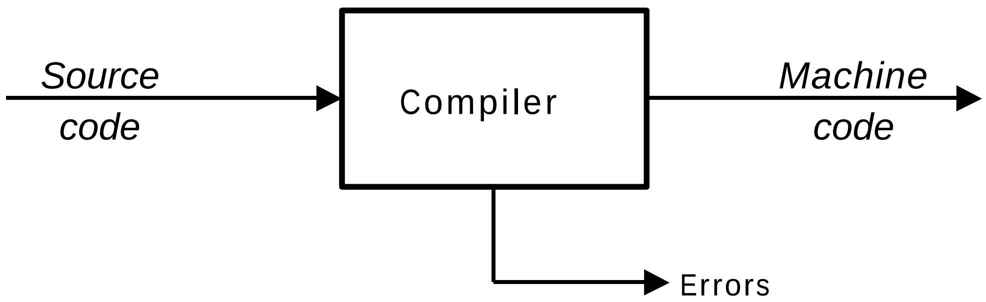
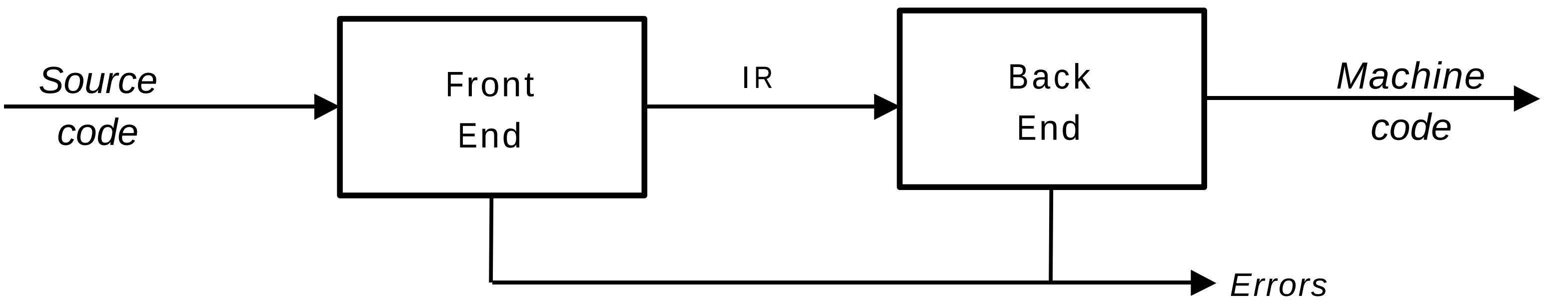
High-level View of a Compiler



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

Traditional Two-pass Compiler

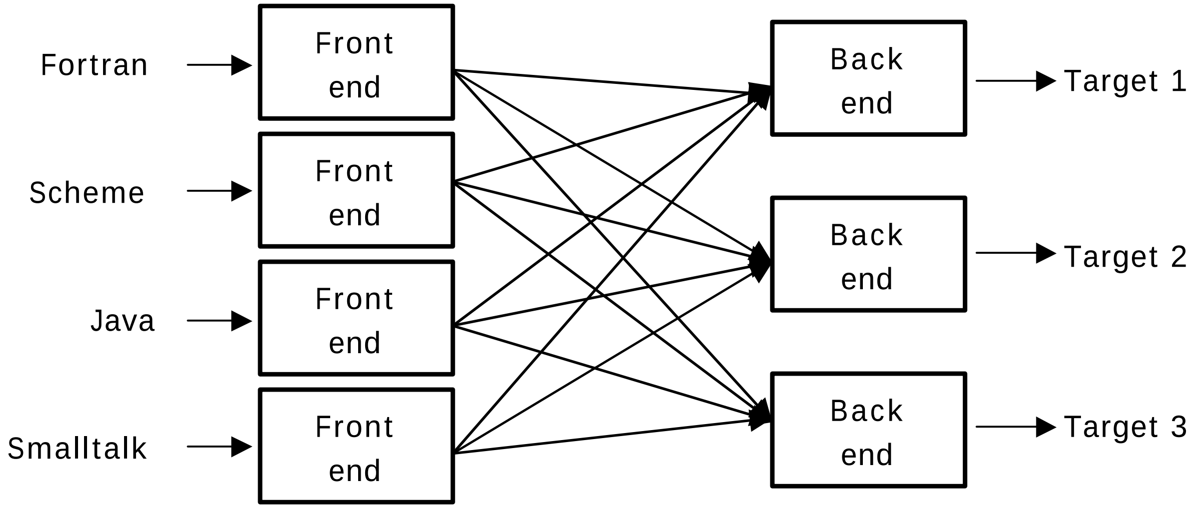


Implications

- 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

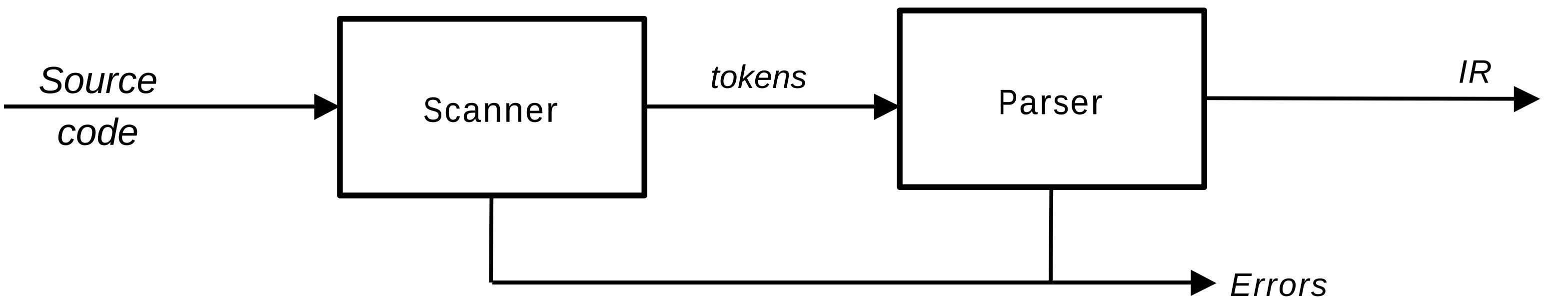


Can we build *n x m* compilers with *n+m* components?

- 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

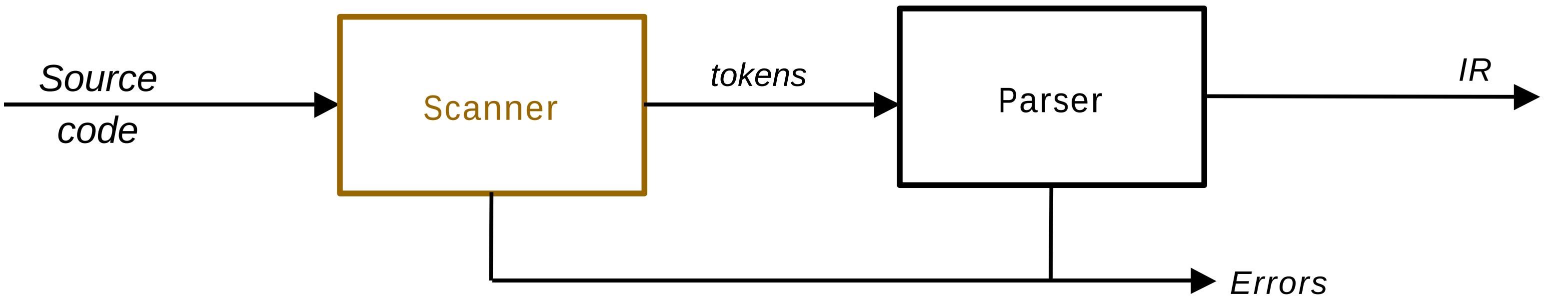
The Front End



Responsibilities

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

The Front End



Scanner

- Maps character stream into words—the basic unit of syntax
- Produces words & their parts of speech

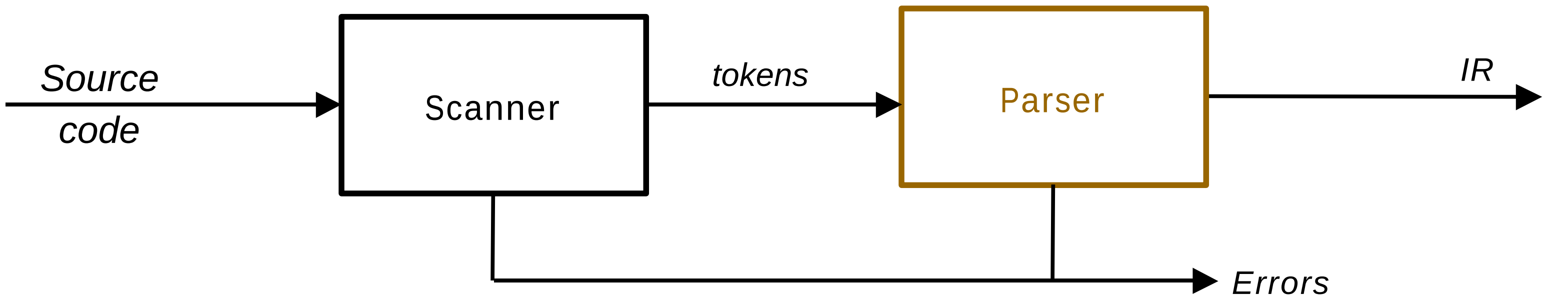
  x = x + y ; *becomes* <id,x> <op,= > <id,x> <op,+ <id,y> ;

  > *word @ lexeme, part of speech @ token*

  > In casual speech, we call the pair a *token*

- Typical tokens include number, identifier, +, -, while, if
- Scanner eliminates white space
- Speed is important ⇒use a specialized recognizer

The Front End



Parser

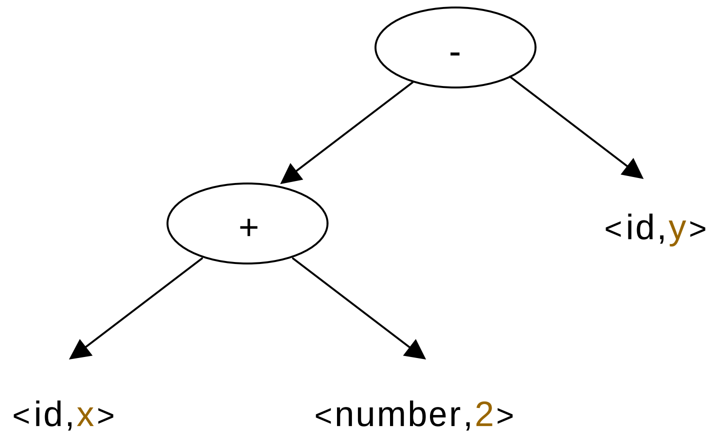
- Recognizes context-free syntax & reports errors
- Guides context-sensitive analysis (*type checking*)
- Builds IR for source program

Hand-coded parsers are fairly easy to build

Most books advocate using automatic parser generators

The Front End

Compilers often use an *abstract syntax tree*

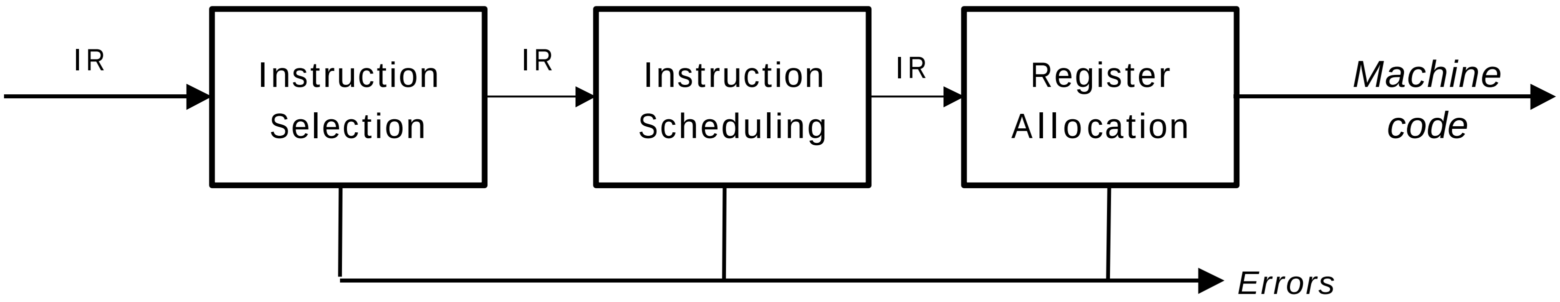


The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

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

The Back End

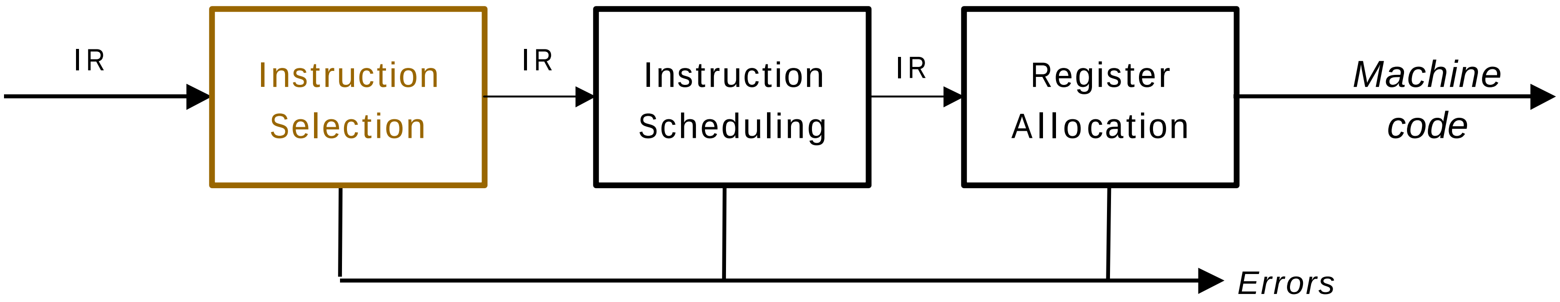


Responsibilities

- 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 *much less* successful in the back end

The Back End



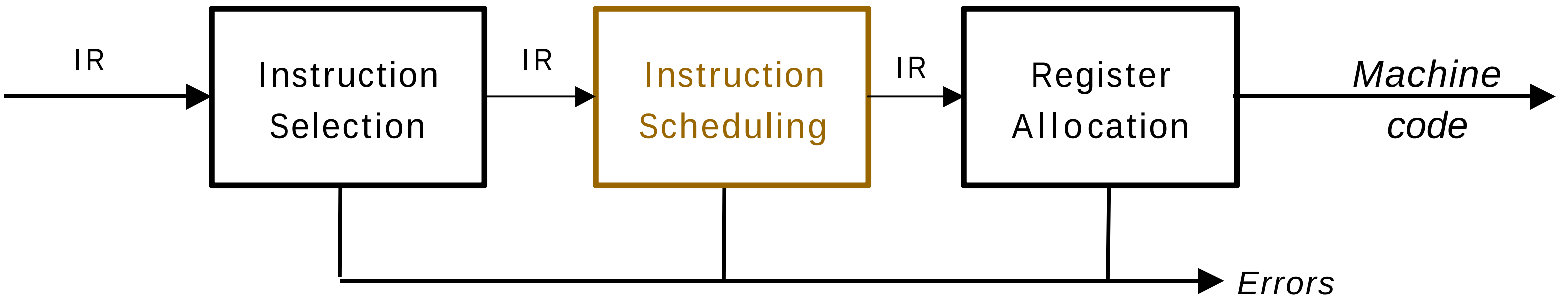
Instruction Selection

- 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

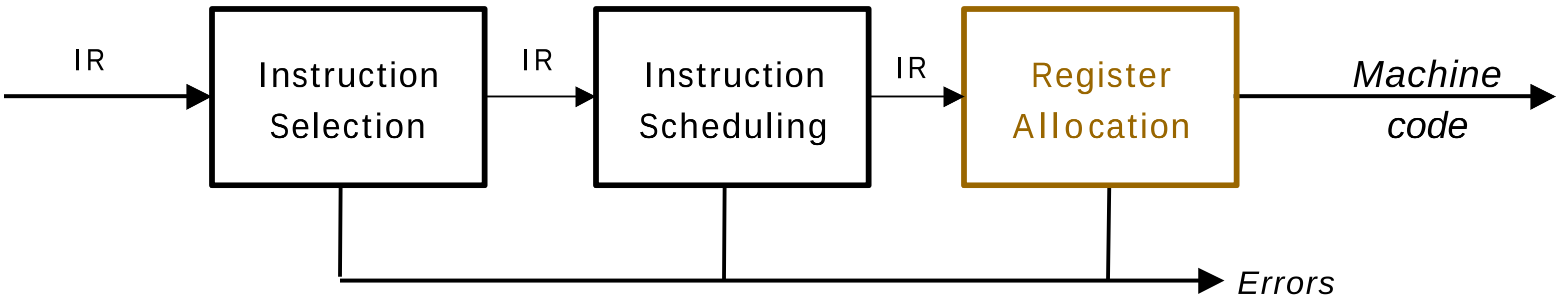


Instruction Scheduling

- 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

Good heuristic techniques are well understood

The Back End

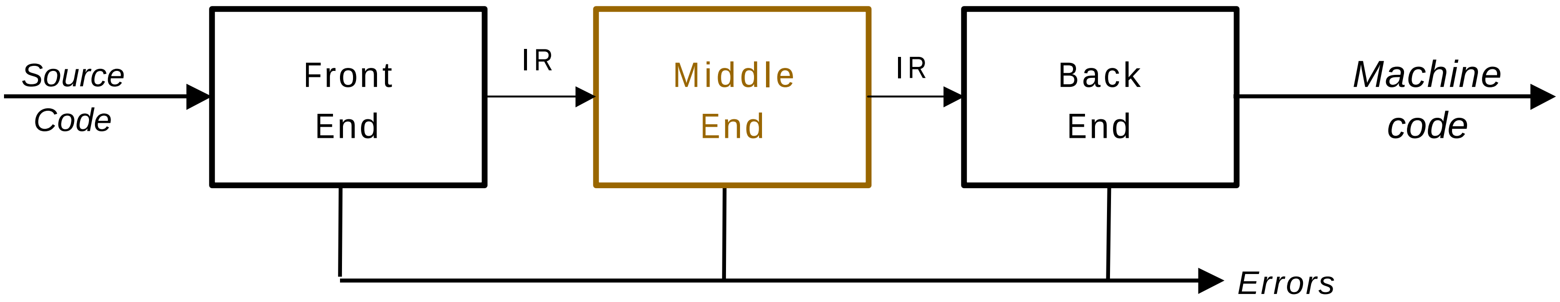


Register allocation

- 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

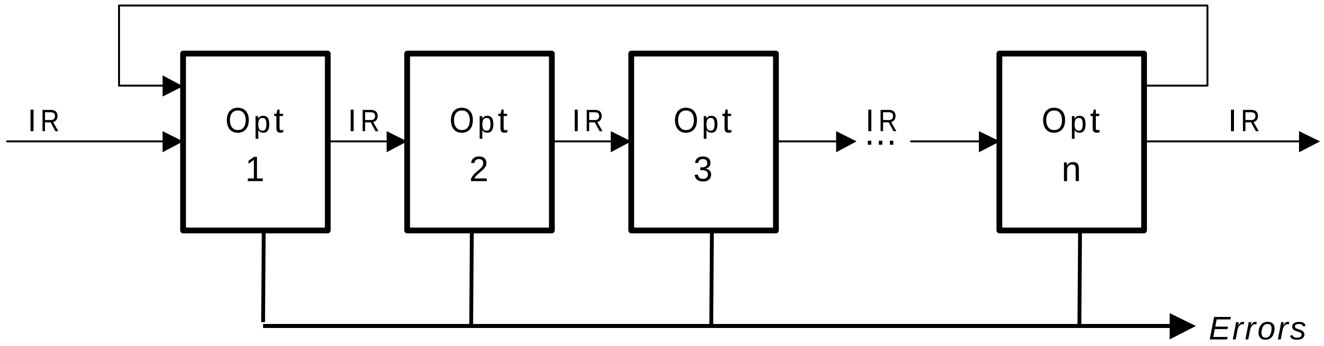
Traditional Three-pass Compiler



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

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
- Discover a redundant computation & remove it
- Remove useless or unreachable code