Chapter 1 Introduction

한양대학교 컴퓨터공학부

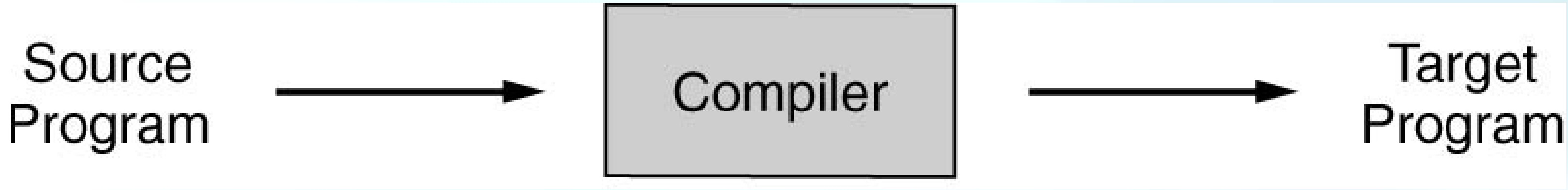
컴파일러

2014년 2학기

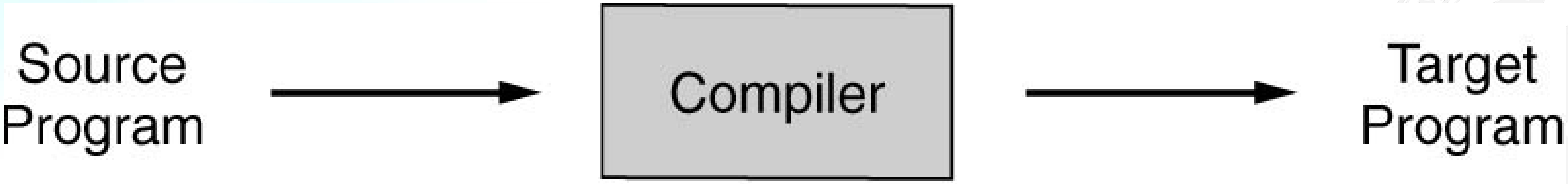
Introduction

- **Compiler**
  - A program to translate a program written in one language to the same program written in another language.

  - Input program: ***source program***
  - output program: ***target program***



Introduction



- ***Source language***: the language for the source program
- ***Target language***: the language for the target program

- Usually, the source language is a ***high-level language*** (C, C++, Java) and the target language is ***object code*** (or ***machine code***).

Why compilers? A brief history

- **Machine language**
  - Ex> **C7 06 0000 0002**
  - Hard to write and read.
  - A code written for one computer must be rewritten for another computer.

Why compilers? A brief history

- **Assembly language**
  - Ex> **MOV X, 2**
  - Easier than machine language.
  - Still not easy to write and read.
  - Extremely dependent on the particular machine
    - A code written for one computer must be rewritten for another computer.

Why compilers? A brief history

- **High-level language**
  - **X = 2**
  - nearly resembling mathematical notation or natural language
  - machine-independent
  - fears
    - might not be possible
    - the obj code would be so inefficient as to be useless
  - Compiler theory is required.

Why compilers? A brief history

- Chomsky hierarchy
  - according to the complexity of their **grammars**
  - **type 0 ~ type 3**
  - **type 2, or context-free, grammars**
    - found fairly complete solution of the parsing problem
  - finite automata and regular expressions

Programs related to compilers

- Interpreters
- Assemblers
- Linkers
- Loaders
- Preprocessors
- Editors
- Debuggers
- Profilers

The translation process

- Lexical analysis (Scanning)
- Syntax analysis (Parsing)
- Semantic analysis
- Intermediate code generation
  - Intermediate code optimization

source program analysis

(front end)

- Code generation
  - Target code optimization

target program synthesis

(back end)

example) **a[index] = 4 + 2** →

MOV R0, index

SHL R0

MOV &a[R0], 6

The translation process

- **Lexical Analysis (Scanning)**
  - Source program → **lexemes** → **tokens**
  - **Lexemes**: smallest meaningful units
    - a[index] = 4 + 2 → a / [/ index /] / = / 4 / + / 2
    - They are similar to words in natural languages.
    - I am a boy → I / am / a / boy

The translation process

- **Tokens**: categories of lexemes

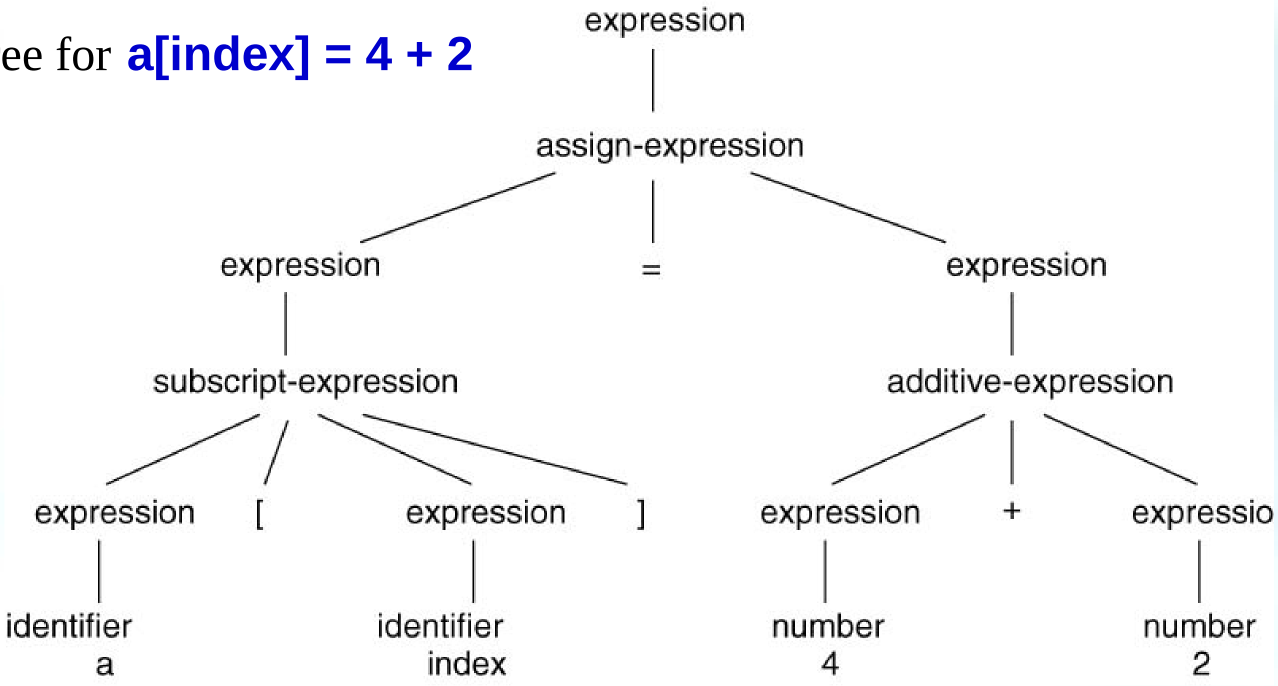
a[index] = 4 + 2

lexemes	tokens
a	identifier
[	left bracket
index	identifier
]	right bracket
=	assignment
4	number
+	plus sign
2	number

The translation process

- **Syntax Analysis (Parsing)**
  - **similar to performing grammatical analysis on a sentence**
  - **tokens →parse tree** or (**abstract**) **syntax tree**

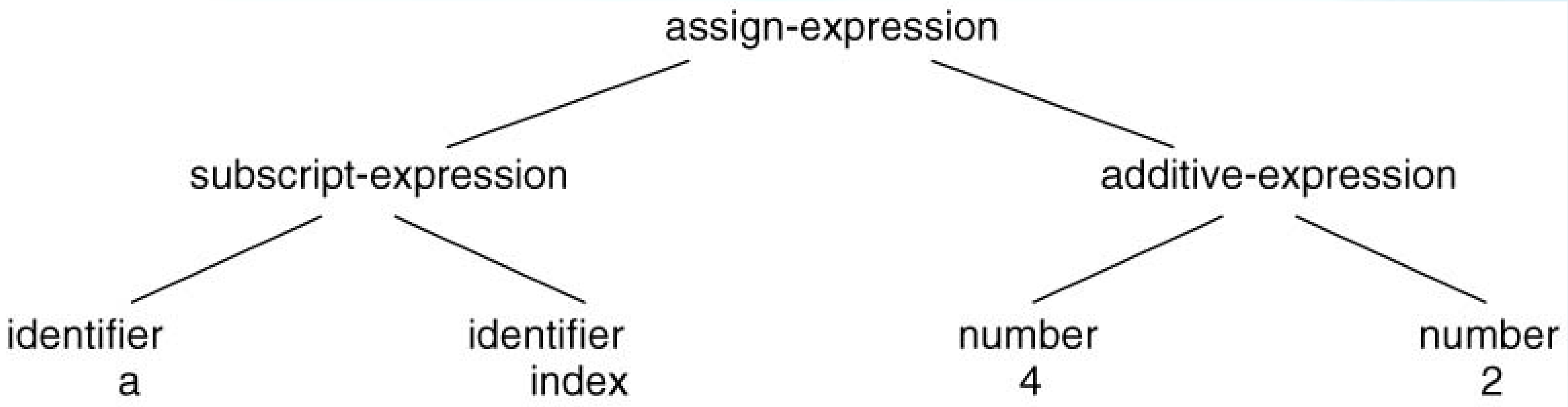
Parse tree for **a[index] = 4 + 2**



The translation process

- Syntax trees (or abstract syntax trees) are simpler than parse trees.

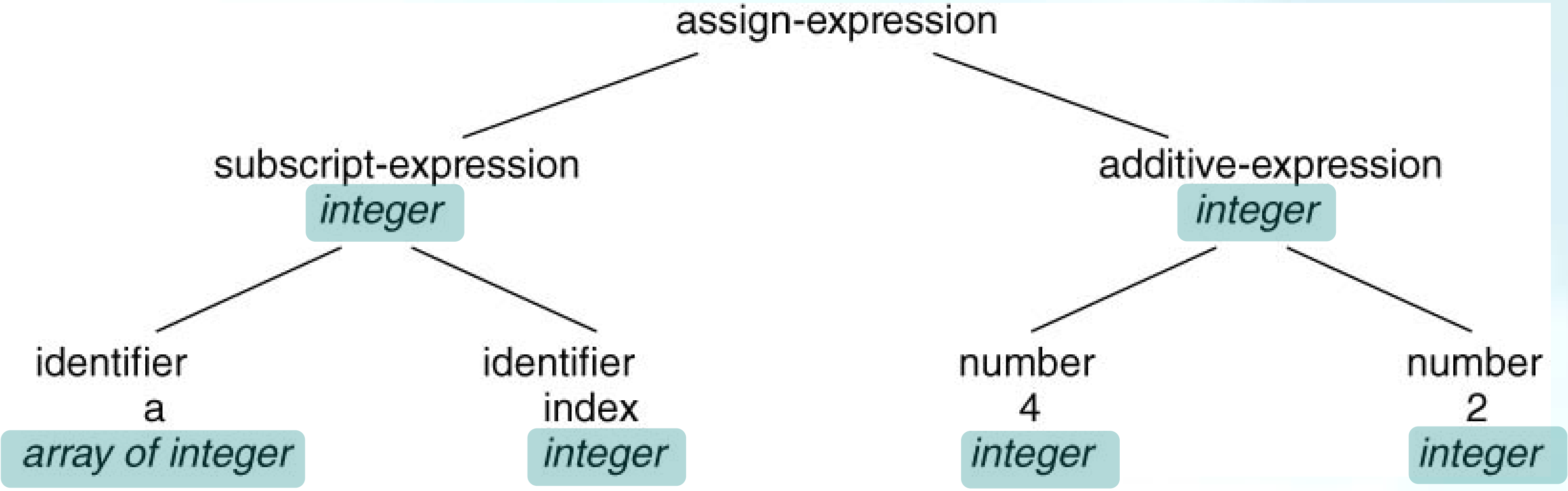
syntax tree for **a[index] = 4 + 2**



The translation process

- **Semantic Analysis**
  - **parse tree** or **syntax tree → annotated tree**
    - Attribute computation
  - declarations and type checking

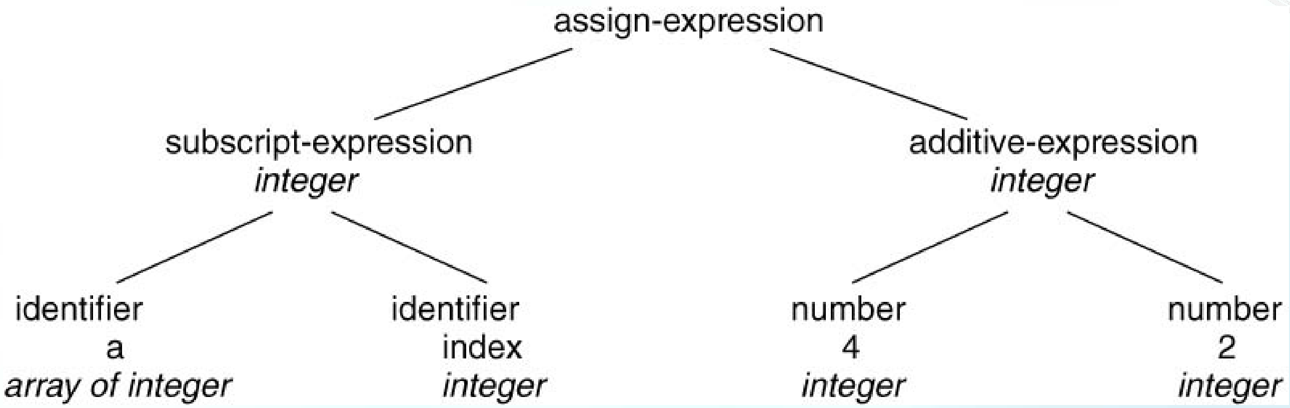
Annotated tree for **a[index] = 4 + 2**



The translation process

- **Source code optimization**
  - **annotated tree → intermediate code**

Annotated tree for **a[index] = 4 + 2**



intermediate code

t = 4 + 2

a[index] = t

The translation process

- **Intermediate code optimization**
  - **intermediate code → optimized intermediate code**
    - *constant folding*

```
t = 4 + 2         →        t = 6            →
a[index] = t               a[index] = t            a[index] = 6
```

The translation process

- Code generation
  - optimized intermediate code → target code

a[index] = 6 →

```
MOV R0, index
MUL  R0, 2
MOV R1, &a
ADD  R1, R0
MOV  *R1, 6
```

The translation process

- target code optimization
  - **target code → optimized target code**

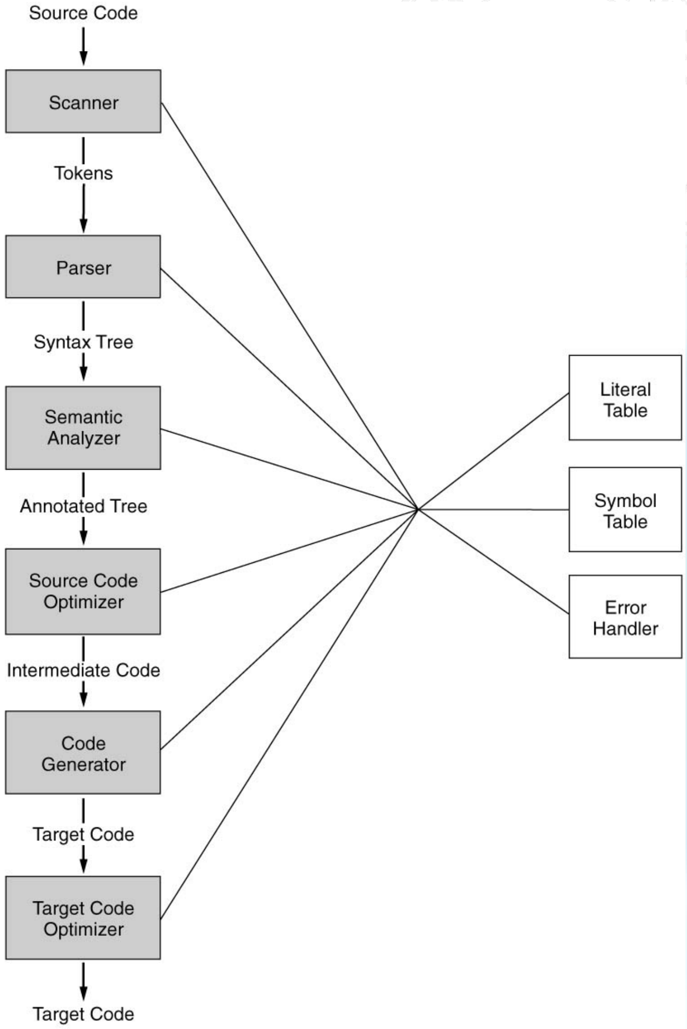
```
MOV R0, index
MUL  R0, 2
MOV R1, &a
ADD  R1, R0
MOV  *R1, 6
```

→

```
MOV R0, index
SHL  R0
MOV &a[R0], 6
```

The translation process

- Lexical analysis (Scanning)
- Syntax analysis (Parsing)
- Semantic analysis
- Intermediate code generation
  - Intermediate code optimization
- Code generation
  - Target code optimization



Major data structures in a compiler

- tokens
- the syntax tree
- the symbol table
  - Accessed frequently
  - Insertion/deletion/access ops to be efficient
  - Hash table
- the literal table
  - No deletion
  - Need to reduce the size of the table
- intermediate code
- temporary files

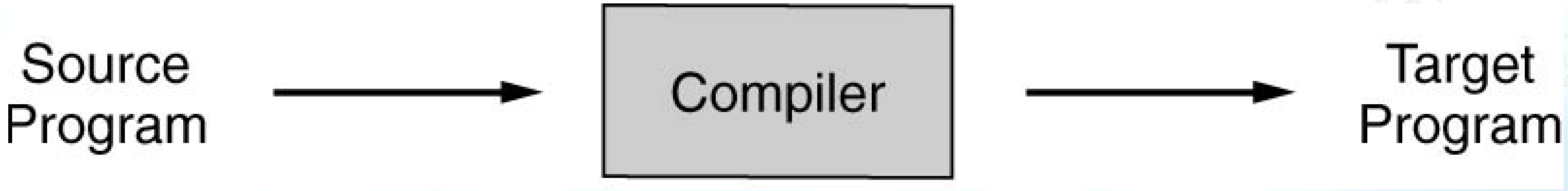
Pass

- One pass
  - Efficient compilation
  - Less efficient target code
- Permit one-pass
  - Pascal, C
- At least two-pass
  - Modula-2
- Typically
  - One pass for scanning and parsing
  - One pass for semantic analysis and source-level optimization
  - One pass for code generation and target-level optimization

Language definition and compilers

- Techniques available to the compiler writer
  - Can have a major impact on the definition of the language
- Language standard
  - Can provide a more common situation for the compiler
  - FORTRAN, Pascal, and C have ANSI standards
- Runtime environments are affected by
  - Structure of data allowed in a PL
  - Kinds of function calls and returned values allowed
- Three basic types of runtime environments
  - Static
    - FORTRAN77
    - No pointers or dynamic allocation, no recursive function calls
  - Semi-dynamic
    - Pascal, C
    - Limited form of dynamic allocation and recursive calls
  - Fully dynamic
    - LISP, Smalltalk
    - All allocation is performed automaticaly

Bootstrapping and Porting



for language B

Compiler for language A
written in language B

Running compiler
for language A

Why compilers? A brief history

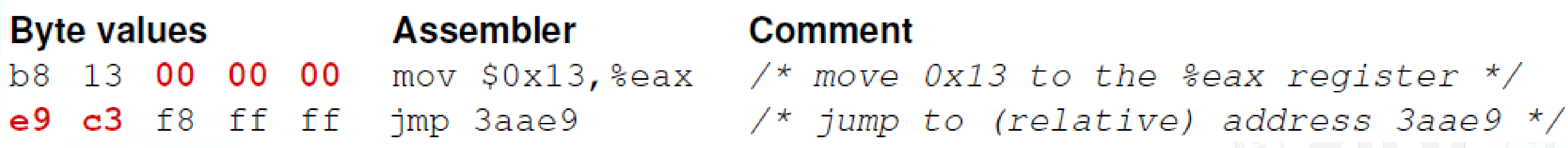
- **Machine language**
  - Ex> **C7 06 0000 0002**
  - Hard to write and read.
  - A code written for one computer must be rewritten for another computer.

Why compilers? A brief history

- **Assembly language**
  - Ex> **MOV X, 2**
  - Easier than machine language.
  - Still not easy to write and read.
  - Extremely dependent on the particular machine
    - A code written for one computer must be rewritten for another computer.

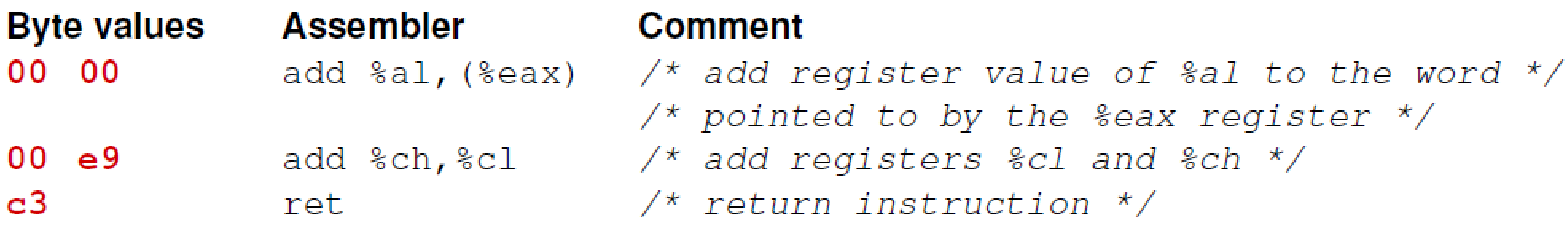
의도되지 않은 명령 순차 찾기

- libc에 다음 명령들이 포함되어 있다고 가정



Byte values	Assembler	Comment
b8 13 00 00 00	mov $0x13,%eax	/* move 0x13 to the %eax register */
e9 c3 f8 ff ff	jmp 3aae9	/* jump to (relative) address 3aae9 */

- b8에서부터 바이트 스트림을 해석하지 않고, 세 번째 바이트인 00에서부터 해석하면 다음의 unintended instruction sequence를 얻을 수 있음



Byte values	Assembler	Comment
00 00	add %al,(%eax)	/* add register value of %al to the word */
		/* pointed to by the %eax register */
00 e9	add %ch,%cl	/* add registers %cl and %ch */
c3	ret	/* return instruction */