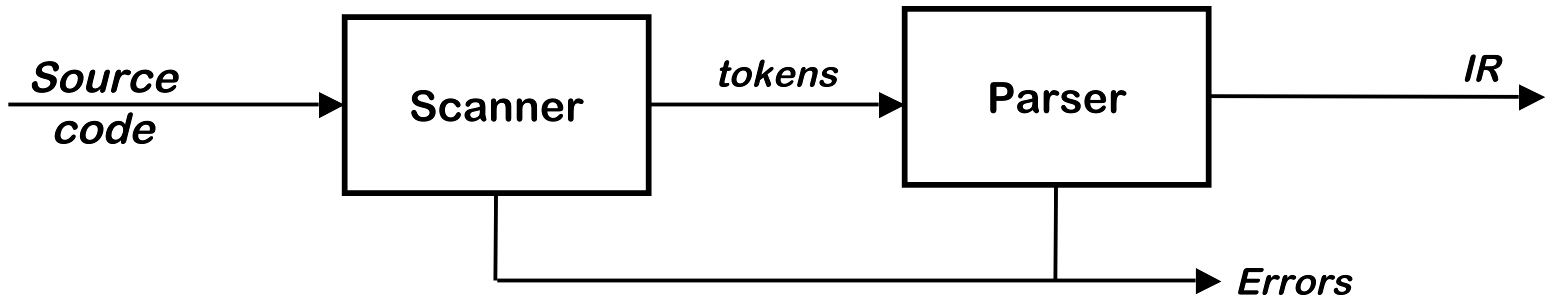


The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

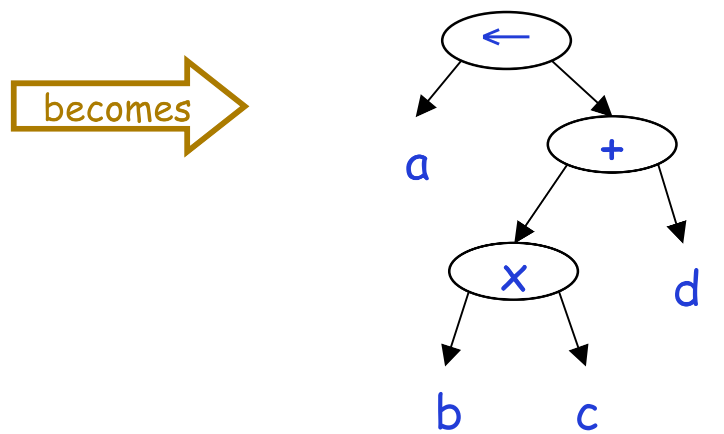
ASTs are one kind of *intermediate representation (IR)*

The Front End

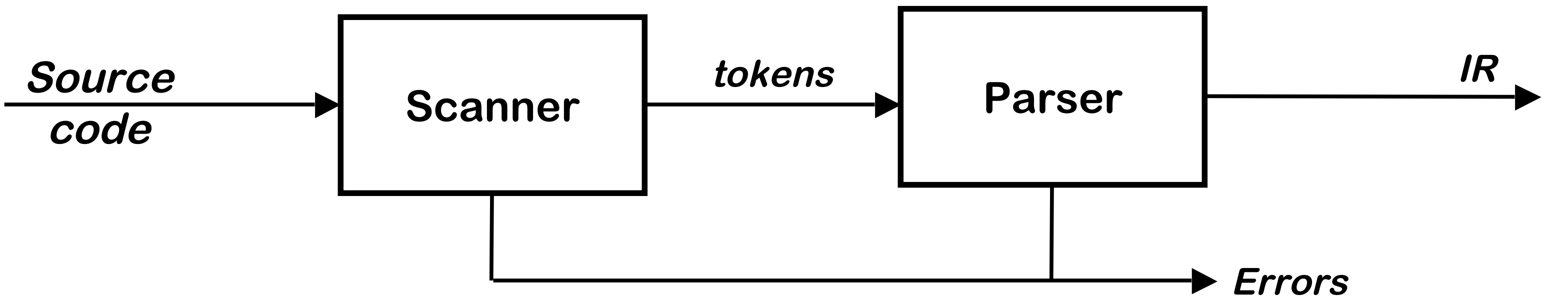


Code shape determines many properties of resulting program

a ← b x c + d



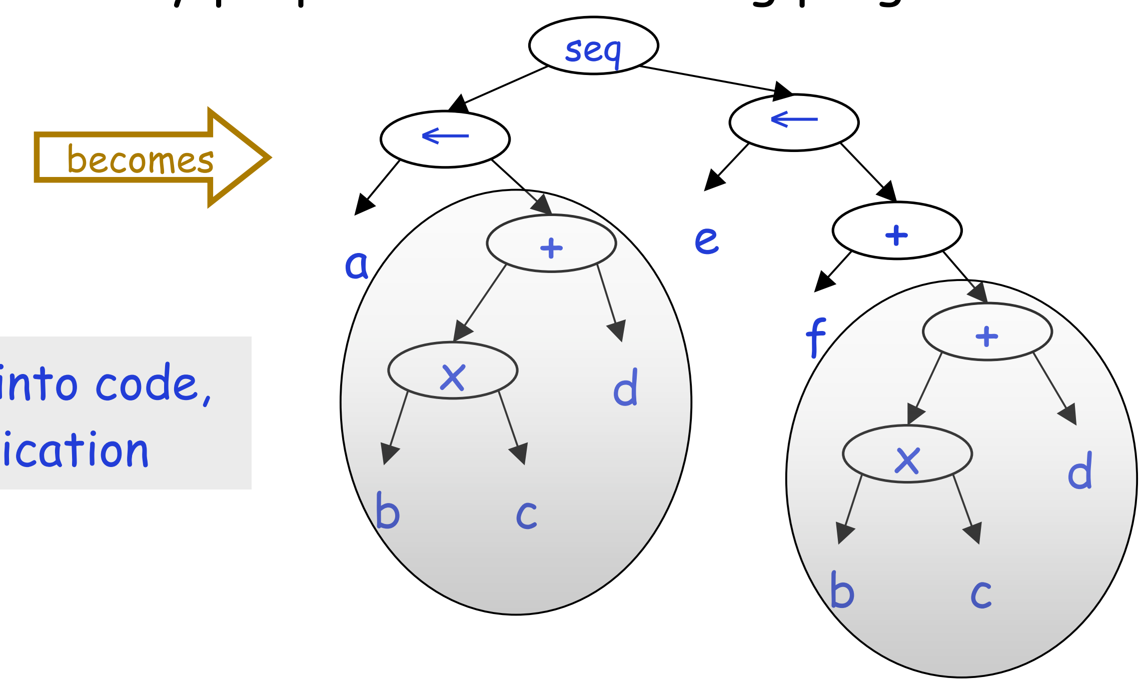
The Front End



Code shape determines many properties of resulting program

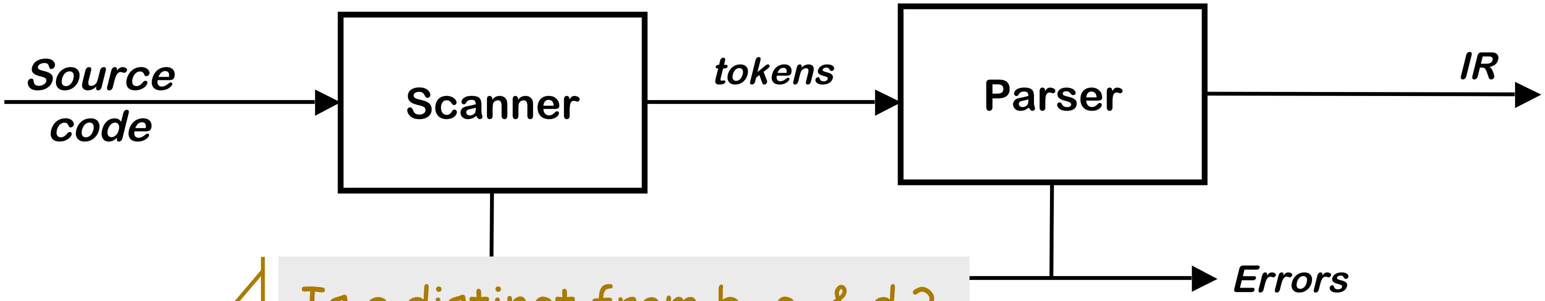
$a \leftarrow b \times c + d$

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



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

$a \leftarrow b \times c + d$

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

becomes

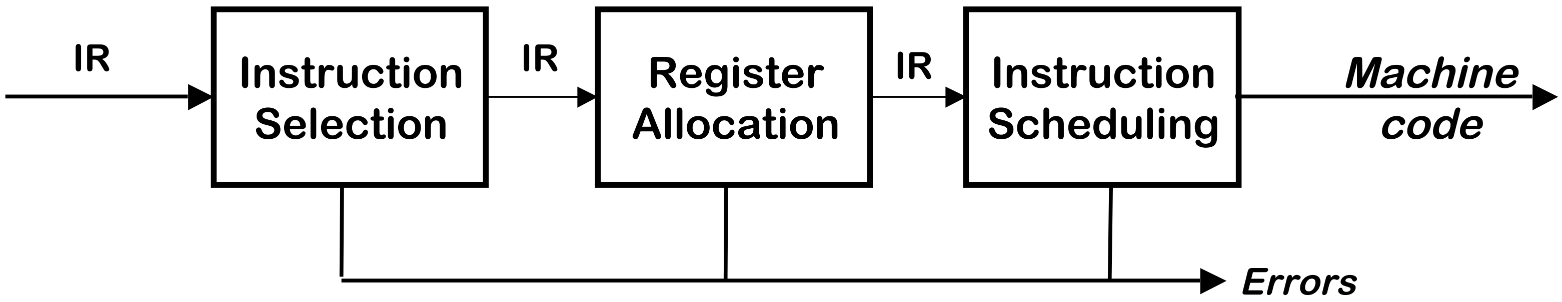
load @b ⇒ r_1
load @c ⇒ r_2
mult r_1, r_2 ⇒ r_3
load @d ⇒ r_4
add r_3, r_4 ⇒ r_5
store r_5 ⇒ @a
load @f ⇒ r_6
add r_5, r_6 ⇒ r_7
store r_7 ⇒ @e

computes $b \times c + d$ (first five instructions)

reuses $b \times c + d$ (load @f and add r_5, r_6)

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

The Back End

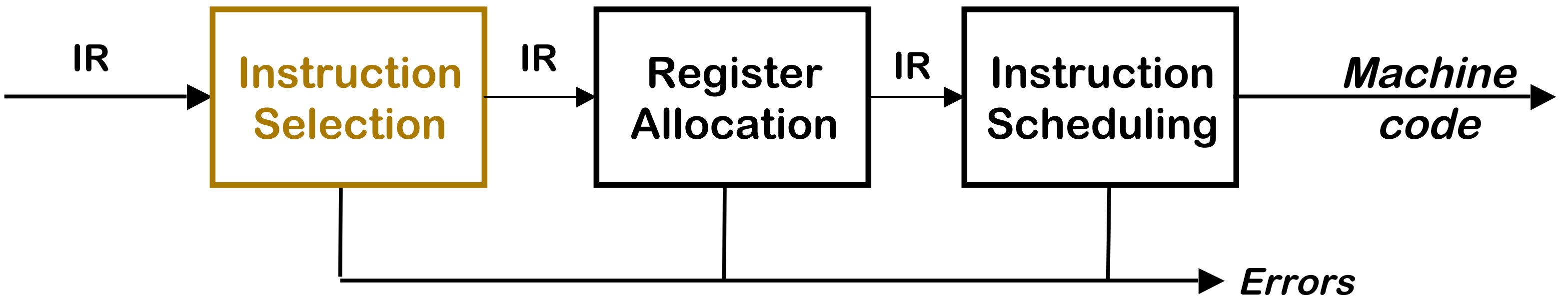


Responsibilities

- 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



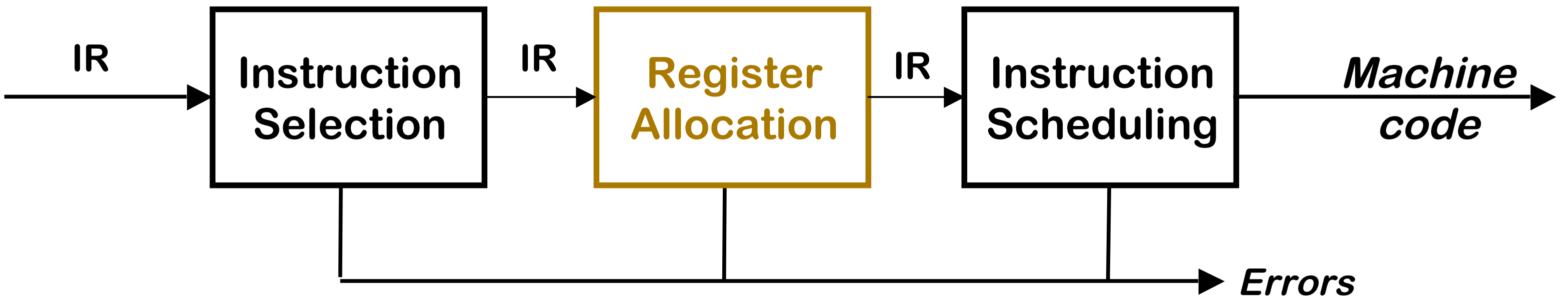
Instruction Selection

- 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

This was the problem of the future in 1978

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

The Back End



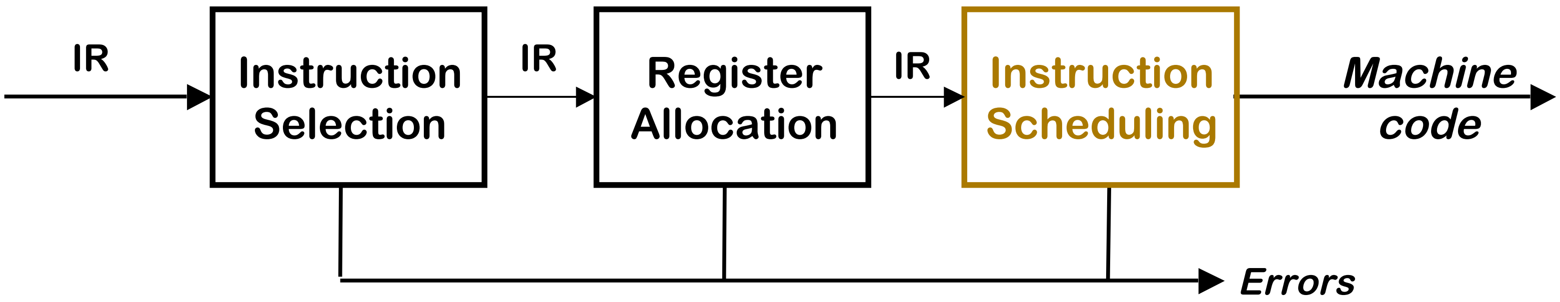
Register Allocation

- 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 (1 or *k* registers)

Compilers approximate solutions to NP-Complete problems

You will become experts over next 3 weeks…

The Back End



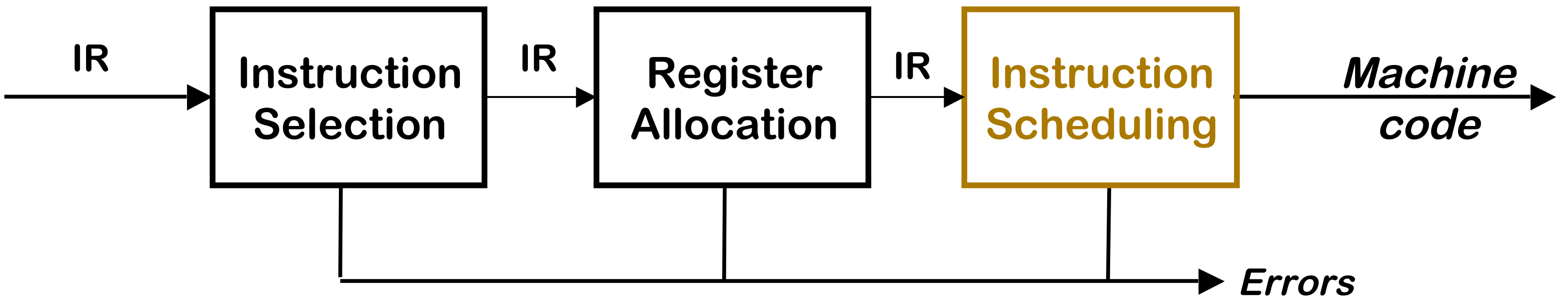
Instruction Scheduling

- 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

The Back End



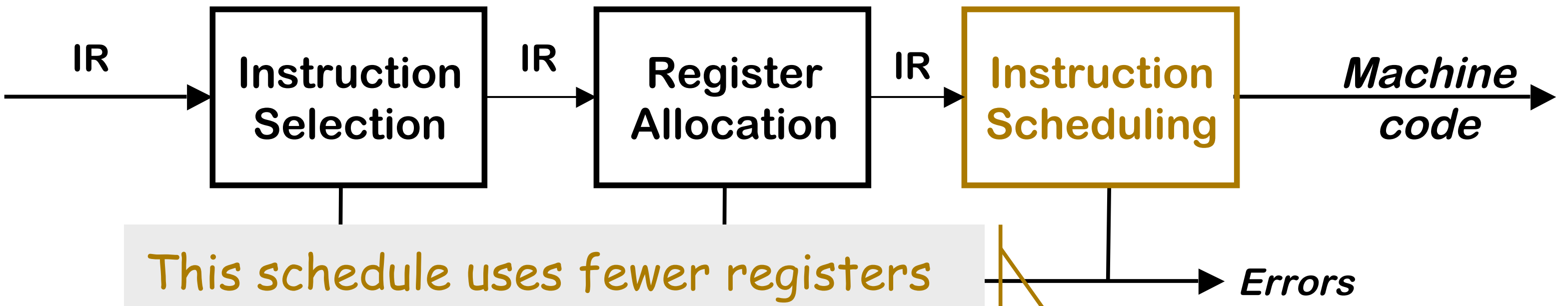
Instruction Scheduling

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



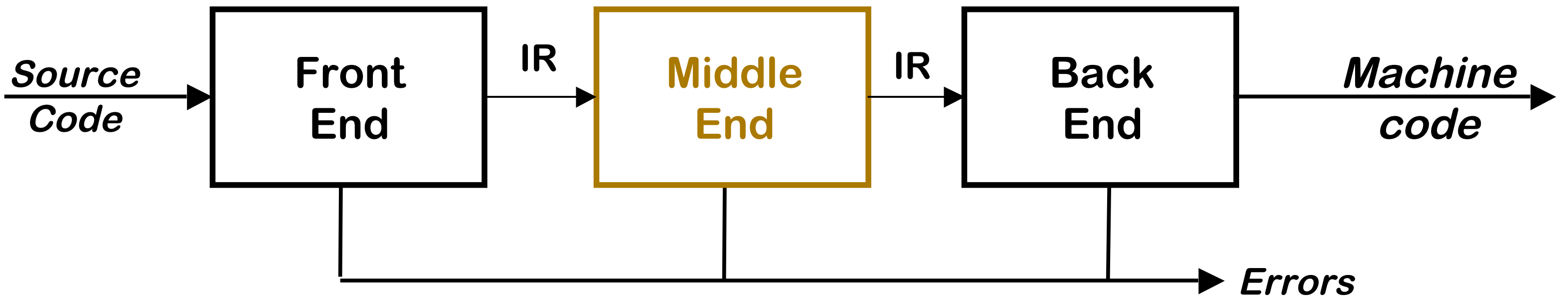
This schedule uses fewer registers without slowing down the code

Instr … load)

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

Traditional Three-pass Compiler

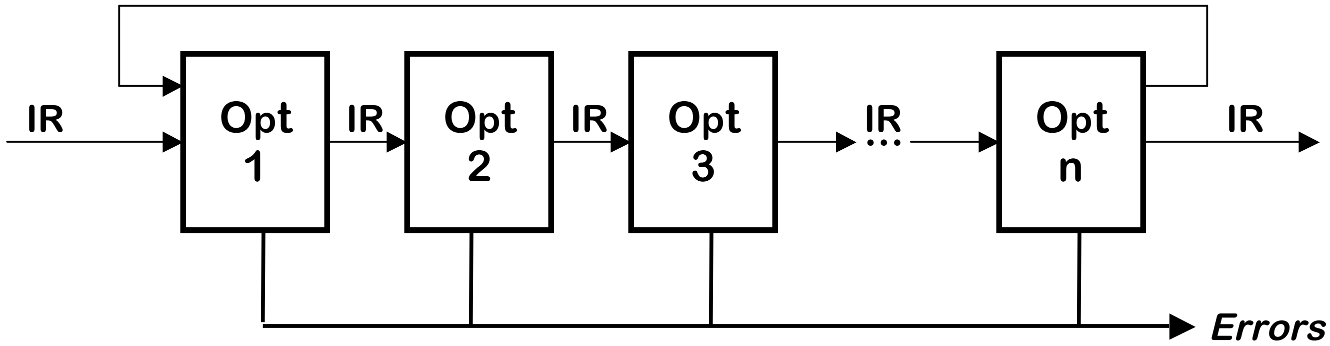


Code Improvement (or *Optimization*)

- 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 1 to get addr(A(I,J))
     A(I,J) = A(I,J) + C
ENDDO
```

Example assumes column-major order; an equivalent issue arises with row major order

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

Next Class

- Introduction to Local Register Allocation
- Announcements:
  - Specs for Lab 1 available by Monday, August 26
    - Due Sept 15 (documentation 1 day later)
    - Practice blocks and simulator will be available
    - Grading blocks will be hidden from you