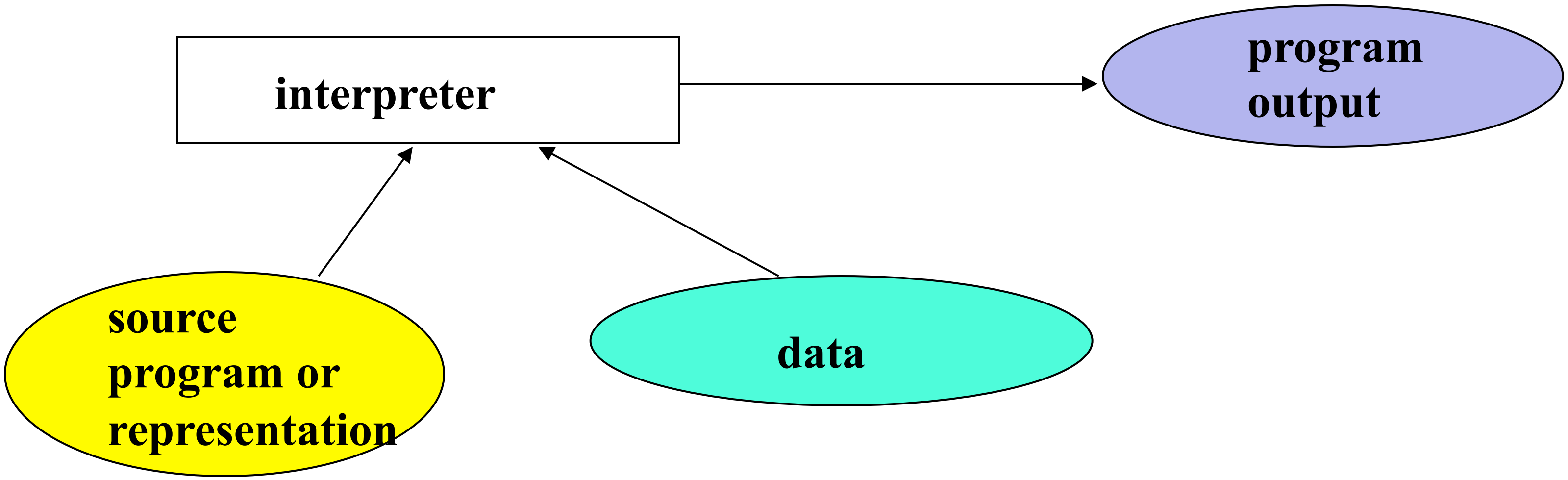
Overview Roadmap

- Language Translators: Interpreters & Compilers
- Context of a compiler
- Phases of a compiler
- Compiler Construction tools
- Terminology
- How related to other CS
- Goals of a good compiler

Compilers and Interpreters

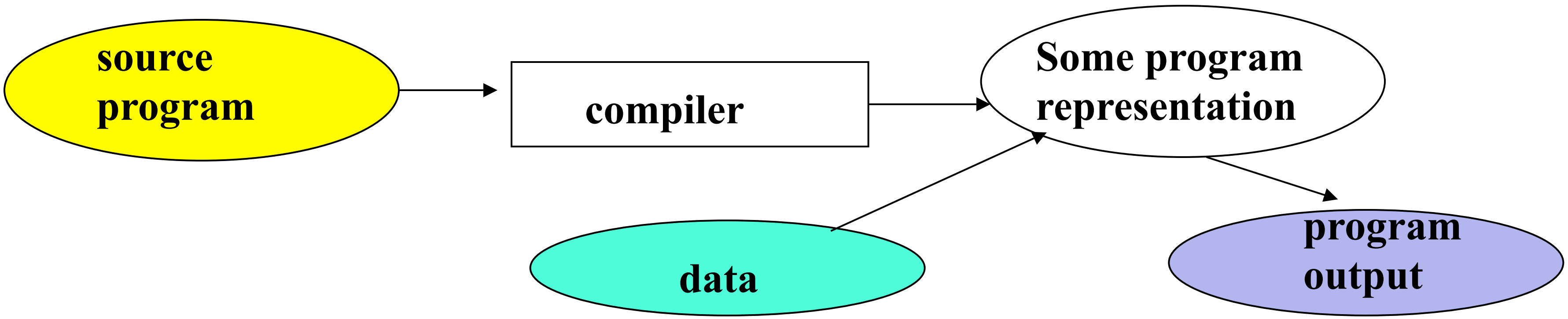
- **What is a compiler?**
- **What is an interpreter?**
- **Implementation of Languages**

Overview of interpreters and compilers



Locus of control - in interpreter, not program

Compiler has distinct translation and execution phase



Interpreters

Advantages

Disadvantages

Compilers

Advantages

Disadvantages

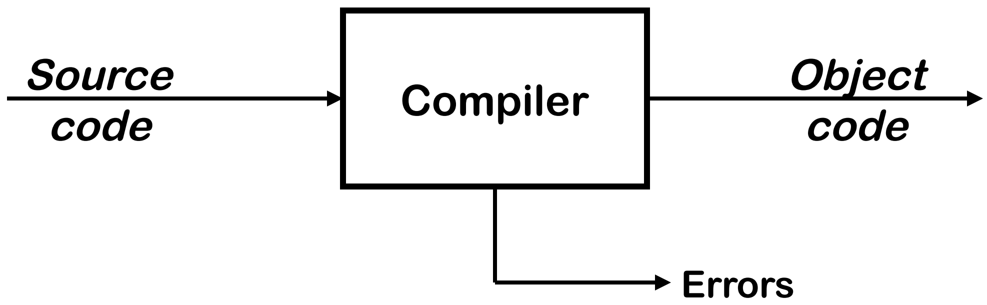
Relation to Other CS

Artificial intelligence	Greedy algorithms, Genetic algorithms Heuristic search techniques
Algorithms	Graph algorithms, union-find Dynamic programming
Theory	DFAs & PDAs, pattern matching Fixed-point algorithms
Systems	Allocation & naming, Synchronization, locality
Architecture	Memory hierarchy management Functional units & pipelines Instruction set use

From Your Experience

- You have used several compilers.
- What qualities do you want in a compiler that you buy ?

High-level View of a Compiler



Must recognize legal (and illegal) programs

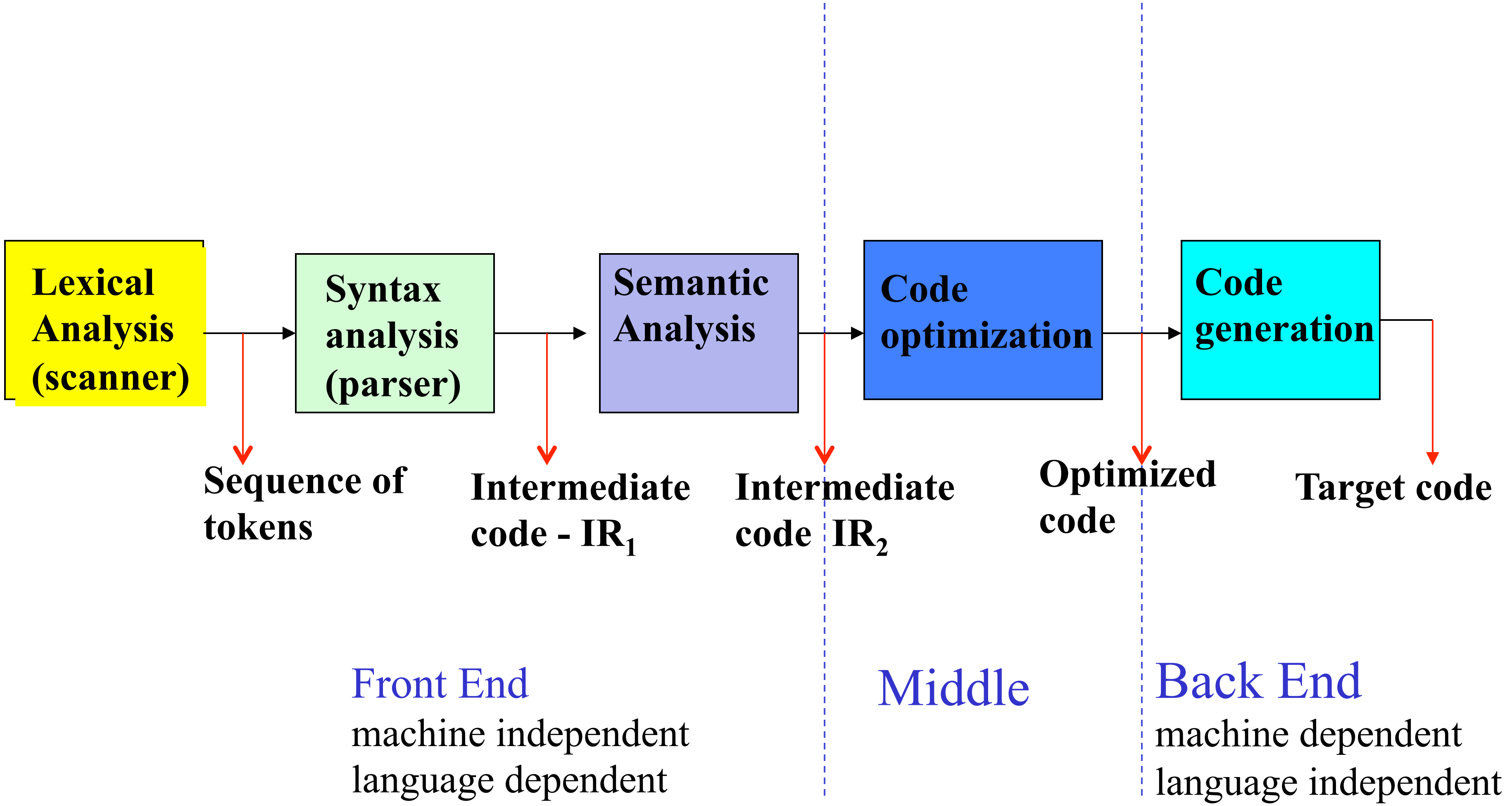
Must generate correct code

Must manage storage of all variables (and code)

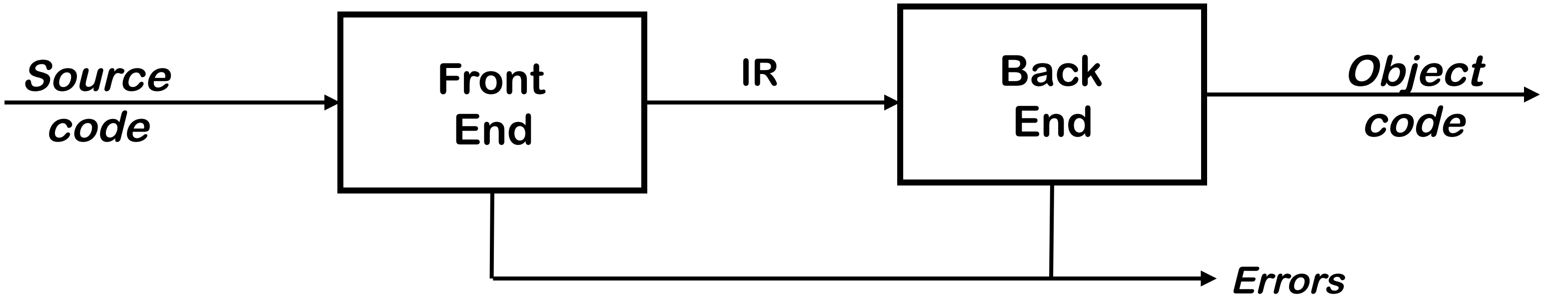
Must agree with OS & linker on format for object code

Big step up from assembly language—use higher level notations

Conceptual phases of compiler



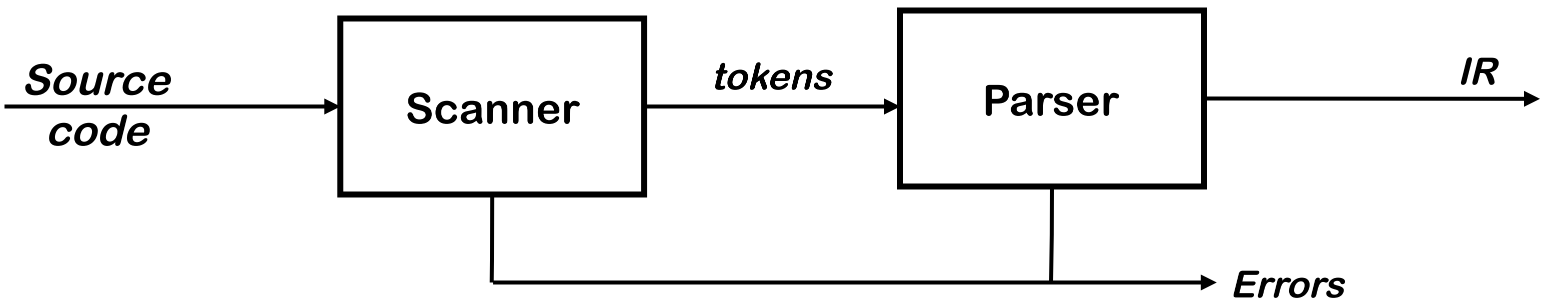
Traditional Two-pass Compiler



Allow 2 passes:

- 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*)

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

Lexical Analysis/Scanner

Purpose: recognize words - smallest unit

Analyze string of characters from source - left to right to recognize units

Character string - lexeme

Type of lexical entity - token

Smallest unit above letters

Example:

Max:= initial * late + 60

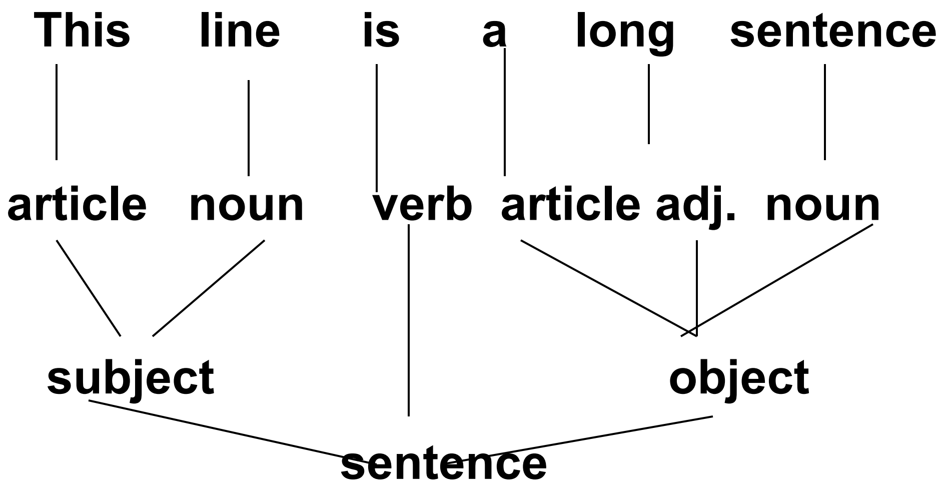
Lexemes: “max”, “:=“, “initial”, “*”, “late”, “+”, “60”

Tokens: Id Id Id := * + Int

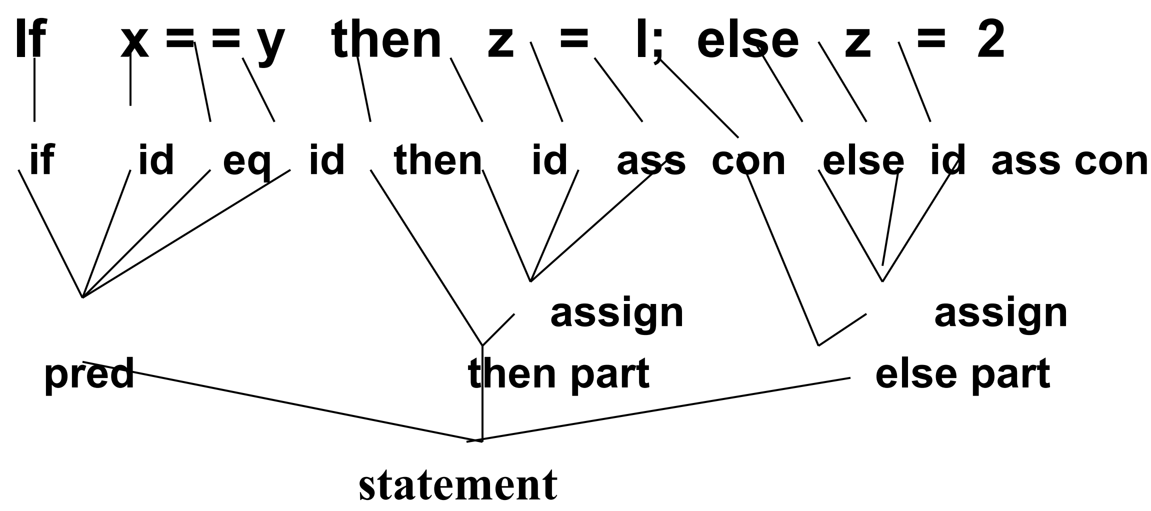
Must recognize blanks, other characters such as % , $, etc

Syntax Analyzer - Parser

Parsing similar to diagramming a natural language sentence



Parsing



Semantic Analysis

Once structure is understood, determine the meaning using the structure.

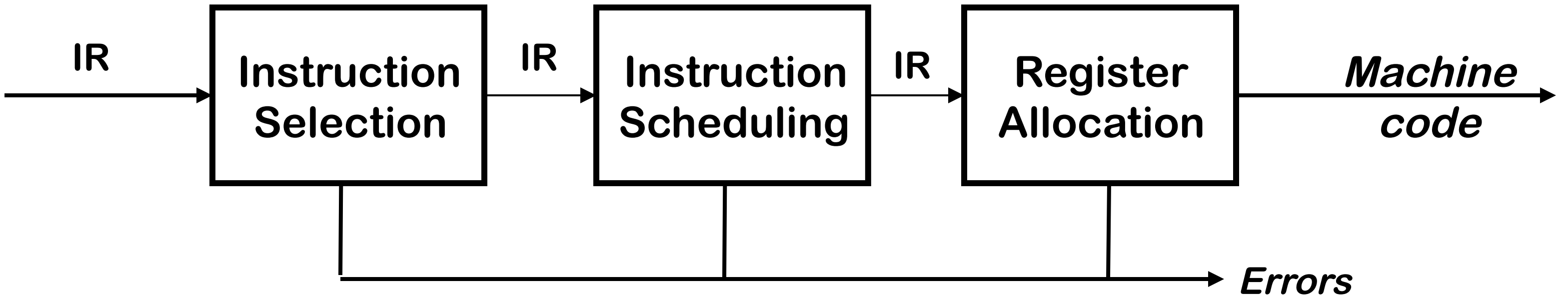
Checks performed to ensure components fit together meaningfully

- information is added to structures
- limited analysis to catch inconsistencies - e.g., type checking

Put semantic meaning in structure -

- produce intermediate form - IR - many forms of IR
- easier to generate machine code from IR
- can be different levels of IR - descending levels of abstraction
    - Highest is source
    - Lowest is target code

The Back End/Code Generation

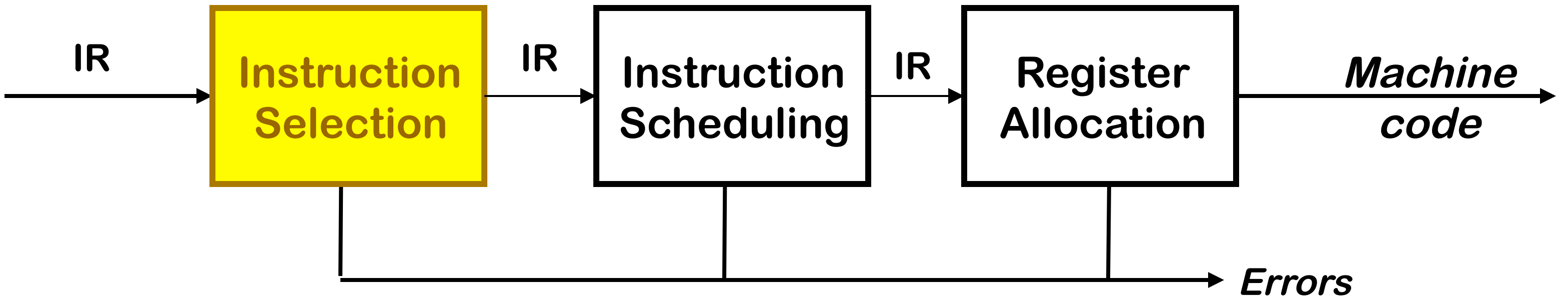


Responsibilities

- 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

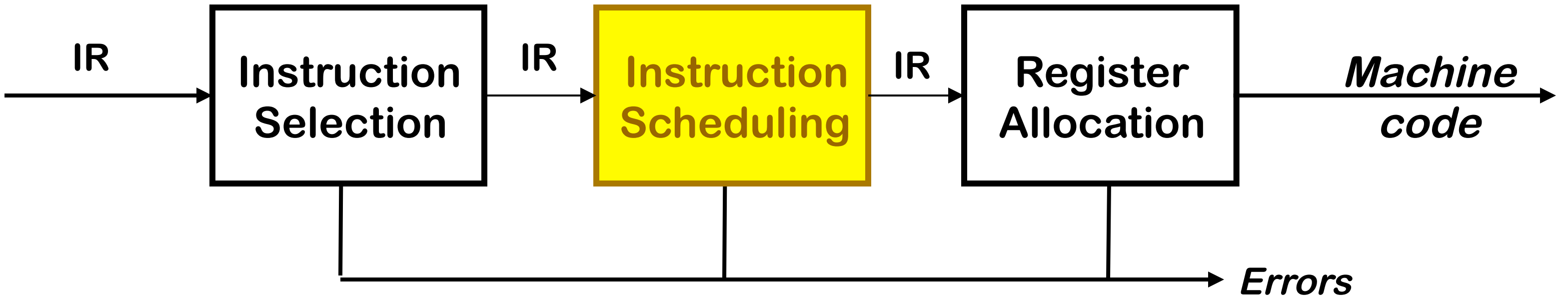
The Back End



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
  - Depends on architecture - CISC, RISC

The Back End

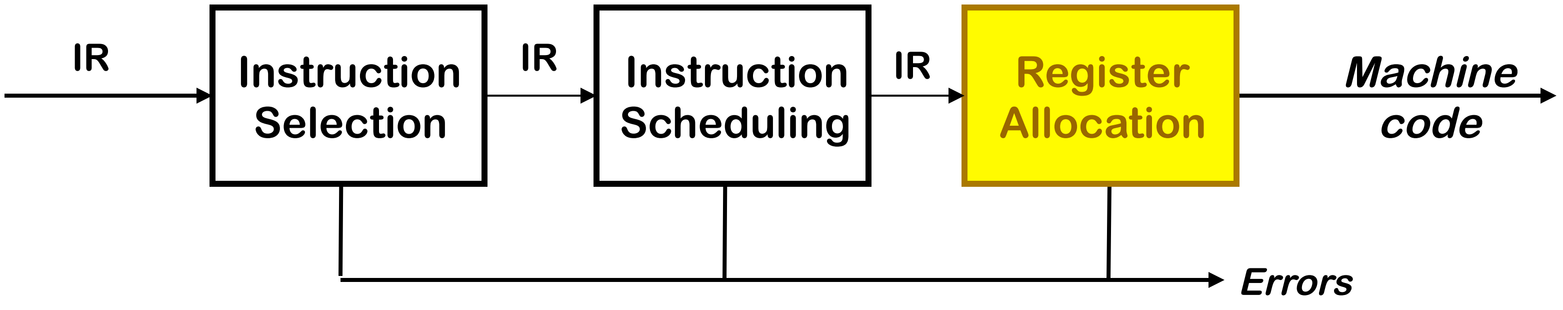


Instruction Scheduling

- 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
- Compilers approximate solutions to NP-Complete problems

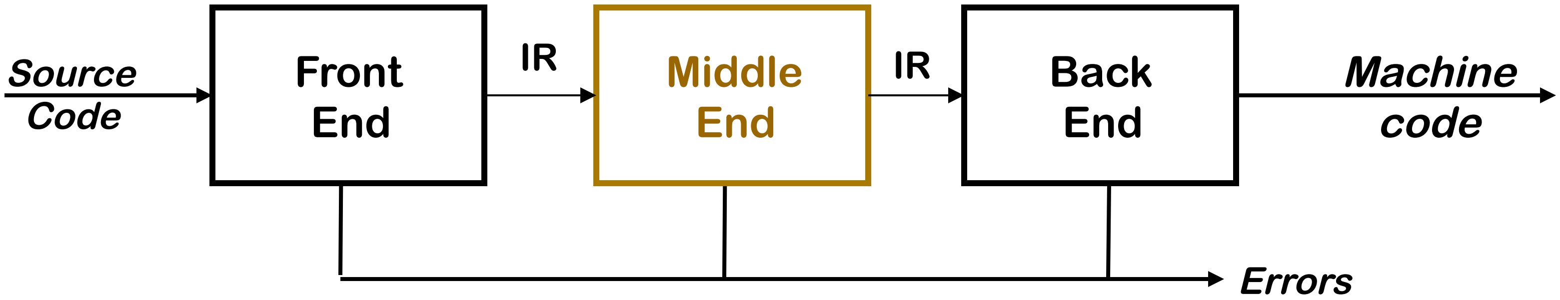
Code Generation – what kind of code

Produce target code - various forms of target code

1. **Assembly Code - symbolic instruction and addresses**
   - **Easier but not done in modern compilers - assembler slow**
2. **Relocatable format –**
   - **Binary form except external references, instruction addresses and data addresses not bound to address**
   - **Need linker and loader**

   Both assembly & relocatable allow program modules to be separately compiled
3. **Another language**

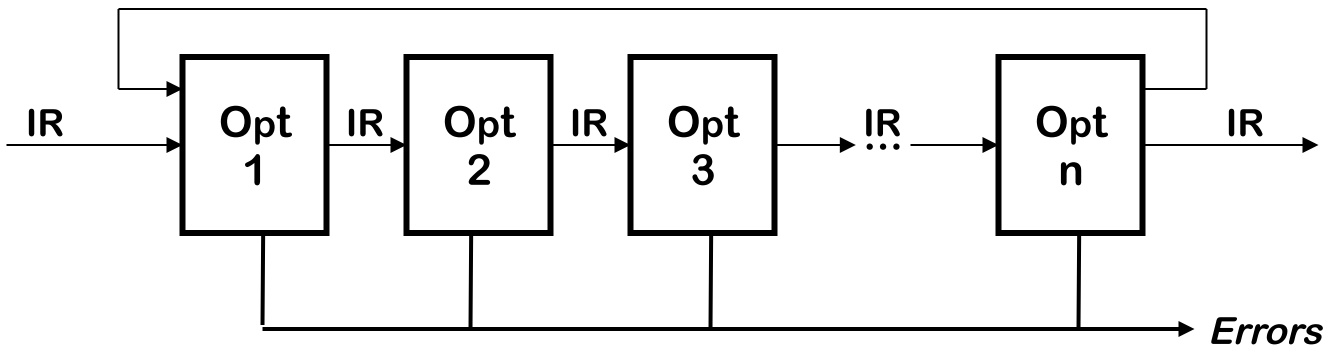
Traditional Three-pass Compiler



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
- 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

Code Optimization

Modify program representation so that the program

- runs faster
- uses less memory
- uses less power
- in general, reduce the resources consumed

e.g., constant propagation and folding

Y:= 3

X:= Y + 4

optimizes to X:= 7

Symbol Table Manager

Collect and maintain information about id's

- **attributes e.g., storage allocation, type, scope, number and type of parameters**

Usually cuts across all phases - lexical, parsing and semantic, code optimization, code generation

- **Phase add information - lexical, parsing and semantic**
- **Phases use information - code optimization, code generation**

Debuggers uses some form of symbol table

Error Reporting

Phases deal with errors - 1st 3 phases handled bulk of errors

Lots of success here

Distinction between phases and passes

Passes - number of times through a program representation

- **1 - passes, 2 - passes, multiple passes**
- **Languages become more complex - more passes**

Phases - conceptual and sometimes physical stages

- **Symbol table coordinating information between phases**

However, phases are not completely separate - semantic phase must do things that syntax phase should do if it could

Some interaction possible:

- **optimization and code generation - what optimizer does affects code generator**

Compiler tools

Scanner generator

- **Generate lexical analyzer from specification of tokens based on regular expressions**
- **Examples: Lex, Flex, JLex**

Parser generator

- **Generate parser from specification of syntactical structure using BNF grammars**
- **Example: YACC, Bison, CUP**

What about compiler generator?

- **How do you specify semantics that is useful for compiler?**
- **How do you specify the architecture?**
- **How do you specify optimizations?**