Chap 1. Overview of Compilation

COMP321 컴파일러

2007년 가을학기

경북대학교 전자전기컴퓨터학부

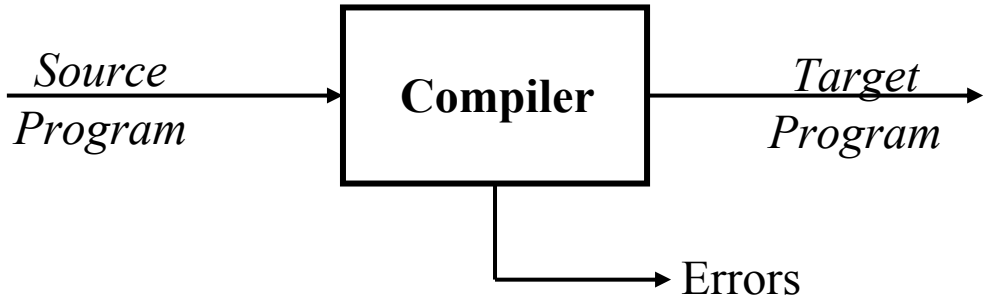
Contents

- Introduction
- Why Study Compiler Construction ?
- The Fundamental Principles of Compilation
- Compiler Structure
- Desirable Properties of a Compiler

1.1 Introduction

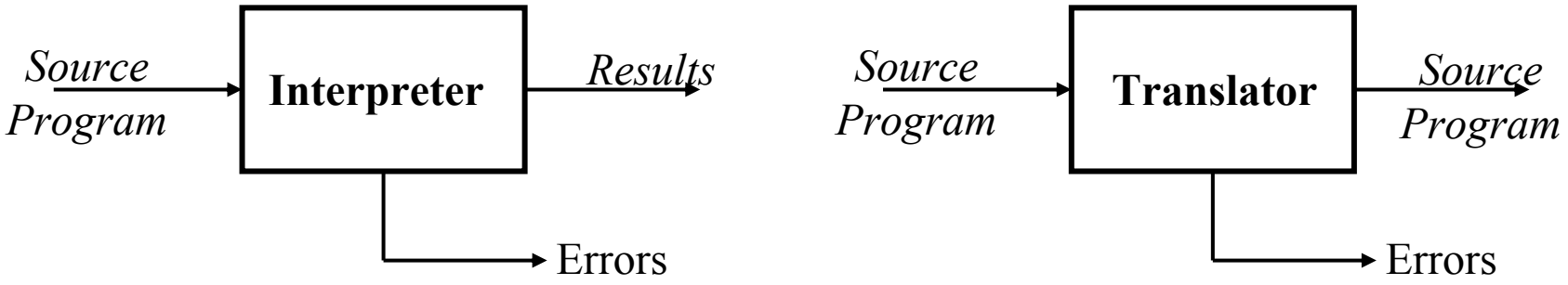
Programming Language & Compiler

- **programming language**
  - formal language with mathematical properties and well-defined meaning
    - **natural language** : ambiguities
- **compiler**
  - A program that translates an executable program in one language into an executable program in another language



Interpreter & Translator

- **interpreter**
  - A program that reads an executable program and produces the results of executing that program
- source-to-source **translator**
  - from a source code to another source code



Taking a Broader View

- Compiler Technology = **Off-Line Processing**
  - improved performance and language usability
    - Making it practical to use the full power of the language
  - **GUI 의 반대 성향**

- Examples
  - Macro expansion
  - Database query optimization
  - Language-based tools

1.2 Why Study Compiler Construction ?

Why Study Compilation?

- Compilers are important system software components
- Compilers include many applications of theory to practice
  - **computer science** 전반에 대한 지식 !

- Many practical applications have **embedded languages**
- Many applications have **input file formats**,
  - application programmer 가 반드시 알아야 할 상식

- Writing a compiler exposes practical algorithmic & engineering issues
  - approximating hard problems; efficiency & scalability

Related Areas

- Compiler construction involves ideas from many different parts of computer science

Artificial intelligence	Greedy algorithms Heuristic search techniques
Algorithms	Graph algorithms, union-find Dynamic programming
Theory (Automata)	DFAs & PDAs, pattern matching Fixed-point algorithms
Systems	Allocation & naming, Synchronization, locality
Architecture	Pipeline & hierarchy management Instruction set use

1.3 The Fundamental Principles of Compilation

Two Fundamental Principles

- compiler가 반드시 지켜야 할 사항들

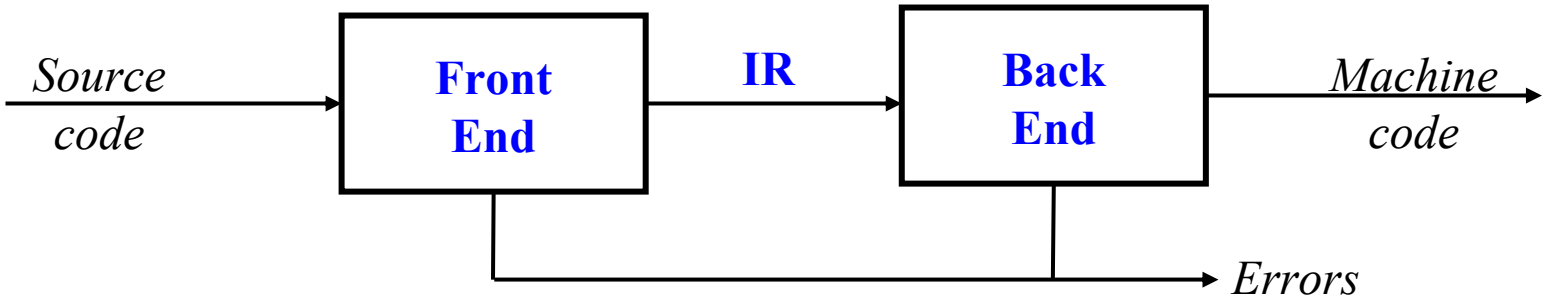
- **correctness**
  - The compiler must preserve the meaning of the source program.
    - compile 결과가 틀린 compiler는 불필요

- **practicality**
  - The compiler must improve the input program.

1.4 Compiler Structure
1.5 High-Level View of Translation

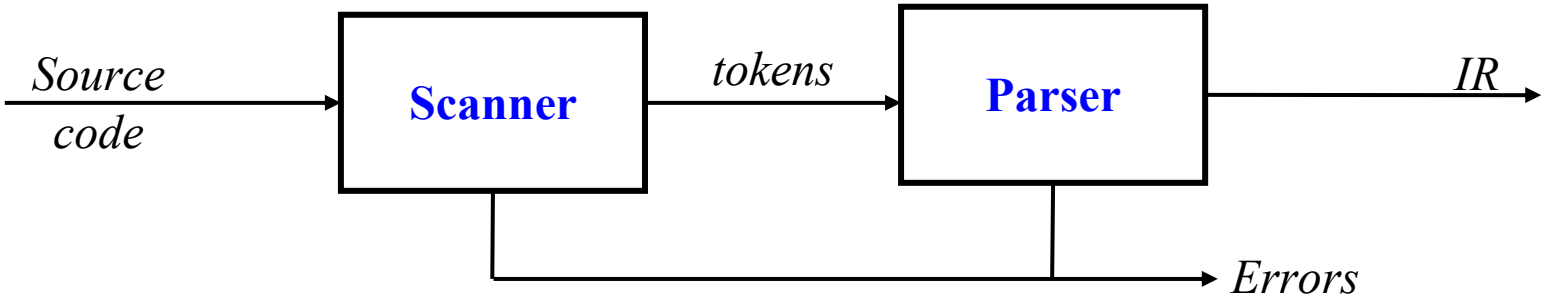
Traditional Two-pass Compiler

- compiler의 가장 기초적인 내부 구조



- IR : intermediate representation
- Front End : source program → IR
- Back End : IR → target machine code

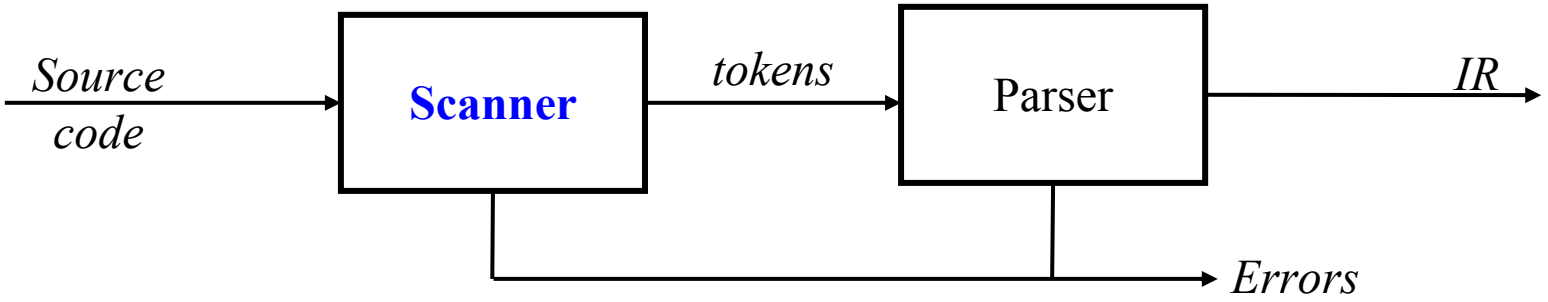
The Front End



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

The Front End : Scanner

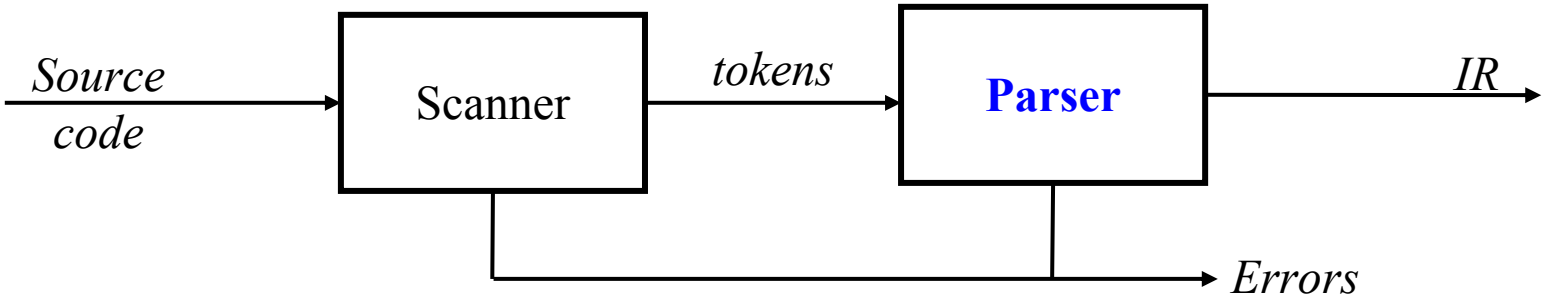


$x + 2 - y$ → <iden, *x*> <+> <number, 2> <–> <iden, *y*>

Scanner

- Maps character stream into words
  → **token : the basic unit of syntax**
- Typical tokens include *number*, *identifier*, +, –, new, while, if, …
- Scanner eliminates white space / comments

The Front End : Parser



<iden, *x*> <+> <number, 2> <–> <iden, *y*>

→ EXPR $\Rightarrow^*$ IDEN + NUM – IDEN

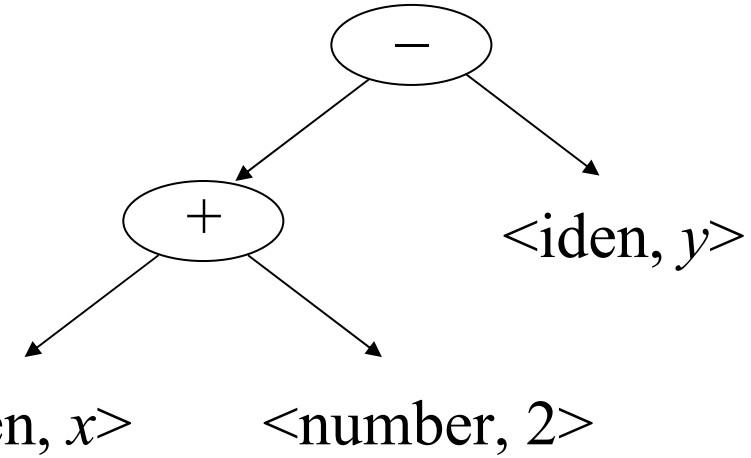
Parser

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

The Front End : AST

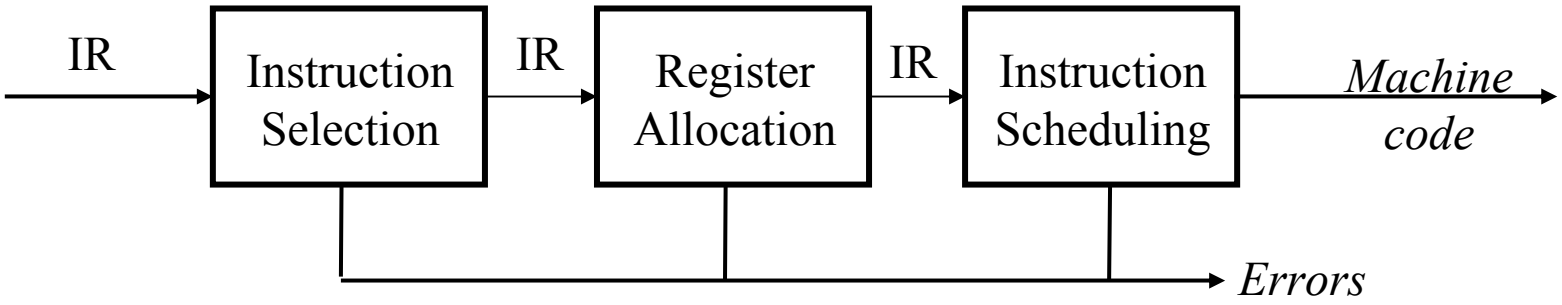
- Compilers often use an **abstract syntax tree**

x + 2 – y



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

The Back End

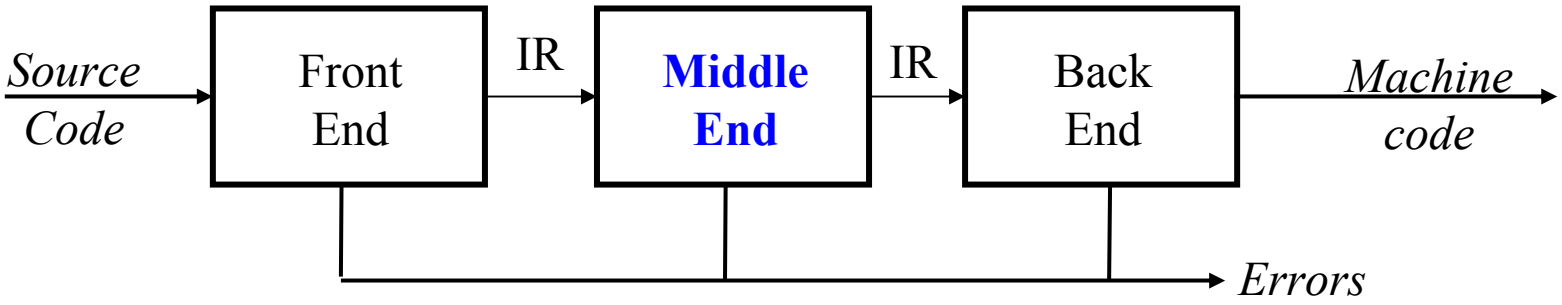


Responsibilities

- Translate IR into target machine code
- Select 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

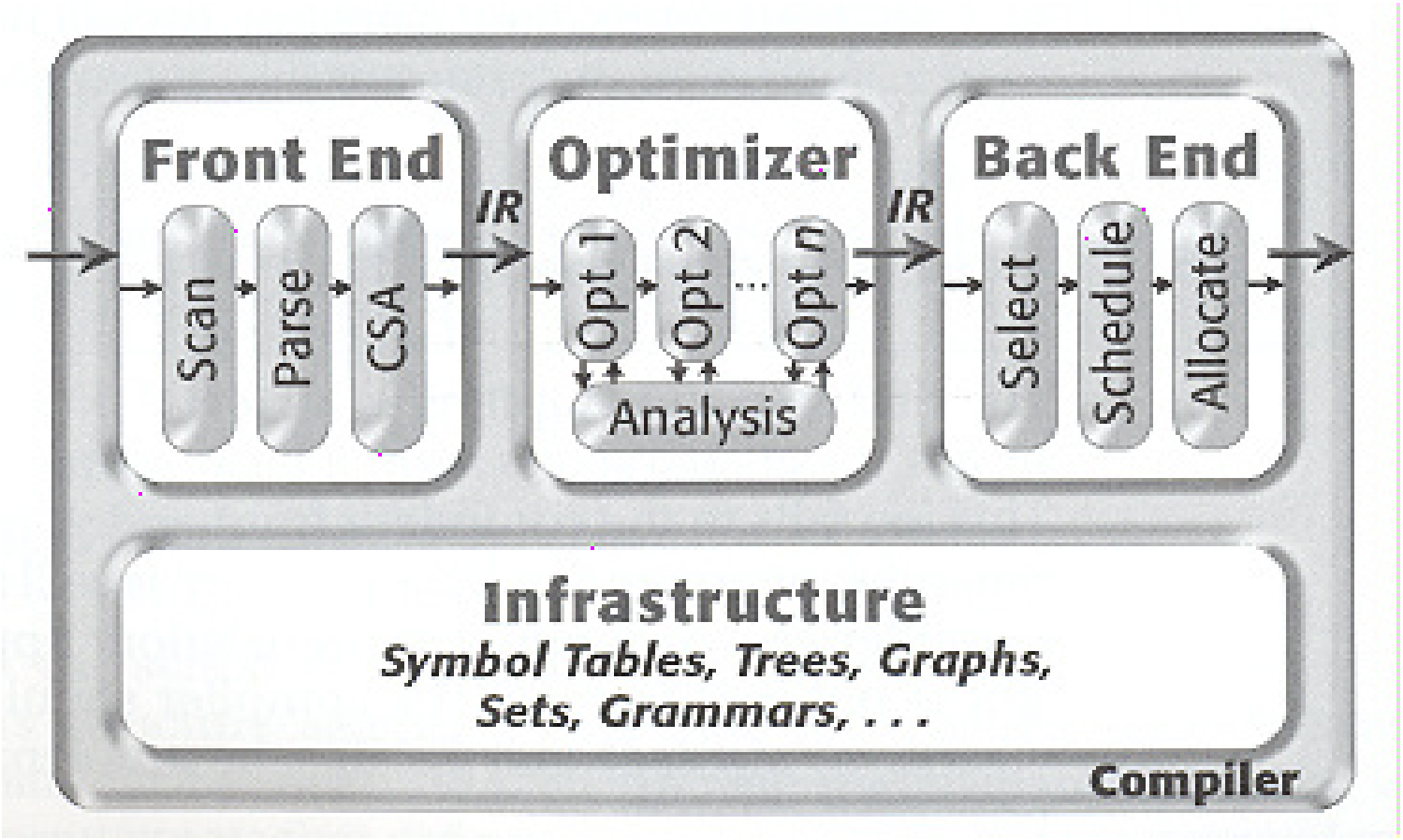
Traditional Three-pass Compiler



Code Improvement (or **Optimization**)

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to **reduce running time of the compiled code**
  - Must preserve "meaning" of the code

A Typical Compiler



1.6 Desirable Properties of a Compiler

Desirable Properties

- **speed**
  - the run-time performance of the compiled code
- **space**
  - the size of compiled code
- **feedback**
  - plentiful error messages
- **debugging**
  - a source-level debugger
- **compile-time efficiency**
  - No one likes to wait for a compiler to finish.