KG REDDY
College of Engineering
& Technology

Compiler Design

Mr N.Srinivas
Assistant Professor-CSE

Outline

- Scope of the course
- Disciplines involved in it
- Abstract view for a compiler
- Front-end and back-end tasks
- Modules

Course scope

- Aim:
  - To learn techniques of a modern compiler
- Main reference:
  - Compilers – Principles, Techniques and Tools, Second Edition by Alfred V. Aho, Ravi Sethi, Jeffery D. Ullman
- Supplementary references:
  - Modern compiler construction in Java 2nd edition
  - Advanced Compiler Design and Implementation by Muchnick

KG REDDY College of Engineering & Technology

Subjects

- Lexical analysis (Scanning)
- Syntax Analysis (Parsing)
- Syntax Directed Translation
- Intermediate Code Generation
- Run-time environments
- Code Generation
- Machine Independent Optimization

KG REDDY College of Engineering & Technology

Grading policy

- Midterm (4-5 points)
- Final exam (7-9 points)
- Home-works (2-3 points)
- Term project (4-5 points)

Compiler learning

- Isn’t it an old discipline?
  - Yes, it is a well-established discipline
  - Algorithms, methods and techniques are researched and developed in early stages of computer science growth
  - 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

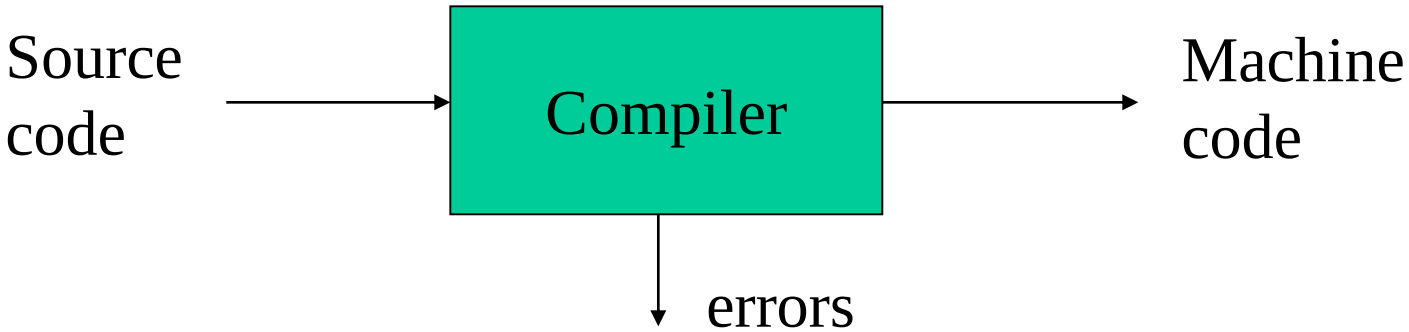
Terminology

- Compiler:
  - a program that translates an *executable* program in one language into an *executable* program in another language
  - we expect the program produced by the compiler to be better, in some way, than the original
- 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

Disciplines involved

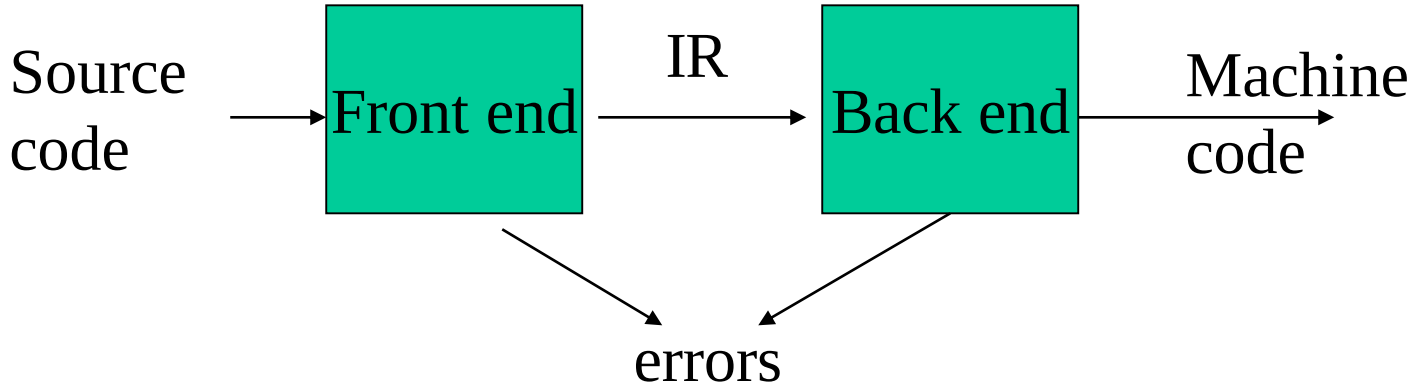
- Algorithms
- Languages and machines
- Operating systems
- Computer architectures

Abstract view



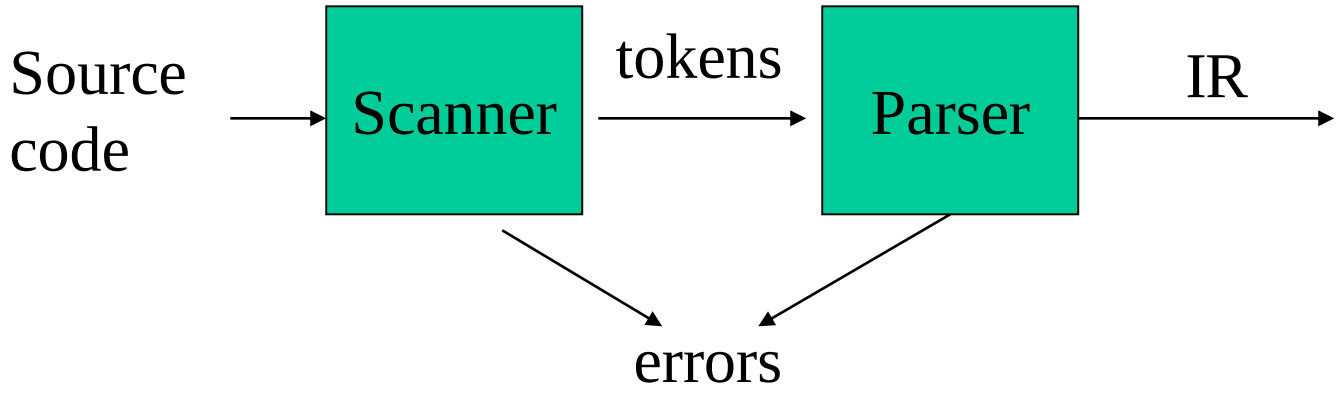
- Recognizes legal (and illegal) programs
- Generate correct code
- Manage storage of all variables and code
- Agreement on format for object (or assembly) code

Front-end, Back-end division



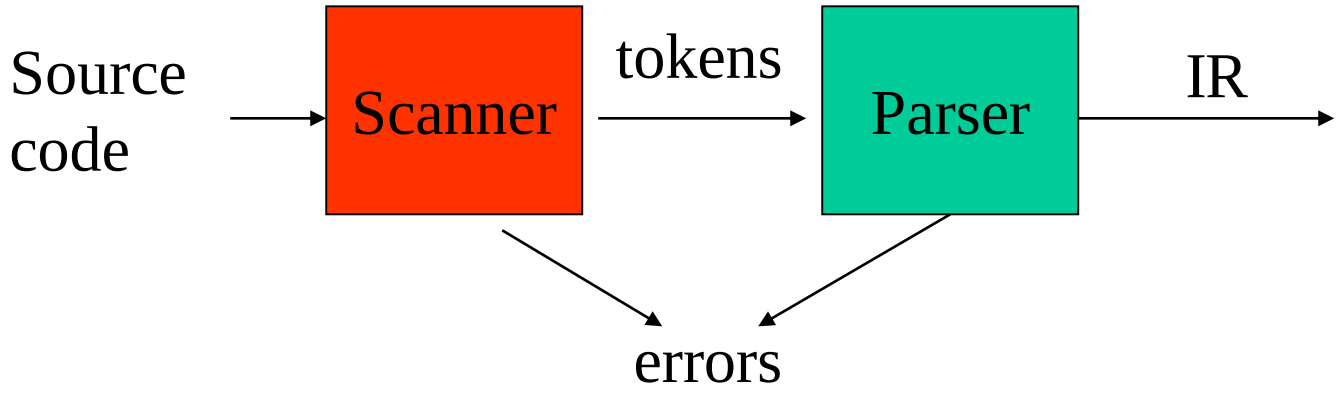
- Front end maps legal code into IR
- Back end maps IR onto target machine
- Simplify retargeting
- Allows multiple front ends
- Multiple passes -> better code

Front end



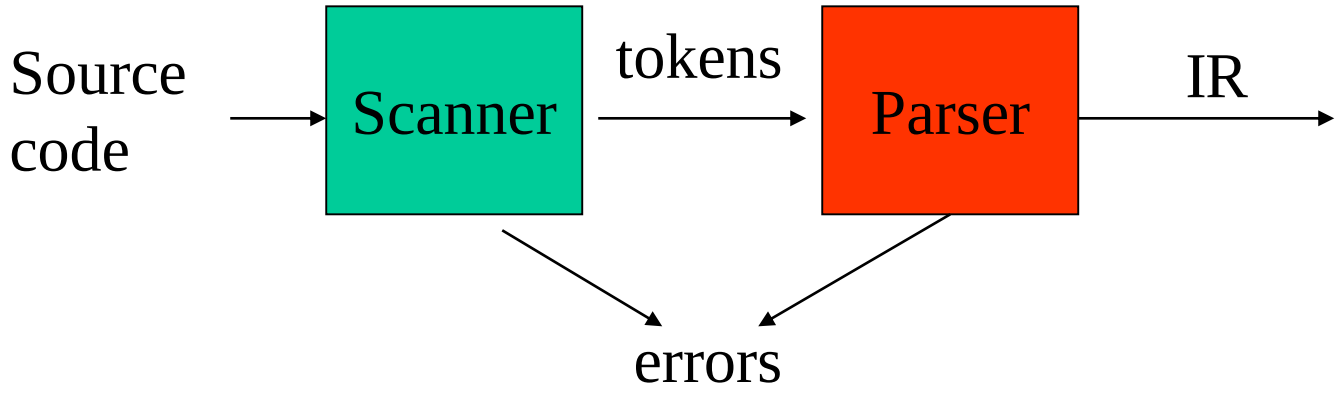
- Recognize legal code
- Report errors
- Produce IR
- Preliminary storage maps

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:

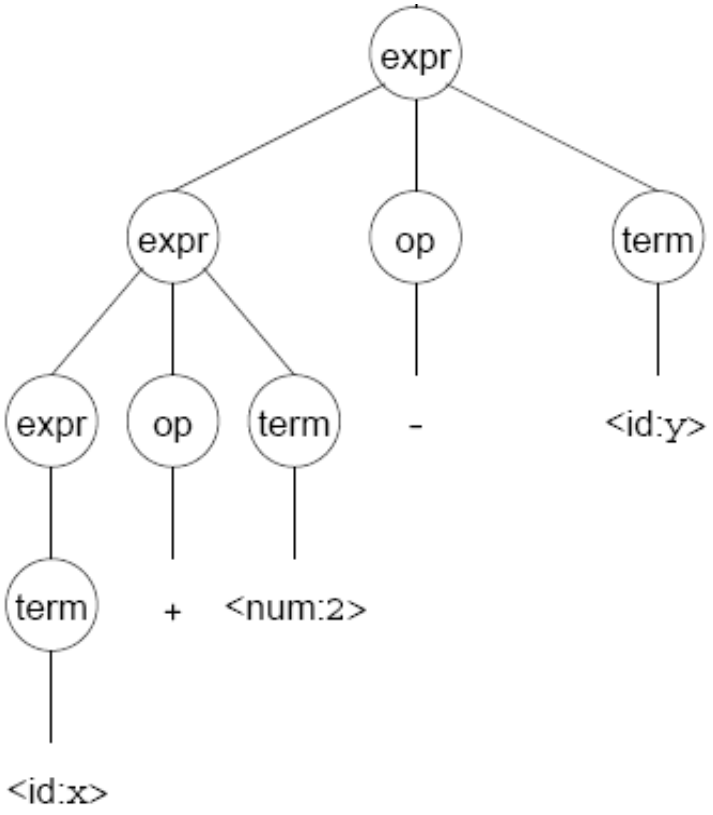
<expr> ::= <expr> <op> <term> | <term>

<term> ::= <number> | <id>

<op> ::= + | –

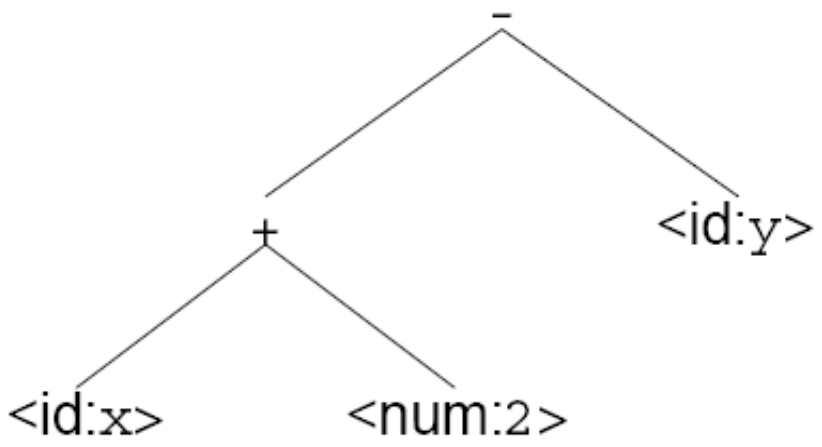
Front end

- 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

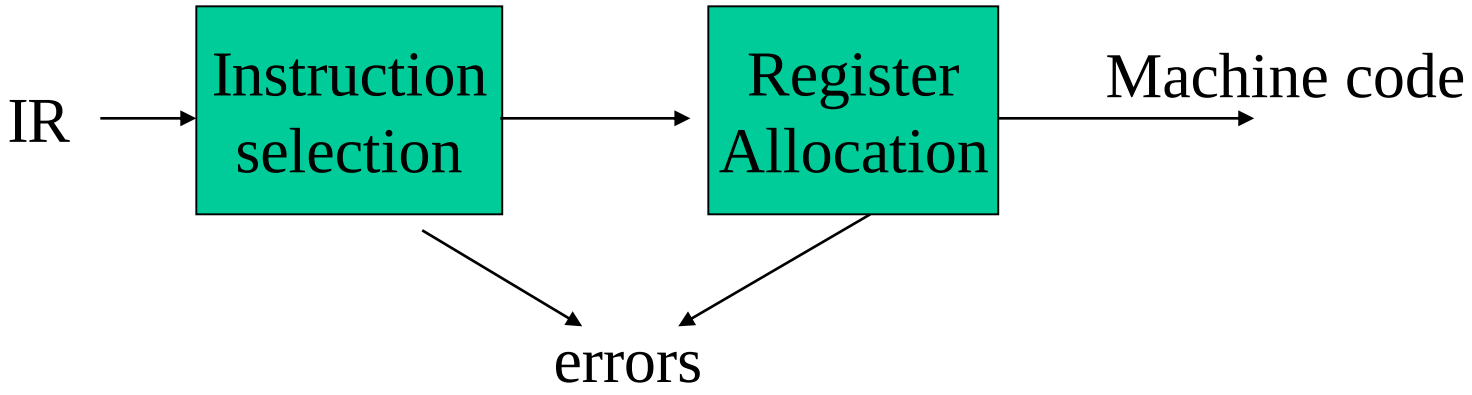


Front end

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

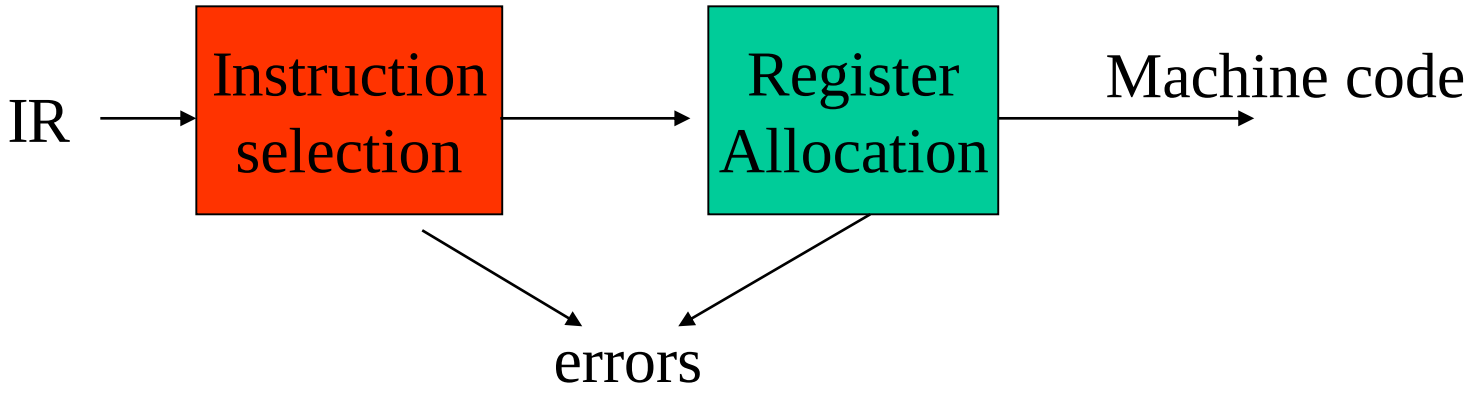


Back end



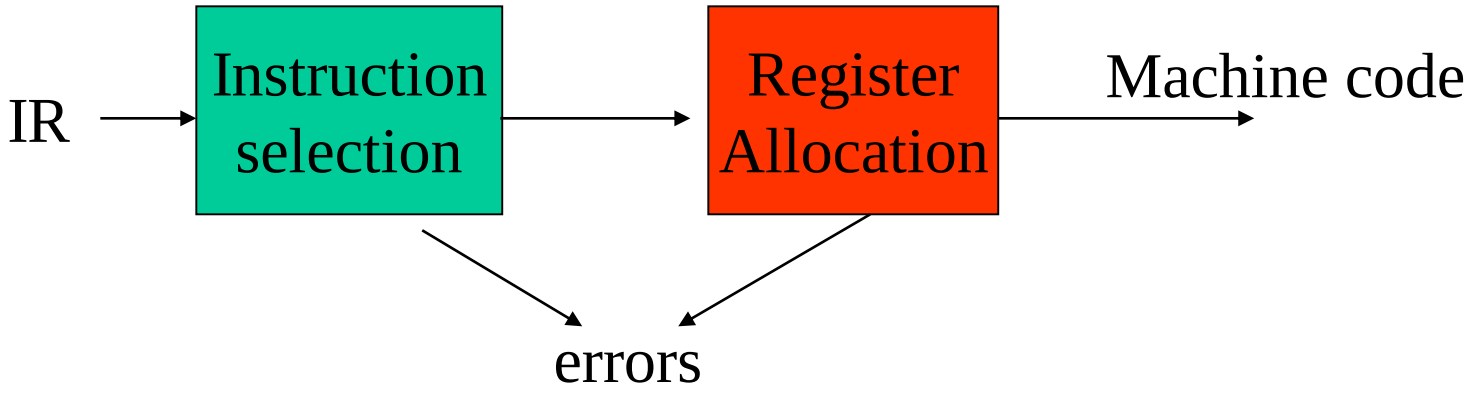
- Translate IR into target machine code
- Choose instructions for each IR operation
- Decide what to keep in registers at each point
- Ensure conformance with system interfaces

Back end



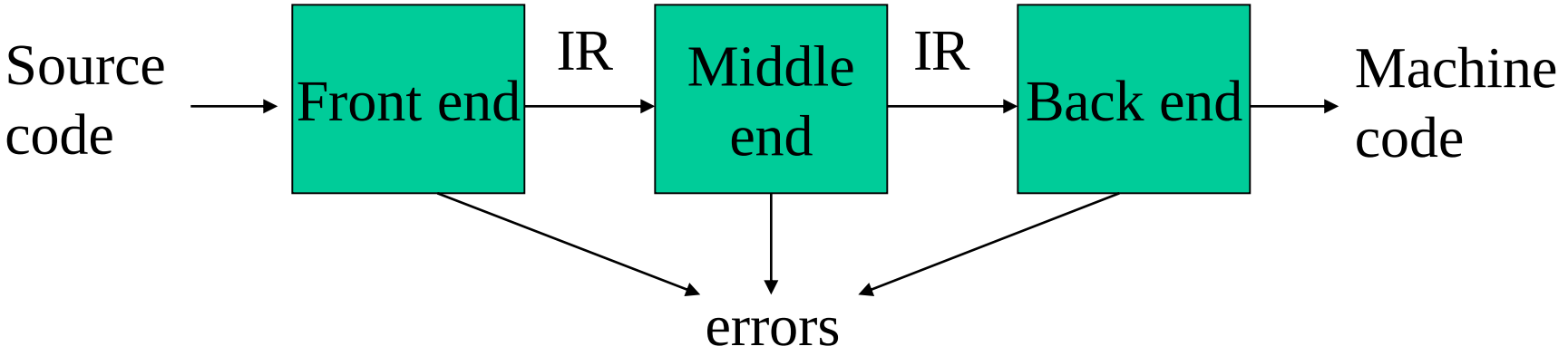
- Produce compact fast code
- Use available addressing modes

Back end



- Have a value in a register when used
- Limited resources
- Optimal allocation is difficult

Traditional three pass compiler



- Code improvement analyzes and change IR
- Goal is to reduce runtime

Middle end (optimizer)

- Modern optimizers are usually built as a set of passes
- Typical passes
  - Constant propagation
  - Common sub-expression elimination
  - Redundant store elimination
  - Dead code elimination