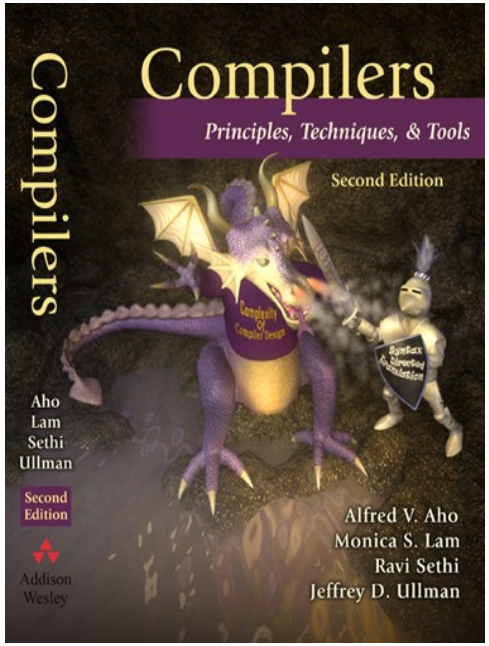
Compiler Design (40-414)

- Main Text Book:
  Compilers: Principles, Techniques & Tools, 2nd ed.,
  Aho, Lam, Sethi, and Ullman, 2007

- Evaluation:
  - Midterm Exam 35%
  - Final Exam 35%
  - Assignments 10%
  - Project 20%



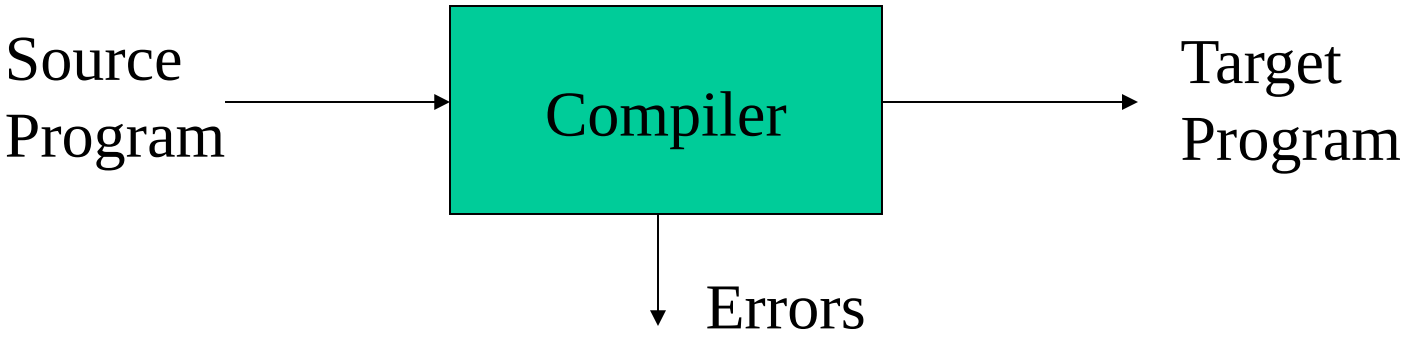
Compiler learning

- Isn't it an old discipline?
  - Yes, it is a well-established discipline
  - Algorithms, methods and techniques were developed in early stages of computer science
  - There are many compilers around, and
  - many tools to generate them automatically
- So, why we need to learn it?
  - Although you may never write a full compiler
  - But the techniques we learn is useful in many tasks like:
    - writing an interpreter for a scripting language,
    - validation checking for forms, and
    - so on

Terminology

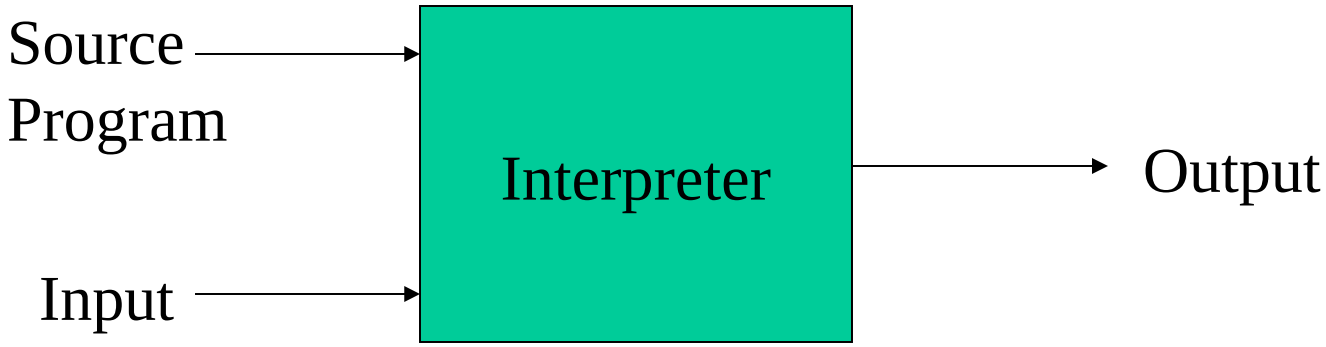
- Compiler:
  - a program that translates an *executable* program in a *source language* (usually high level) into an equivalent *executable* program in a *target language* (usually low level)
- Interpreter:
  - a program that reads an *executable* program and produces the results of running that program
  - usually, this involves executing the source program in some fashion
- Our course is mainly about compilers but many of the same issues arise in interpreters

A Compiler


Source
Program
Compiler
Target
Program
Errors

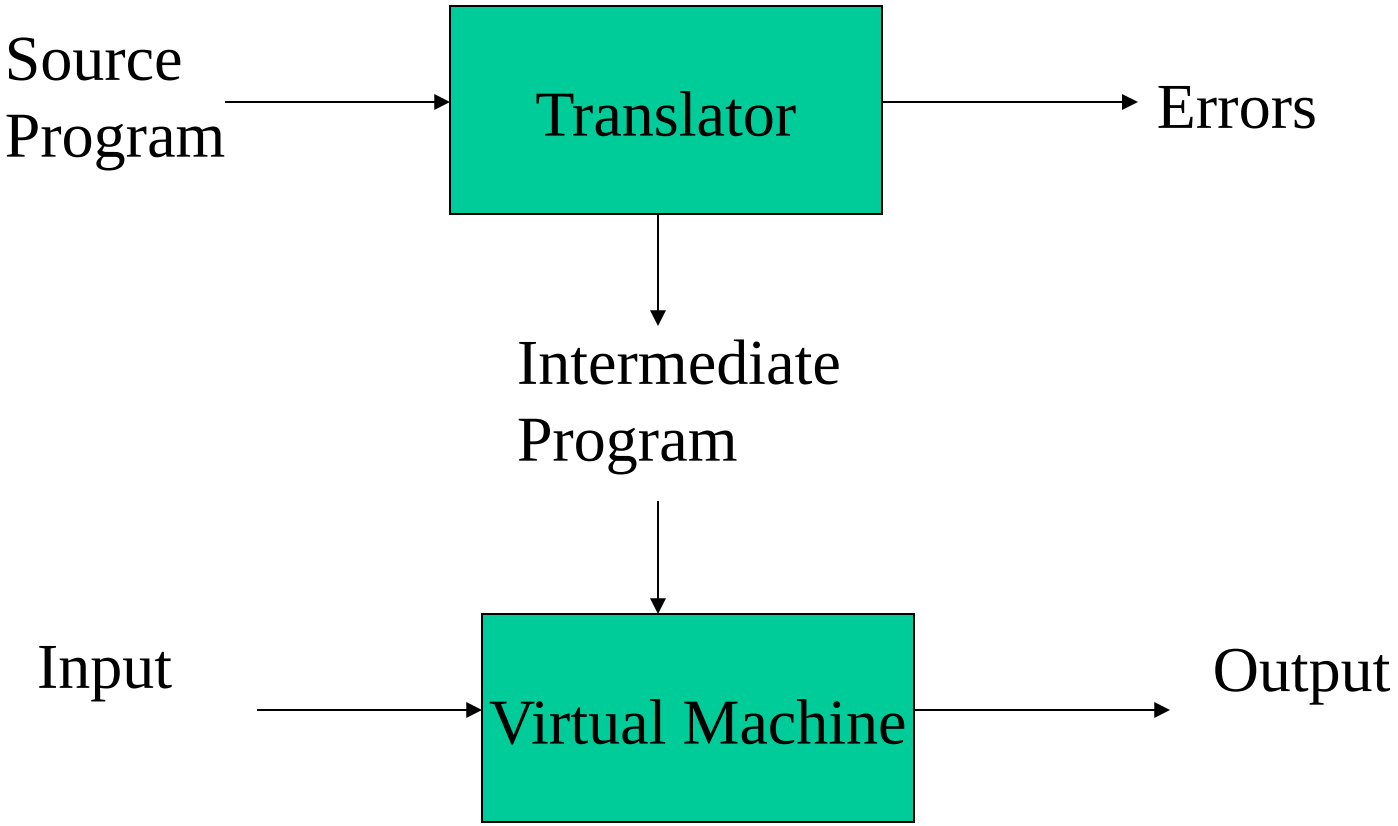

Input
Target Progtam
Output

An Interpreter



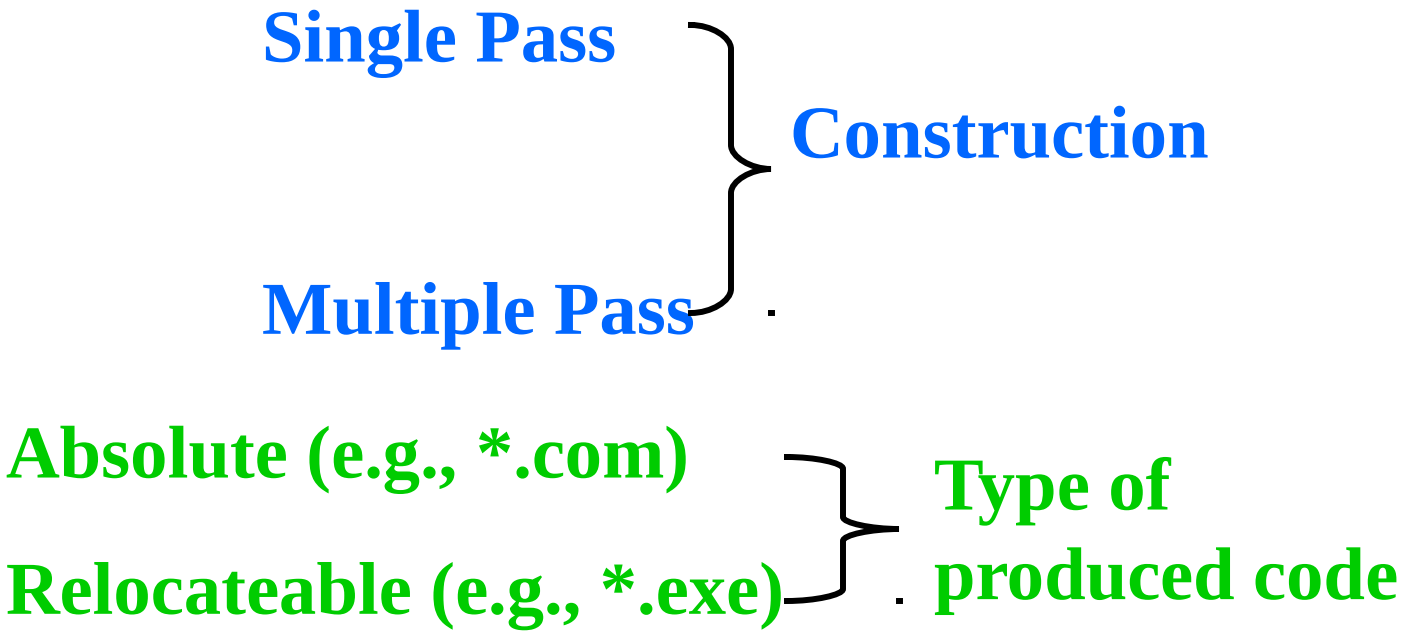
- Translates line by line
- Executes each translated line immediately
- Execution is slower because translation is repeated
- But, usually give better error diagnostics than a compiler

A Hybrid Compiler

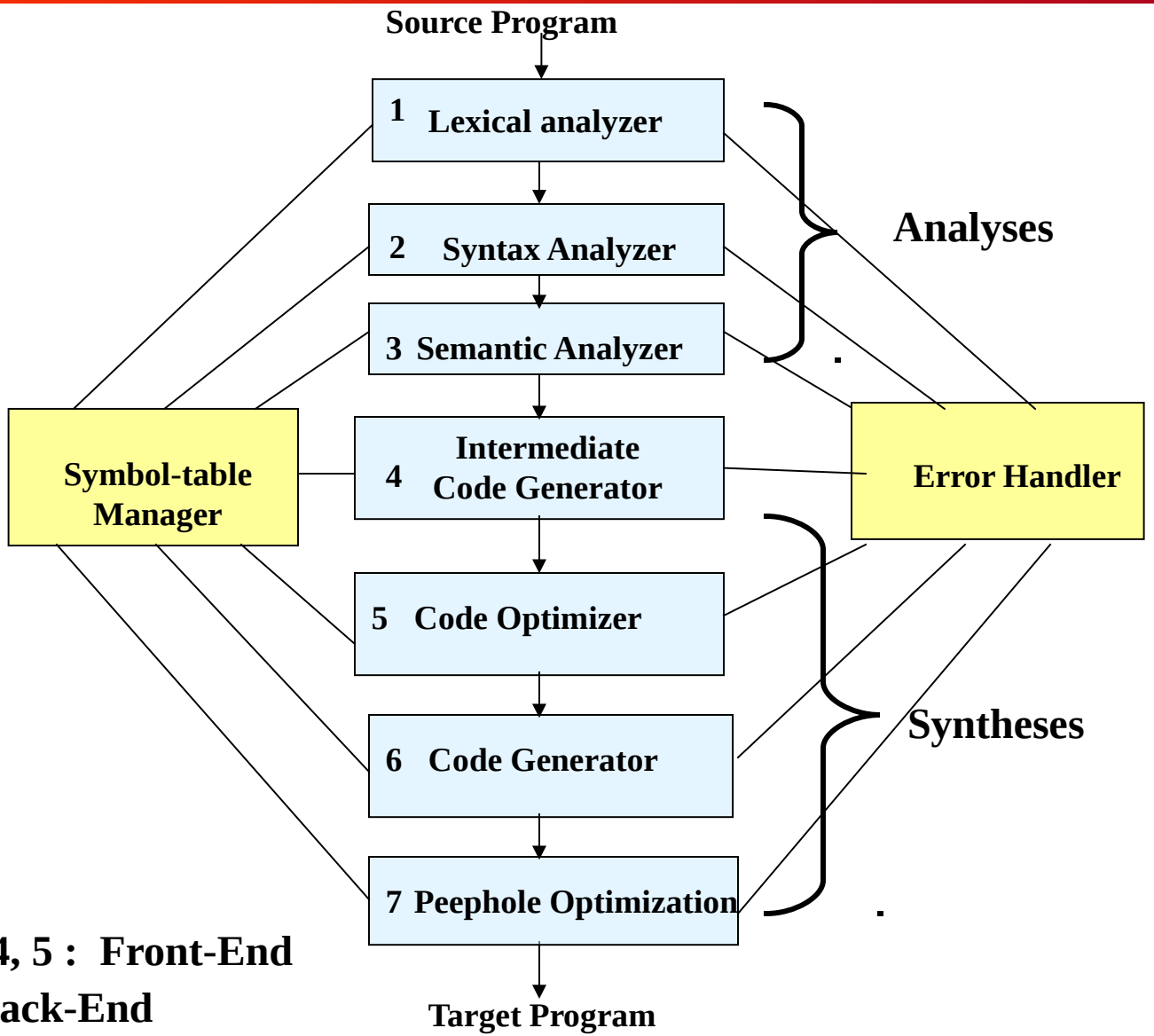


Classifications of Compilers

- There are different types of Compilers:



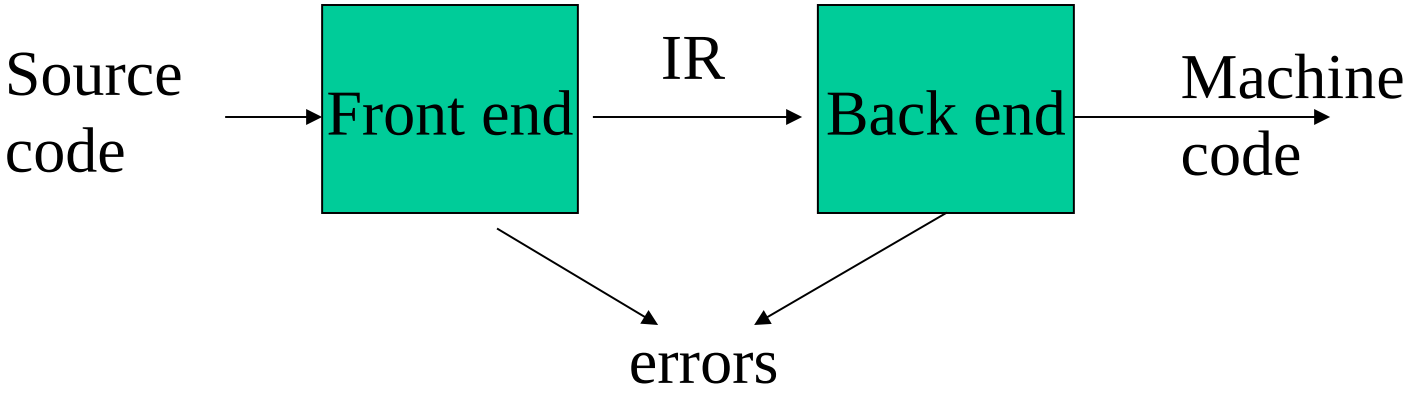
The Many Phases of a Compiler



1, 2, 3, 4, 5 : Front-End

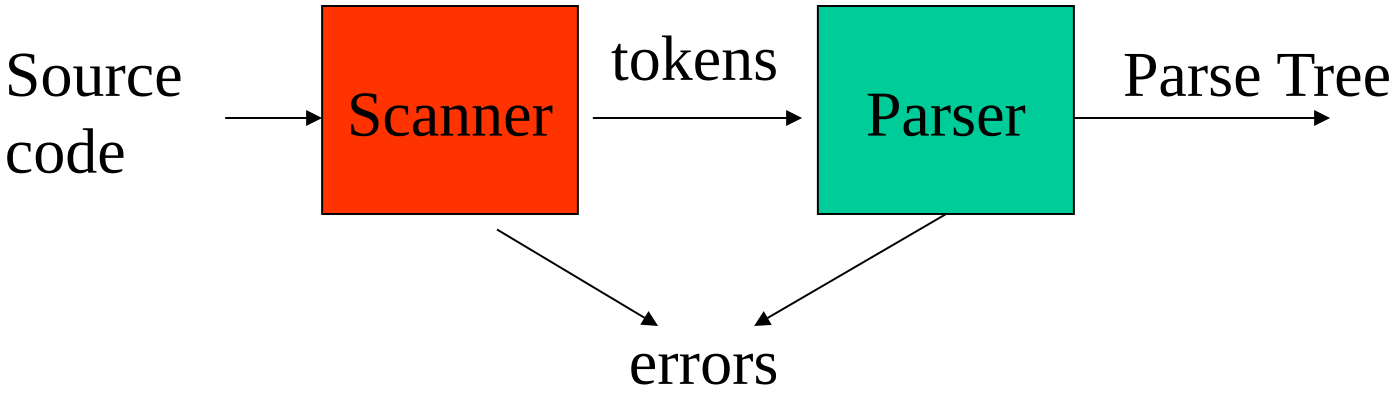
6, 7 : Back-End

Front-end, Back-end division



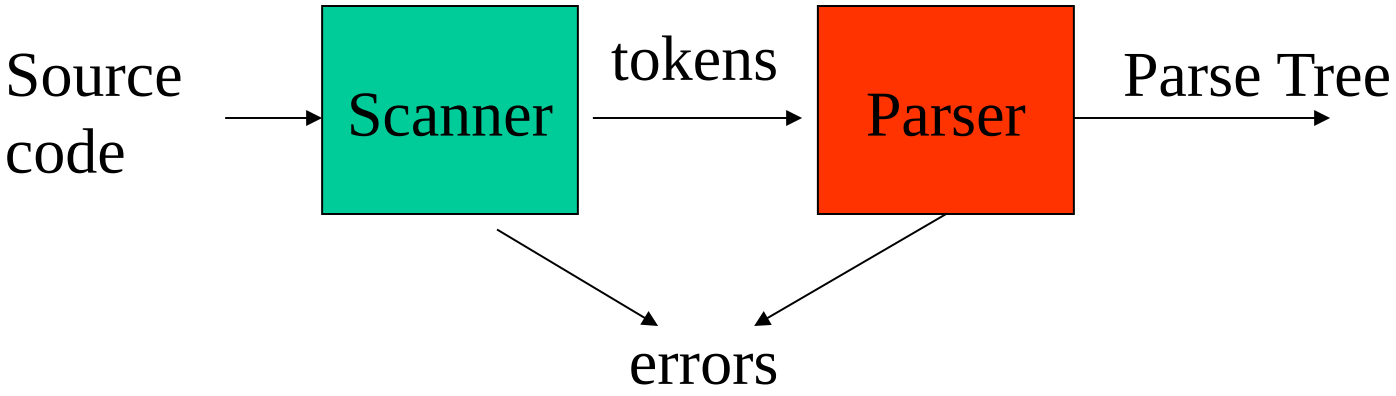
- Front end maps legal code into IR
- Back end maps IR onto target machine
- Simplifies retargeting
- Allows multiple front ends

Front end



- Scanner:
  - Maps characters into tokens – the basic unit of syntax
    - x = x + y becomes <id, x> = <id, x> + <id, y>
  - Typical tokens: number, id, +, -, *, /, do, end
  - Eliminate white space (tabs, blanks, comments)
- A key issue is speed so instead of using a tool like LEX it sometimes needed to write your own scanner

Front end



- Parser:
  - Recognize context-free syntax
  - Guide context-sensitive analysis
  - Construct IR
  - Produce meaningful error messages
  - Attempt error correction
- There are parser generators like YACC which automates much of the work

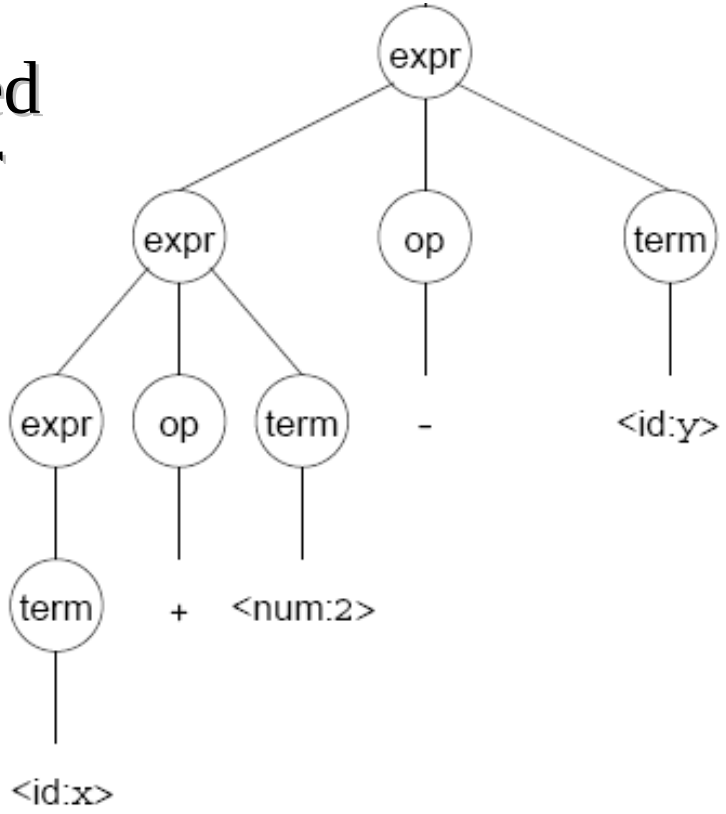
Front end

- Context free grammars are used to represent programming language syntaxes:

```
<expr> ::= <expr> <op> <term> |
    <term>
<term> ::= <number> | <id>
<op>   ::= + | -
```

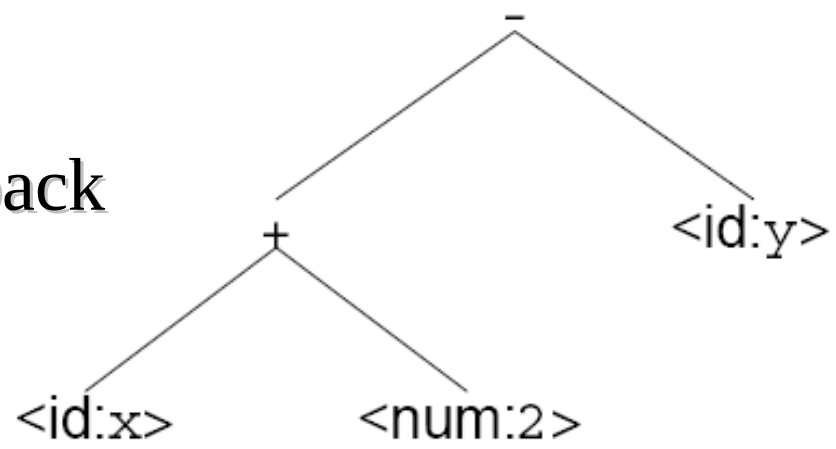
Front end

- A parser tries to map a program to the syntactic elements defined in the grammar
- A parse can be represented by a tree called a parse or syntax tree

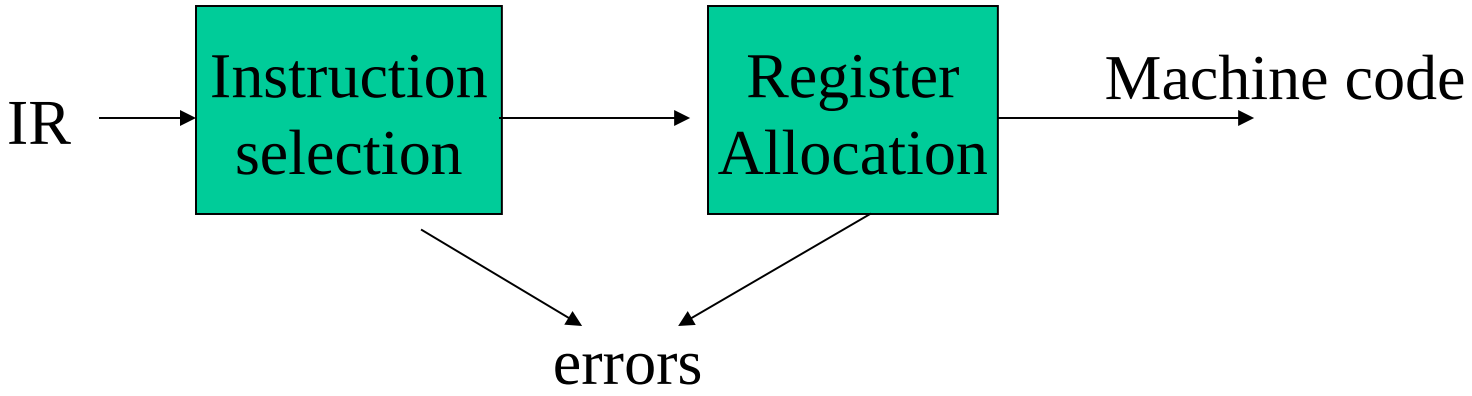


Front end

- A parse tree can be represented more compactly referred to as Abstract Syntax Tree (AST)
- AST can be used as IR between front end and back end

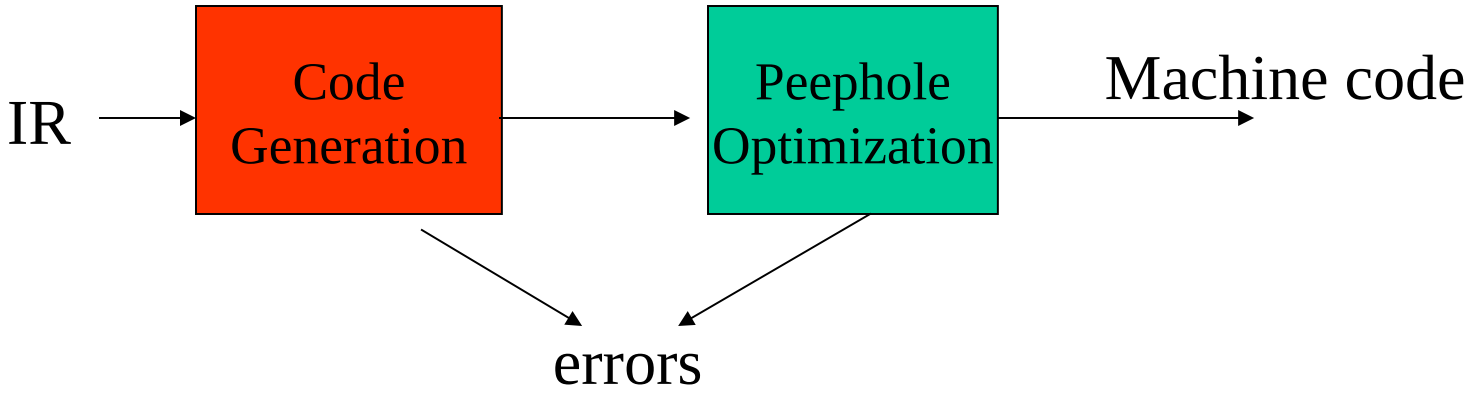


Back end



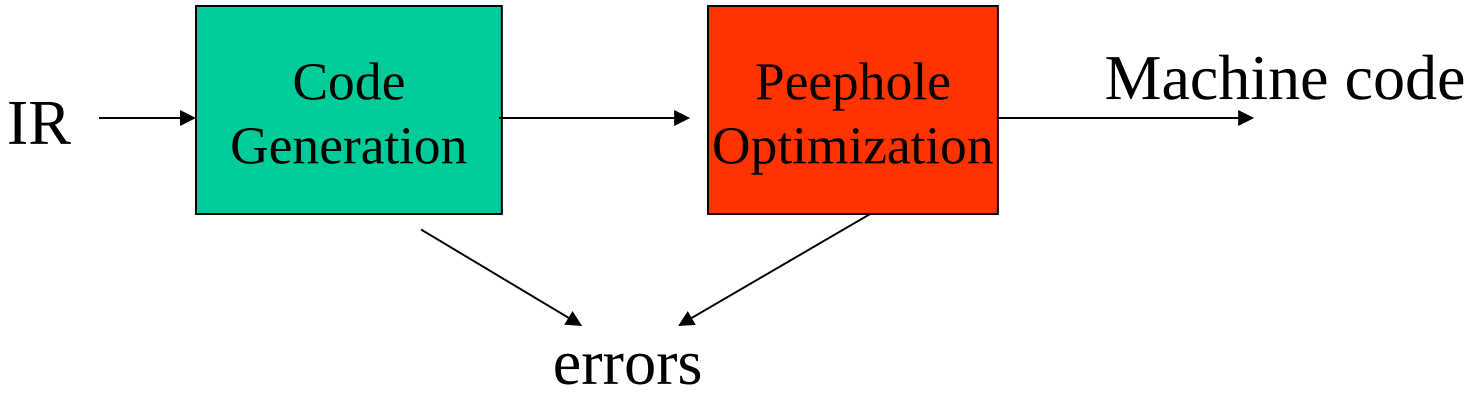
- Translate IR into target machine code
- Choose instructions for each IR operation
- Decide what to keep in registers at each point

Back end



- Produce compact fast code
- Use available addressing modes

Back end



- Limited resources
- Optimal allocation is difficult

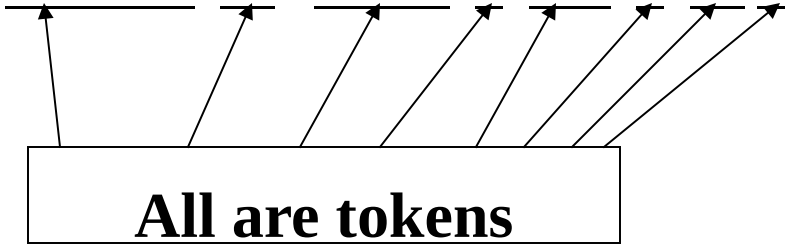
The Analysis Task For Compilation

- Three Phases:
  - **Lexical Analysis**:
    - Left-to-right Scan to Identify Tokens
      token: sequence of chars having a collective meaning
  - **Syntax Analysis**:
    - Grouping of Tokens Into Meaningful Collection
  - **Semantic Analysis**:
    - Checking to ensure Correctness of Components

Phase 1. Lexical Analysis

Easiest Analysis - Identify tokens which are the basic building blocks

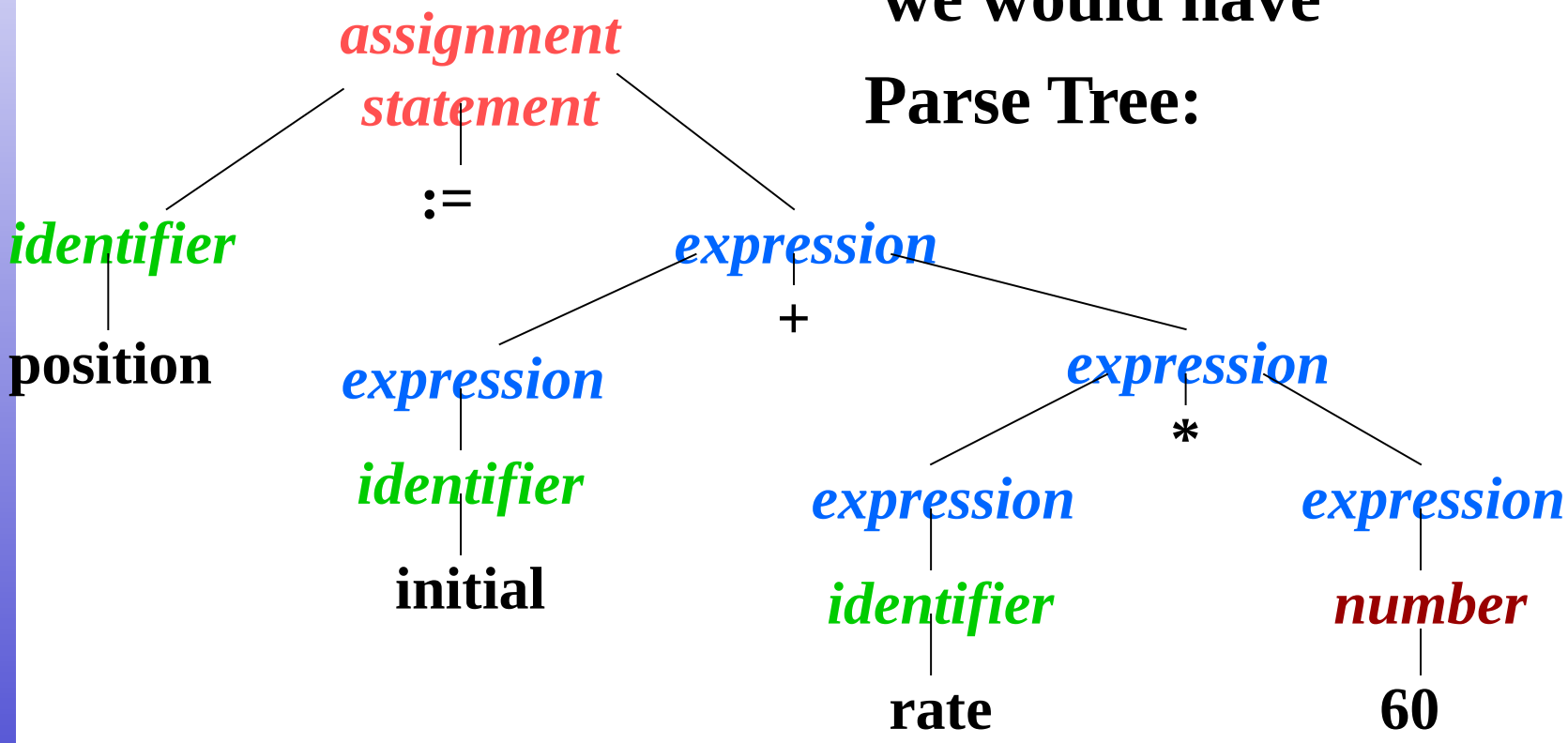
For Example: **Position := initial + rate * 60 ;**



Blanks, Line breaks, etc. are scanned out

Phase 2. Syntax Analysis or Parsing

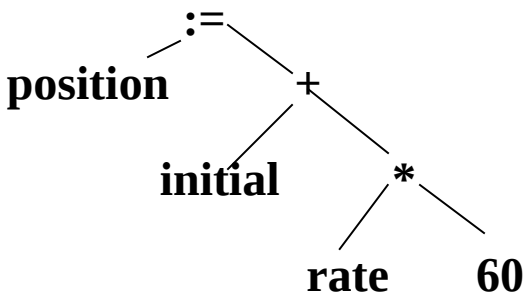
For previous example, we would have Parse Tree:



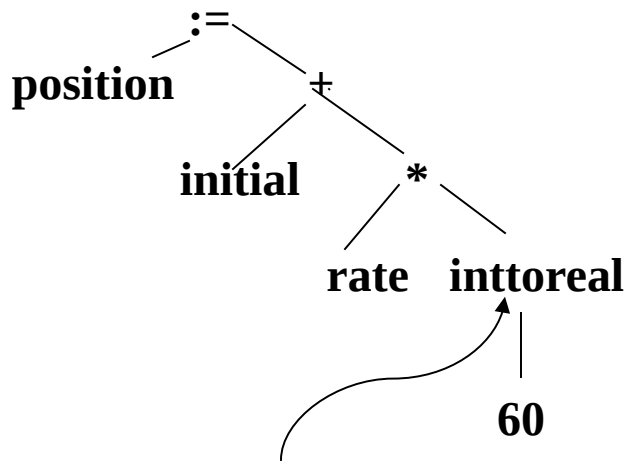
Nodes of tree are constructed using a grammar for the language

Phase 3. Semantic Analysis

- Finds Semantic Errors



Syntax Tree



Conversion Action

- One of the Most Important Activity in This Phase:
- Type Checking - Legality of Operands

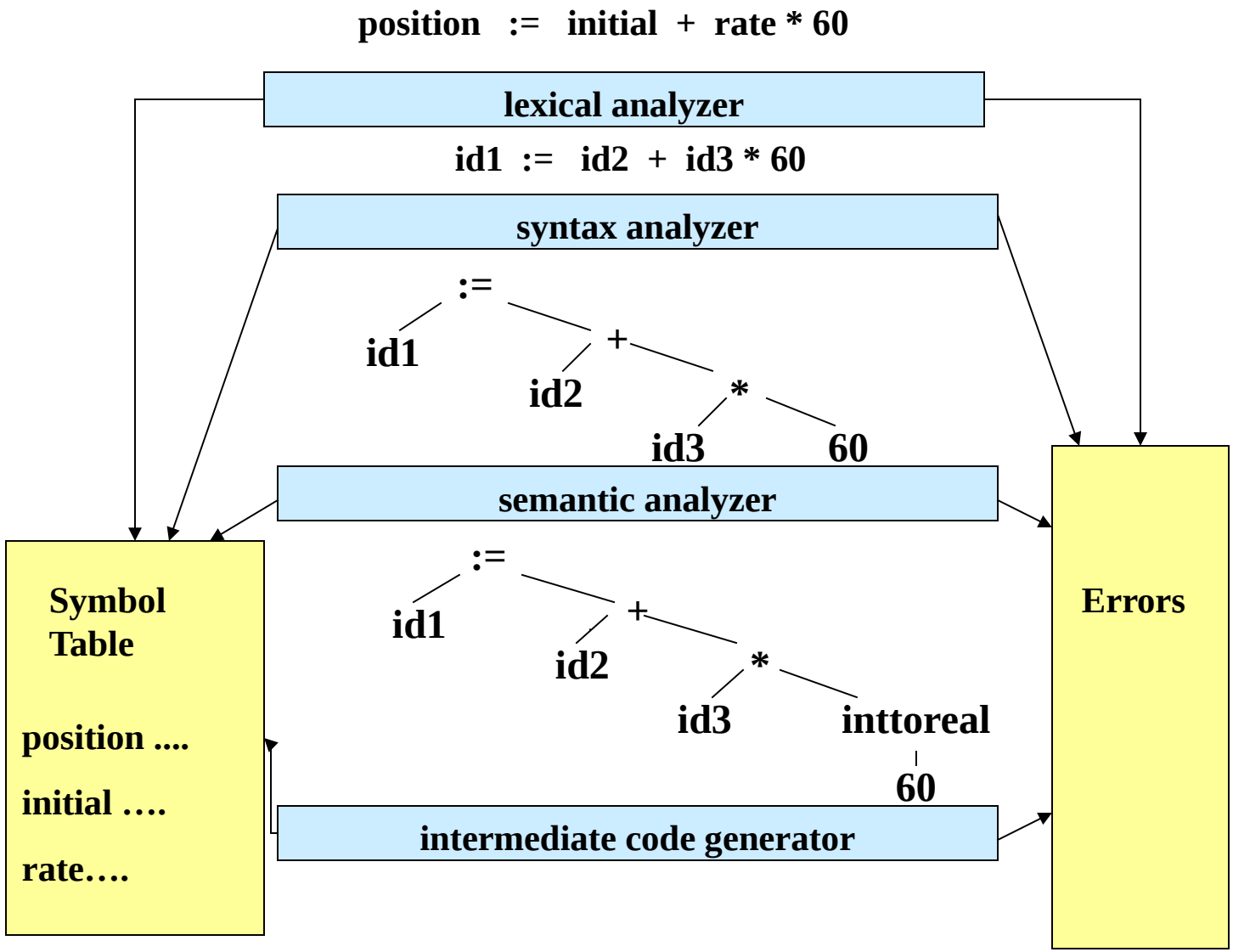
Supporting Phases/ Activities for Analysis

- Symbol Table Creation / Maintenance
  - Contains Info (storage, type, scope, args) on Each “Meaningful” Token, Typically Identifiers
  - Data Structure Created / Initialized During Lexical Analysis
  - Utilized / Updated During Later Analysis & Synthesis
- Error Handling
  - Detection of Different Errors Which Correspond to All Phases
  - What Happens When an Error Is Found?

The Synthesis Task For Compilation

- Intermediate Code Generation
  - Abstract Machine Version of Code - Independent of Architecture
  - Easy to Produce and Do Final, Machine Dependent Code Generation
- Code Optimization
  - Find More Efficient Ways to Execute Code
  - Replace Code With More Optimal Statements
- Final Code Generation
  - Generate Relocatable Machine Dependent Code
- Peephole Optimization
  - With a Very Limited View Improves Produced Final Code

Reviewing the Entire Process



Reviewing the Entire Process