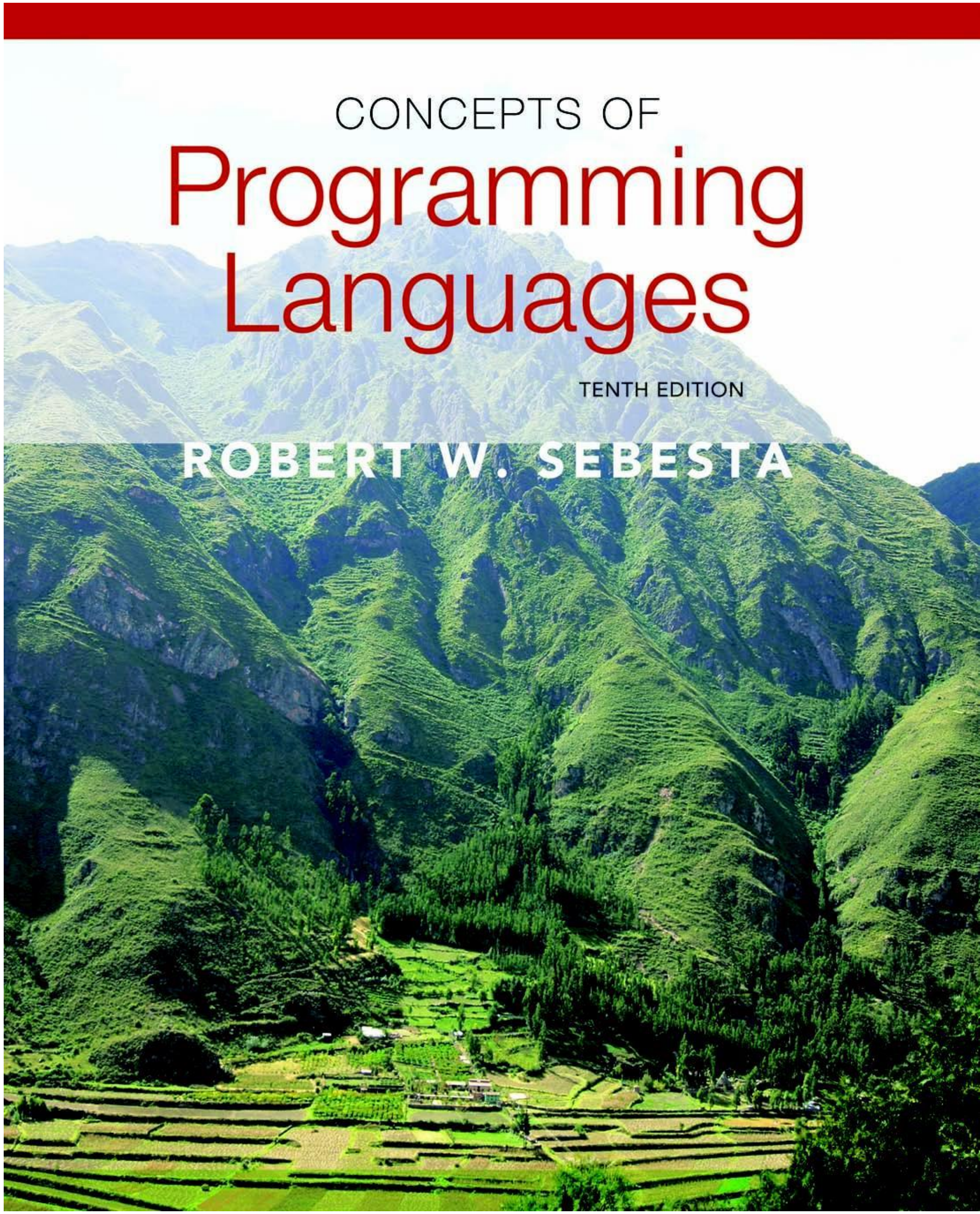
Chapter 4

Lexical and Syntax Analysis



Chapter 4 Topics

- Introduction
- Lexical Analysis
- The Parsing Problem
- Recursive-Descent Parsing
- Bottom-Up Parsing

Introduction (1/4)

- Three different approaches to implementing programing languages are compilation, pure interpretation, and hybrid implementation

- All three of the implementation approaches use both lexical and syntax analyzers

두가지는 필수

Introduction (2/4)

- Language implementation systems must analyze source code, regardless of the specific implementation approach

- Nearly all syntax analysis is based on a formal description of the syntax of the source language (CFG or BNF) – ref) ch.3

context free grammer bakcus noum form

Introduction (3/4)

- Advantages of using BNF to describe syntax

  스캐너와 파서의 관계

  - Clear and concise
  - The BNF description can be used as the direct basis <u>for the syntax analyzer</u>
  - Implementation based on BNF are relatively easy to maintain because of their <u>modularity</u>

Introduction (4/4)

- Why should lexical analysis separate from syntax analysis?
  - **Simplicity:** Lexical analysis is simple → Syntax analysis is more simple
  - **Efficiency**: Lexical analysis requires a significant portion of total compilation time.

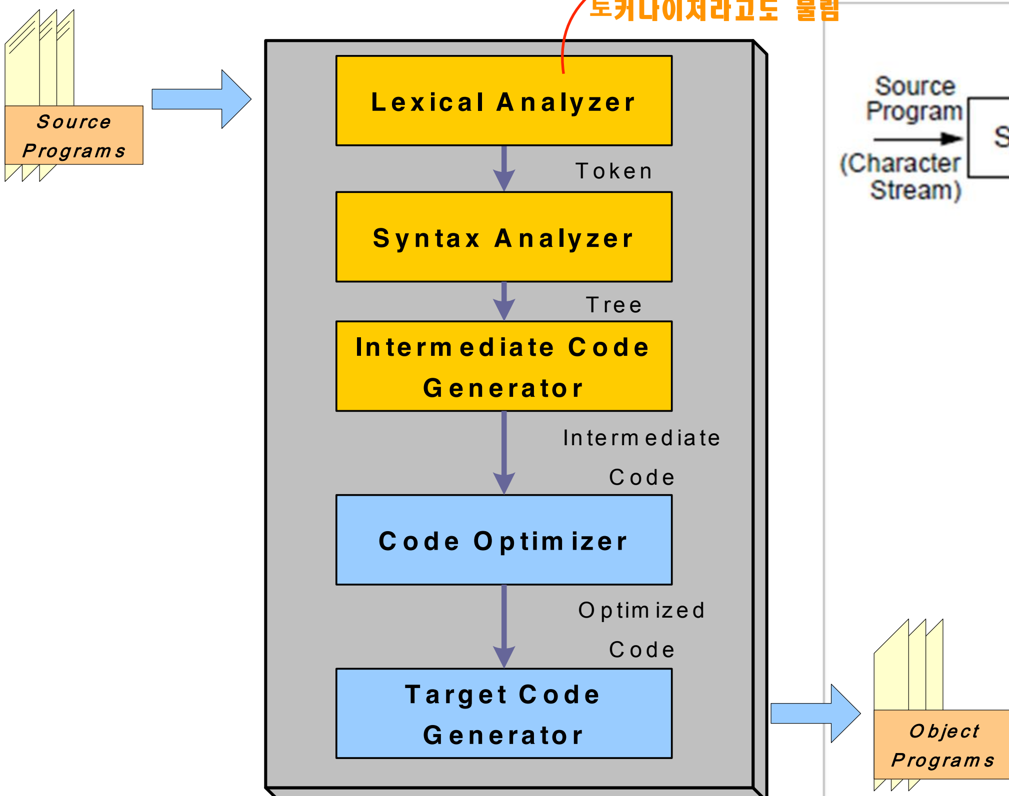
    Which one is more efficient of lexical analysis or syntax analysis for **optimization?**

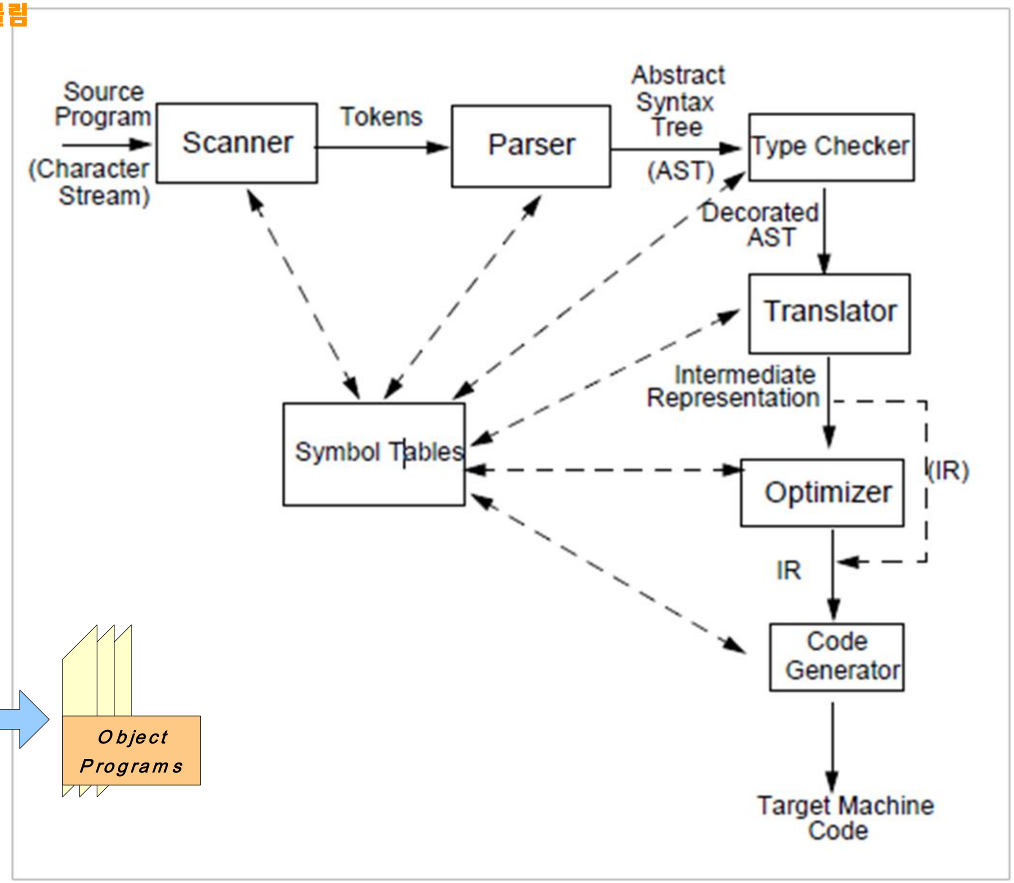
    시간적으로 렉시컬 어널라이제이션이 오래걸림
    〉최적화 할때 이쪽을 줄이는게 용이
  - **Portability:** Lexical analyzer reads input program files and often include buffering of that input.

    Which one is **machine-dependent?**

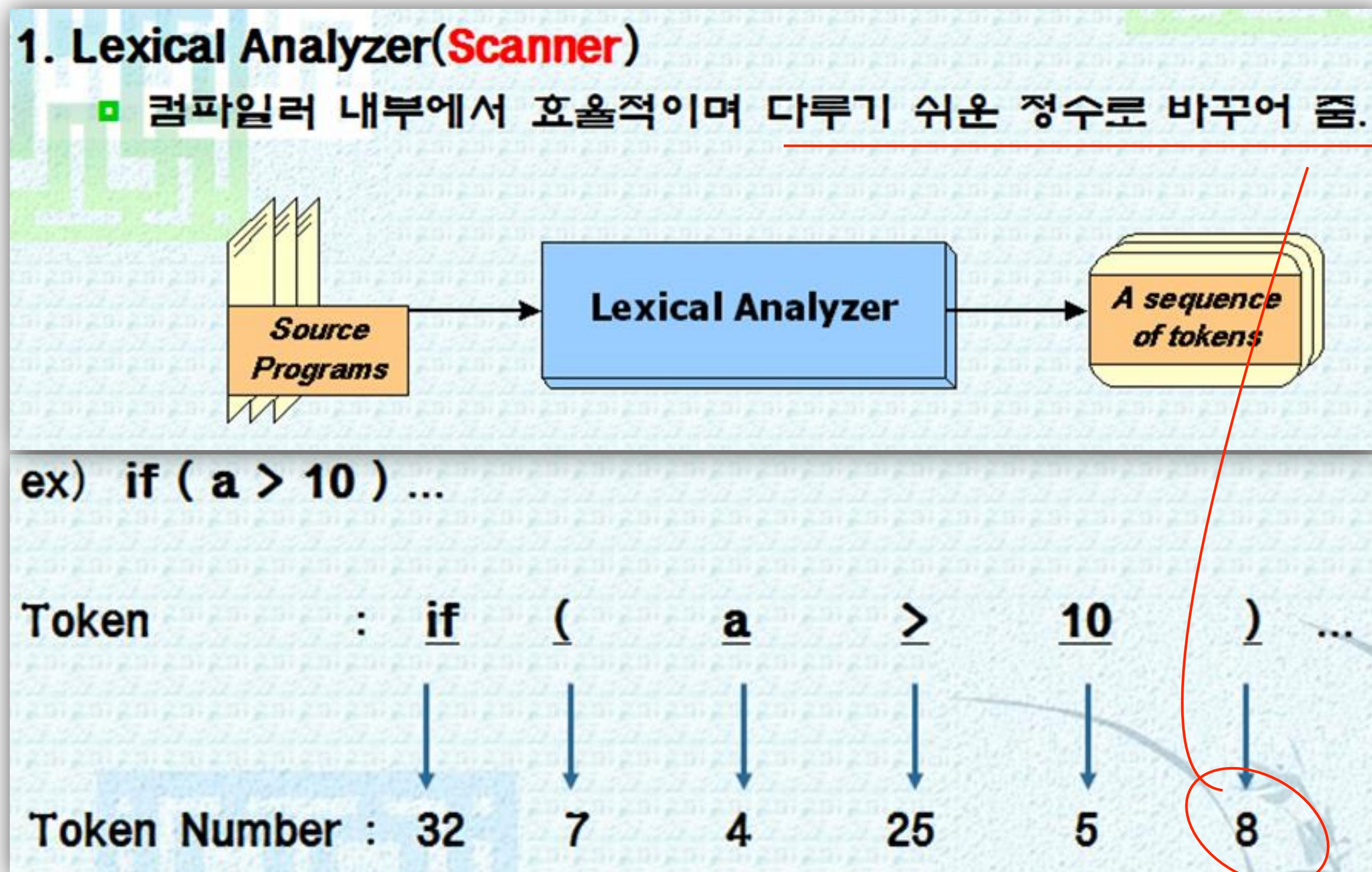
The Structure of a Compiler

=토큰별로 자르니까
토커나이저라고도 불림

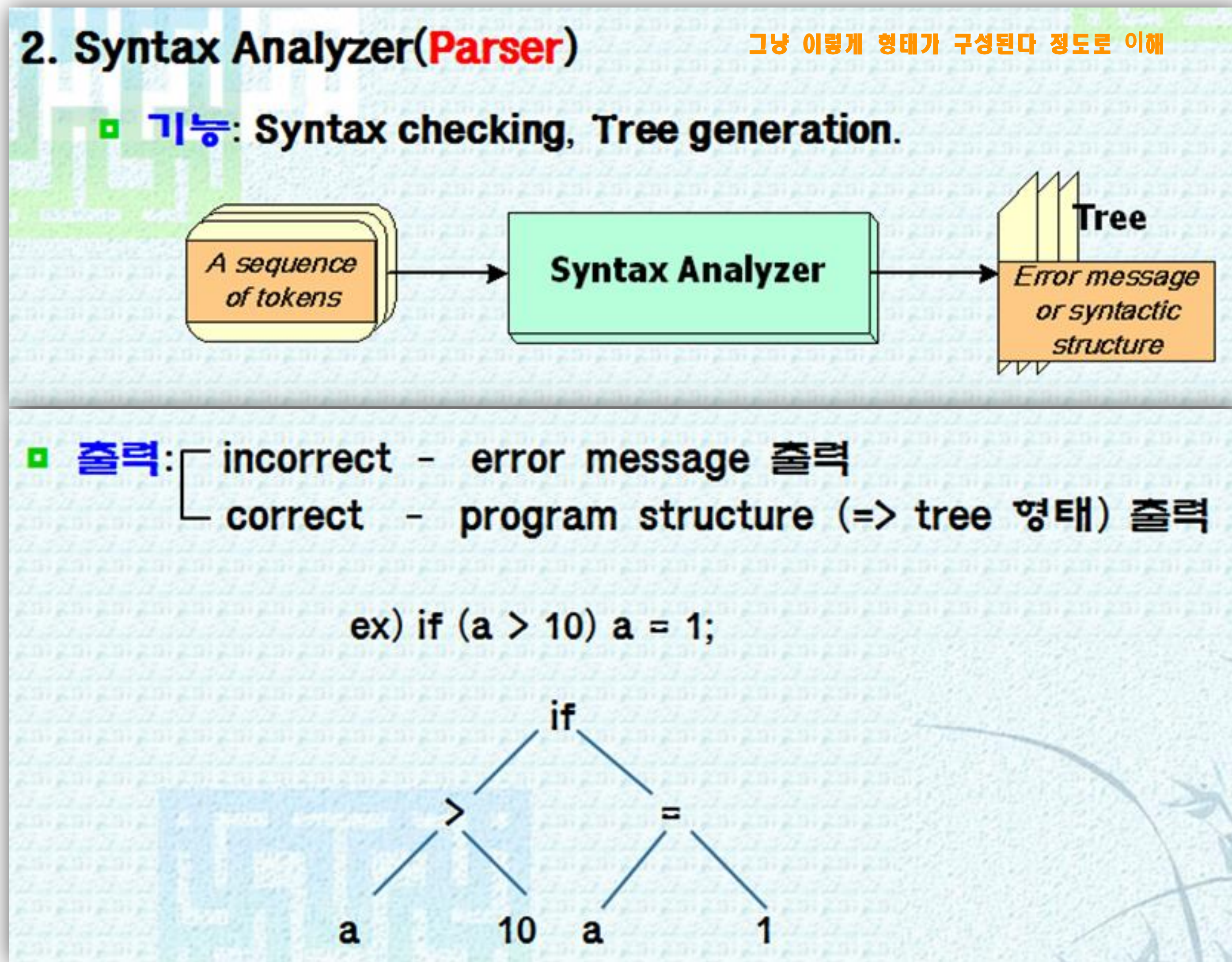




Lexical Analyzer



Syntax Analyzer (con't)



Syntax Analysis

- Technically, lexical analysis is a part of syntax analysis
- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a **lexical analyzer**
  - A high-level part called a **syntax analyzer**, or **parser**
- A lexical analyzer is a "front-end" for the parser

자이기억

- Lexical Analysis
  작은 규모의 언어구조를 처리(names, numeric literals)
- Syntax Analysis
  큰 규모의 언어구조를 처리(expressions, sentences, program units)

Lexical Analysis (con't)

- A lexical analyzer is a pattern matcher for character strings
  - Finding a substring of a given string of characters that matches a given character pattern

- Identifies substrings of the source program that belong together – *lexemes*
  - Lexemes match a character pattern, which is associated with a lexical category called a ***token***
  - **`sum`** is a lexeme; its token may be **`IDENT`**
  - **`Example (p. 190)`**

Lexical Analysis (con't)

- The **lexical analyzer** is usually a function that is called by **the parser** when it needs the next token
  - Lexical analyzers extract lexemes from a given string and produce the corresponding tokens
  - Skipping comments and white space
  - Lexical analyzer Inserts lexemes for user-define names into the symbol table, which is used by later phases of the compiler.
  - Lexical analyzers detect syntactic errors in tokens, such as ill-formed floating-point literals, and report such errors to the user

Wait a minute! Literal??

== 상수

- In computer science, a literal is a notation for representing a fixed value in source code. (ref. wikipedia)
  - Literal example (Numbers, characters, and strings)
  - 변수의 초기화에 주로 사용됨 (int i = 1; String s = “cat";)
  - Literal도 타입이 있다? (in JAVA)
    - Int num = 1; // 소수점이 없는 수치 리터럴은 기본적으로 int 형
    - Double sum = num + 0.5;//소수점이 있는 수치 리터럴은 기본적으로 double 형

Lexical Analysis (continued)

- **Three approaches to building a lexical analyzer:**
  - Write a formal description of the tokens and use a software tool that constructs **a table-driven lexical analyzer** from such a description (**LEX : generator of lexical analyzer)**
  - Design **a state diagram** that describes the tokens and write **a program** that implements the state diagram
  - Design **a state diagram** that describes the tokens and hand-construct **a table-driven implementation of the state diagram**

State Diagram Design

- A state diagram would have a transition from every state on every character in the source language – such a diagram would be very large!

State Diagram Design (con't)

- In many cases, transitions can be combined to simplify the state diagram
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent
    - Use a character class (LETTER)that includes all letters (52 characters, any uppercase or lowercase letter)

      대,소문자 26개씩

  - When recognizing an integer literal, all digits are equivalent
    - use a digit class (DIGIT)

State Diagram Design (con't)

- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word

5번문제 풀때 이 lookup table을 작성해야한다

State Diagram Design (continued)

- Convenient utility subprograms:
  (the common tasks inside the lexical analyzer)
  - **`addChar`** : puts the character in **`nextChar`** into the string array **`lexeme`** in a subprogram named **`addChar`**
  - **`getChar`** : gets the next character of input, puts it in **`nextChar`** ***`(global variable)`***, **determines its class and puts the class** in **`charClass`**

이게 지금읽어온 캐릭터임

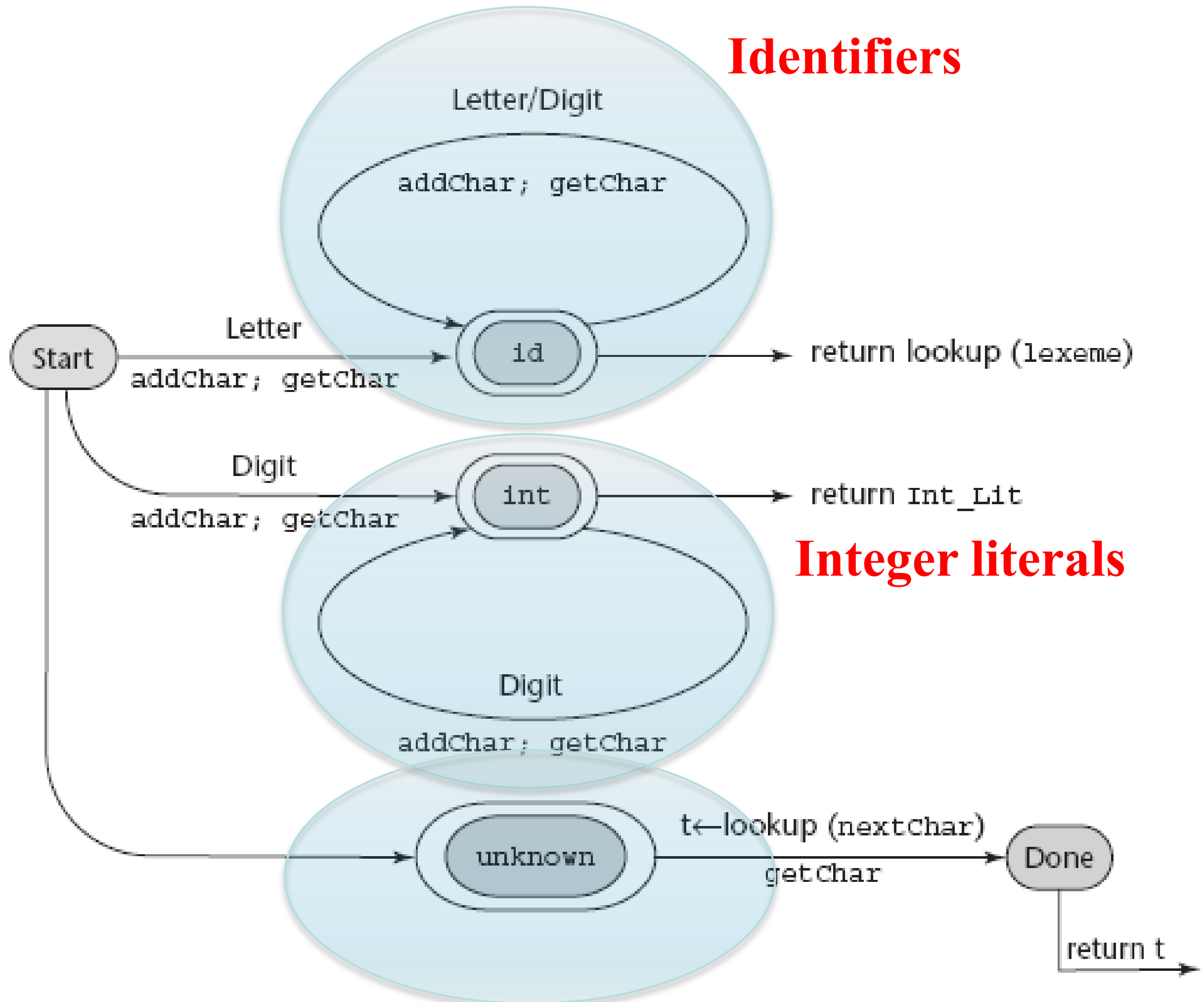
식별자, 디짓, 언노운을 구별

  - **`lookup`** : determines whether the string in **`lexeme`** is a reserved word
    (returns a code : **token code is a arbitrary number**)

State Diagram

Figure 4.1

A state diagram to recognize names, parentheses, and arithmetic operators



괄호 혹은 연산자와 같은
〉싱글캐릭터토큰

Single-character tokens (parentheses and operators)

Lexical Analyzer

Implementation:

→ SHOW **front.c** (pp. 192–197)

– Following is the output of the lexical analyzer of `front.c` when used on **`(sum + 47) / total`**

```
Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is sum
Next token is: 21 Next lexeme is +
Next token is: 10 Next lexeme is 47
Next token is: 26 Next lexeme is )
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is total
Next token is: -1 Next lexeme is EOF
```

The Parsing Problem

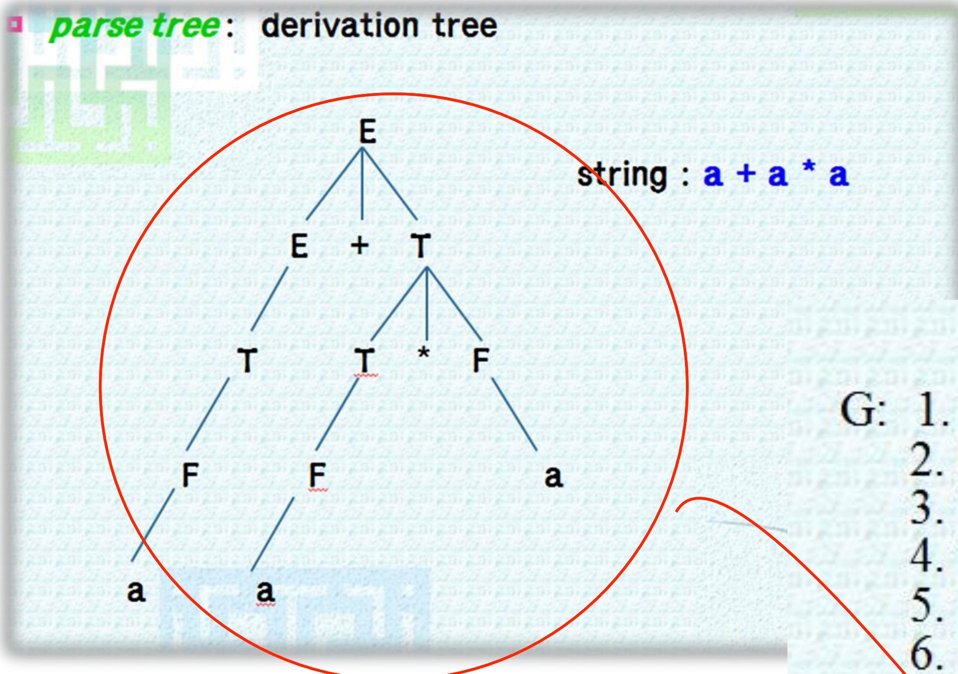
- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly
  - Produce the parse tree, or at least a trace of the parse tree, for the program
    - The parse tree (or its trace) is used as the basis for translation

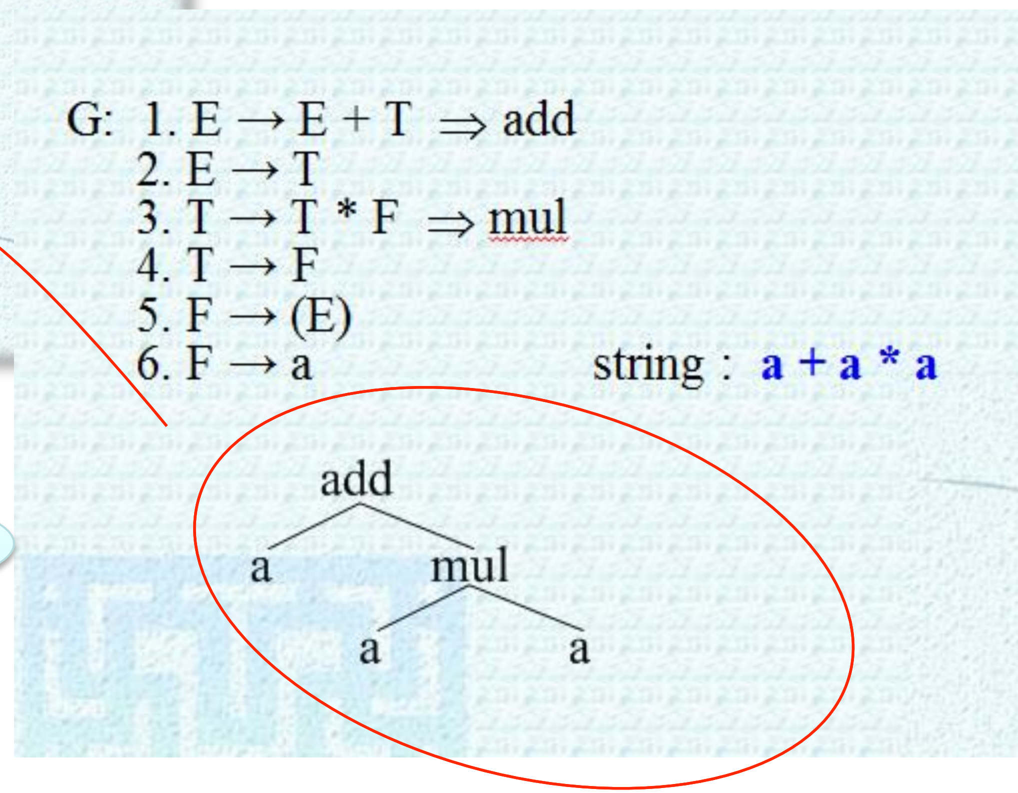
The Parsing Problem (continued)

- Two categories of parsers
  - *Top down* : produce the parse tree, beginning at the root
    - Order is that of a leftmost derivation
    - Traces or builds the parse tree in preorder
  - *Bottom up* : produce the parse tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation

Parse Tree vs Abstract Syntax Tree

조금 더 추상화, 심플해진 트리이다





AST is more efficient

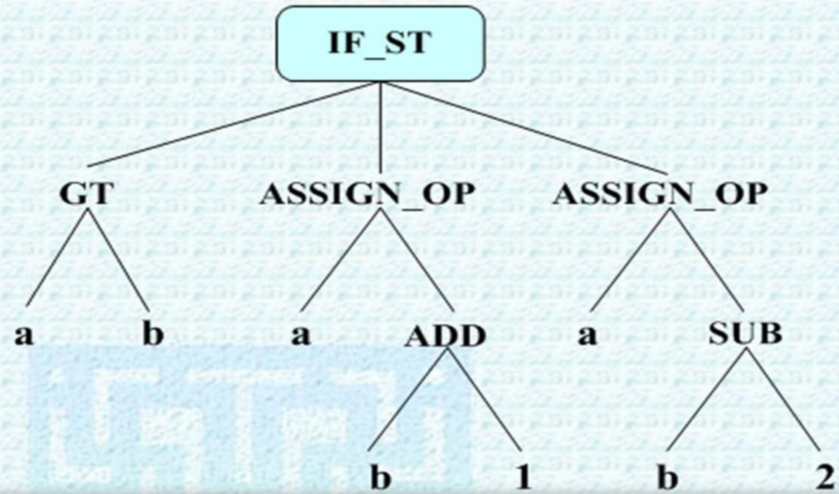
Abstract Syntax Tree

※ 의미 있는 terminal ⇒ terminal node

의미 있는 production rule ⇒ nonterminal node

→ naming : compiler designer가 지정.

ex) if (a > b) a = b + 1; else a = b – 2;



The Parsing Problem (continued)

- Top-down Parsers
  - Given a sentential form, $xA\alpha$, the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A
- The most common top-down parsing algorithms:
  - Recursive descent – a coded implementation
  - LL parsers – table driven implementation

The Parsing Problem (continued)

- Bottom-up parsers
  - Given a right sentential form, α, determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
  - The most common bottom-up parsing algorithms are in the **LR** family

The Parsing Problem (continued)

- The Complexity of Parsing
  - Parsers that work for any unambiguous grammar are complex and inefficient ($O(n^3)$), where n is the length of the input)
  - Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time ($O(n)$), where n is the length of the input)


감사합니다