1장. 컴퓨테이션 역사
2 장. 프로그램의 탄생(컴퓨터 조직)
3장: 프로그래밍 언어의 발전
4장: 고급 프로그래밍 언어
5 장: 컴파일 언어 번역

5장: 고급 언어 번역: 컴파일러

2024년 컴개론 교재 7장 3절

이전 교재 4장 3절

4.1 고급 언어 번역: 컴파일러

4.2 컴파일 과정

4.1 고급 언어 번역과 컴파일러

언어의 변화

1세대 언어 : 기계어

2세대 언어 : 어셈블리어

3세대 언어 : 고급 프로그래밍 언어

기계어(machine language) : 2진수 코드 명령어, 컴퓨터 직접 실행

어셈블러 : 어셈블리어는 기호로 이루어진 명령어,

: 어셈블리어를 기계어(**목적코드**)로 번역하는 시스템 소프트웨어

컴파일러 : 고급 프로그래밍 언어를 어셈블리어 (또는 기계어)로 번역하는 시스템 소프트웨어

언어의 변화

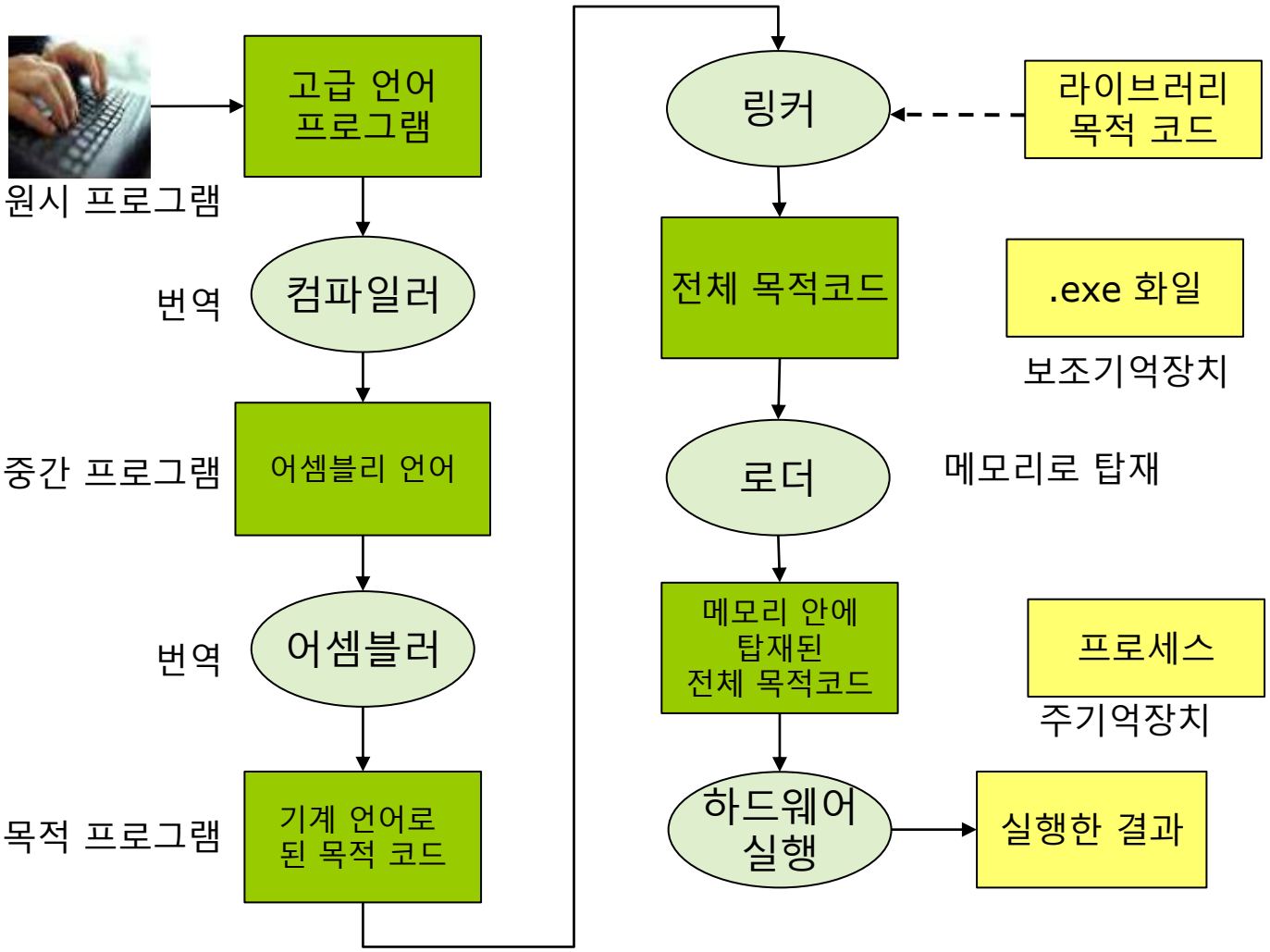


그림 3.1 컴파일러의 고급 언어 프로그램 번역 과정과 실행

프로그래밍 언어의 유형

고급 프로그래밍 언어 :

문법 구조(syntax)와 **문장의 의미**(semantic)

- syntax : 문장을 정확하게 작성하도록 하는 정해진 규칙
- semantic: 프로그램에 작성된 문장의 동작 구조

아버지 가방에 들어 가신다.

- 문법 맞음 - 주어 목적어 동사
- 의미 틀림

고급 언어 번역과 컴파일러

어셈블리어와 기계어는 1:1 대응 관계

어셈블리어 인스트럭션은 정확하게 한 개의 기계어 인스트럭션을 생성

어셈블러가 테이블에서 대응하는 기계어 인스트럭션을 찾아서 대치

ADD → 명령어 테이블에서 찾은 0101로

피연산자의 위치 : 심볼테이블

- 주소 X → 000010
- 주소 A → 010111

결국 어셈블리어 기계어

- ADD X, A → **0101**000010010111

고급 언어의 문장을 예를 들어 보자

고급언어 (source code)

a = b + c – d;

어셈블리어

LOD B

ADD C

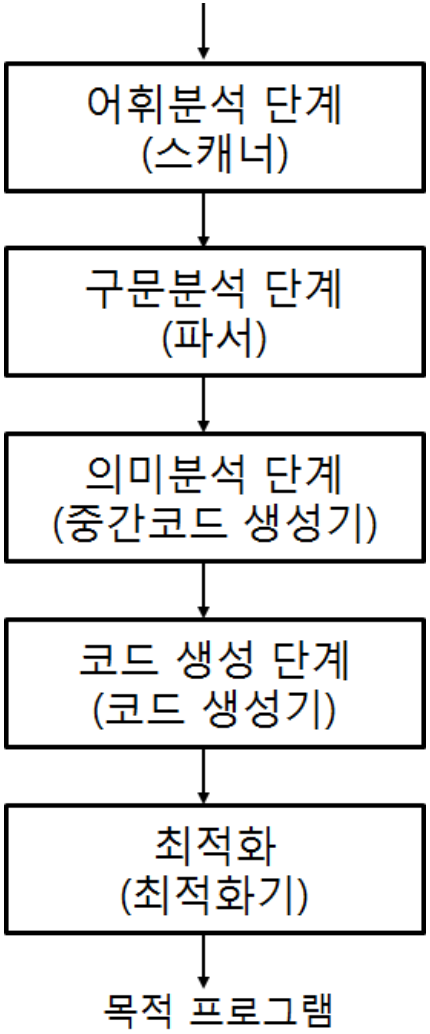
SUB D

STO A

기계어 (target code)

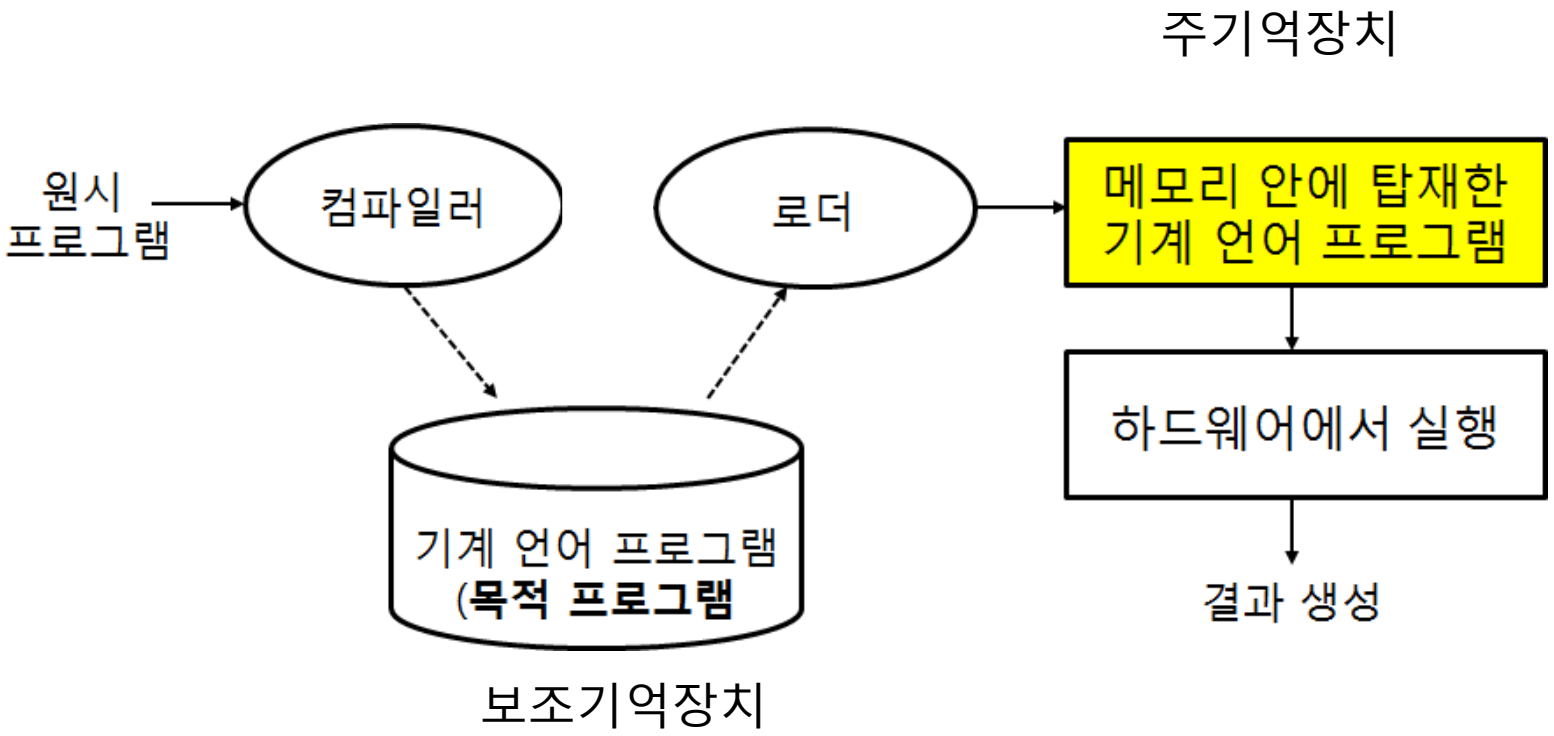
4.2 컴파일 과정

컴파일 단계



❙그림 4.1❙ 간략한 컴파일러 단계

목적 프로그램



❙그림 4.2❙ 고급 언어 프로그램이 실행되는 과정

4.2.1 어휘 분석

어휘 분석기(lexical analysis)

regular grammar에 따라 토큰으로 분류

문자열을 토큰이라는 단위로 쪼갠다.

토큰은 더 이상 쪼갤 수 없는 한 개의 단어(어휘)

(예시) area = b + 3.14 * radius;

문자열 area, b, radius, ;

숫자 3.14이며,

연산자는 =, +, *, ;

이 한 문장에 아래와 같이 14개의 토큰이 존재한다.

- **area, 빈칸, =, 빈칸, b, 빈칸, +, 빈칸, 3.14, 빈칸, *, 빈칸, radius, ;**

if (x == y) area = 3.14 * radius * radius ;

입력된 원시 C 프로그래밍 언어 문장 분석 결과

토큰	유형
if	심볼
(	괄호 열기 연산자
x	심볼
==	관계 비교 연산자
y	심볼
)	괄호 닫기 연산자
area	심볼
=	할당 연산자
3.14	상수
*	곱하기 연산자
radius	심볼
*	곱하기 연산자
radius	심볼
;	문장 끝 심볼

regular grammar – 어휘 정의

BNF 문법

<number> ::= <digit> | <digit> <number>

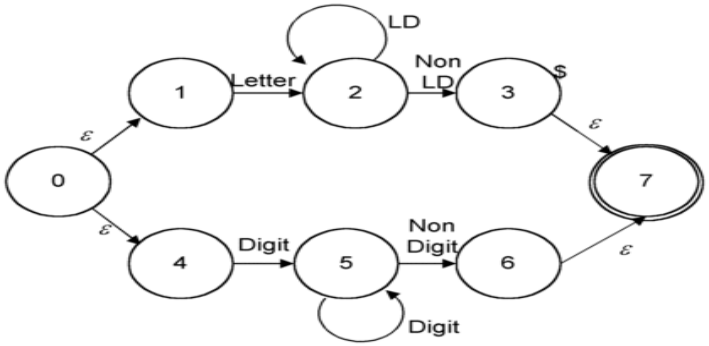
<identifier> ::= <letter> | <variable> <letter> | <variable> <digit>

<letter> ::= "a" | "b" | ... | "z" | "A" | "B" | ... | "Z" | "_"

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9“

<operator> ::= “+”|”-”|”*”|”/”|”;”|”++”|”—”|……

state transition diagram(상태천이도)



regular expression

number : ^[0-9]+$

identifier : ^[a-zA-Z_][a-zA-Z0-9_]*$

4.2.2 syntax analysis(구문 분석) 단계

파서(parser)는 어휘 분석 단계에서 토큰으로 분해된 문장이 문법에 맞는지 판단

프로그래밍 언어의 문법에 맞게 프로그램을 작성하였나 판단

이 단계에서 수행하는 과정을 아래 예로 살펴보자.

문법

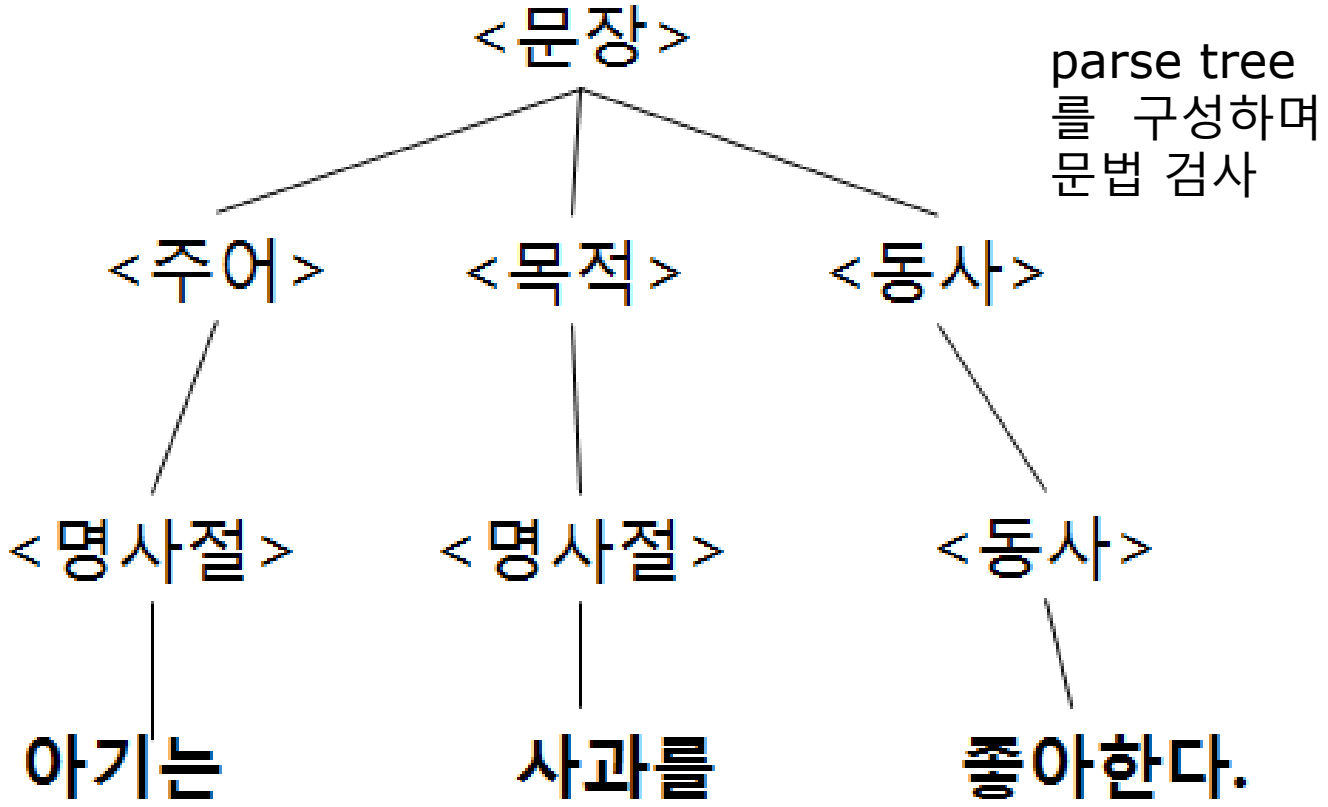
- 문장::= <주어> <목적어><동사>

작성 문장

아기는 사과를 좋아한다.

4.2.2.1 파싱

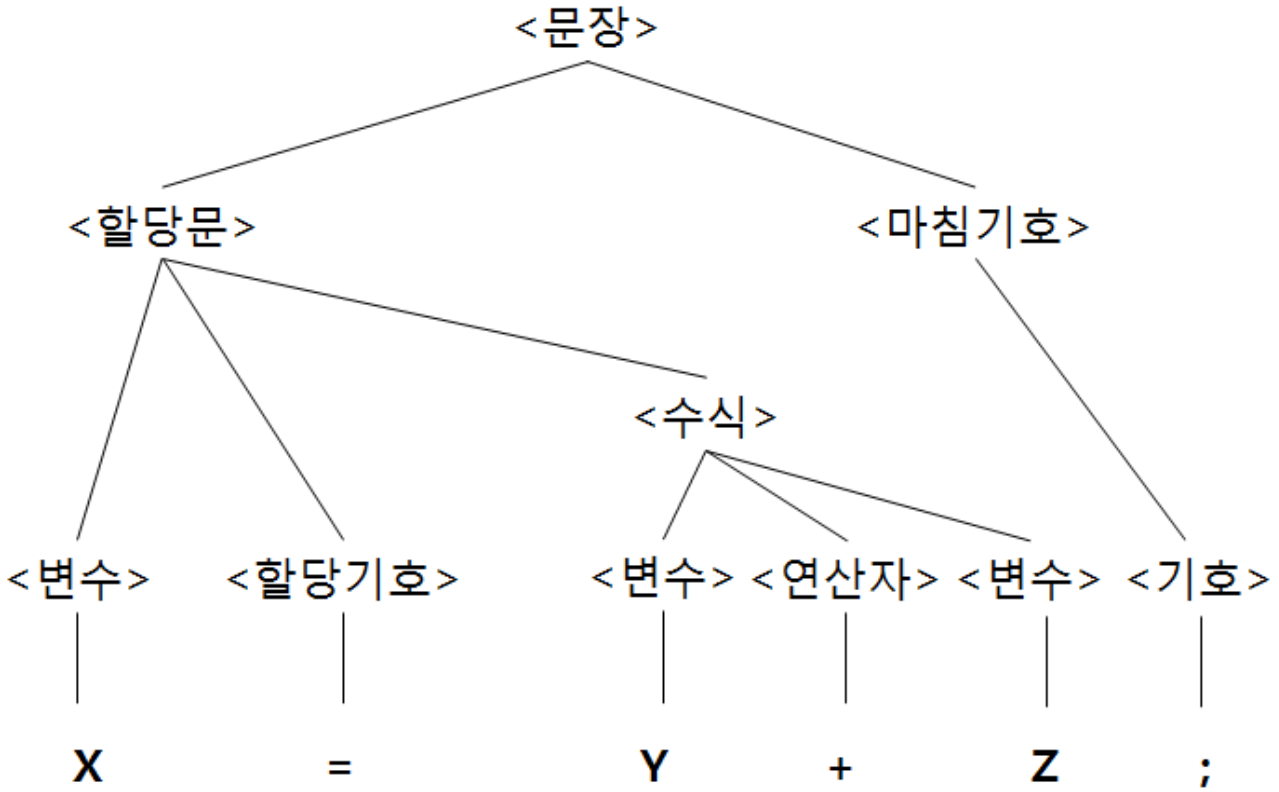
작성 문장 : 아기는 사과를 좋아한다.



parse tree
를 구성하며
문법 검사

고급 프로그래밍 언어 X = Y + Z;

어떤 문장이 문법에 맞는지를 검사하기 위해 문법을 대조하는 구조를 **파스 트리**라 한다.



4.2.2.2 문법과 언어

syntax : 프로그래밍 언어 문법(구문 구조:syntax)

syntax의 수학적 표현 형식

BNF(Backus–Naur form) 표기법

파서는 토큰으로 분해된 문장이 문법에 맞는지 판단

문법 : 구문 구조 규칙의 집합(규칙을 모아놓은 것)

왼쪽 ::= “정의”

BNF(John Backus와 Peter Naur) Form

<할당문> ::= <심볼> = <수식>;

<할당문>가 기호 ::= 의 오른쪽에 있는 <심볼> = <수식>;를 정의

<심볼>,할당 기호(연산자) ‘=‘, <수식>, ‘;’ 순서로 나온다.

할당문 예

<assignment> ::= <identifier> "=" <expression> ";"

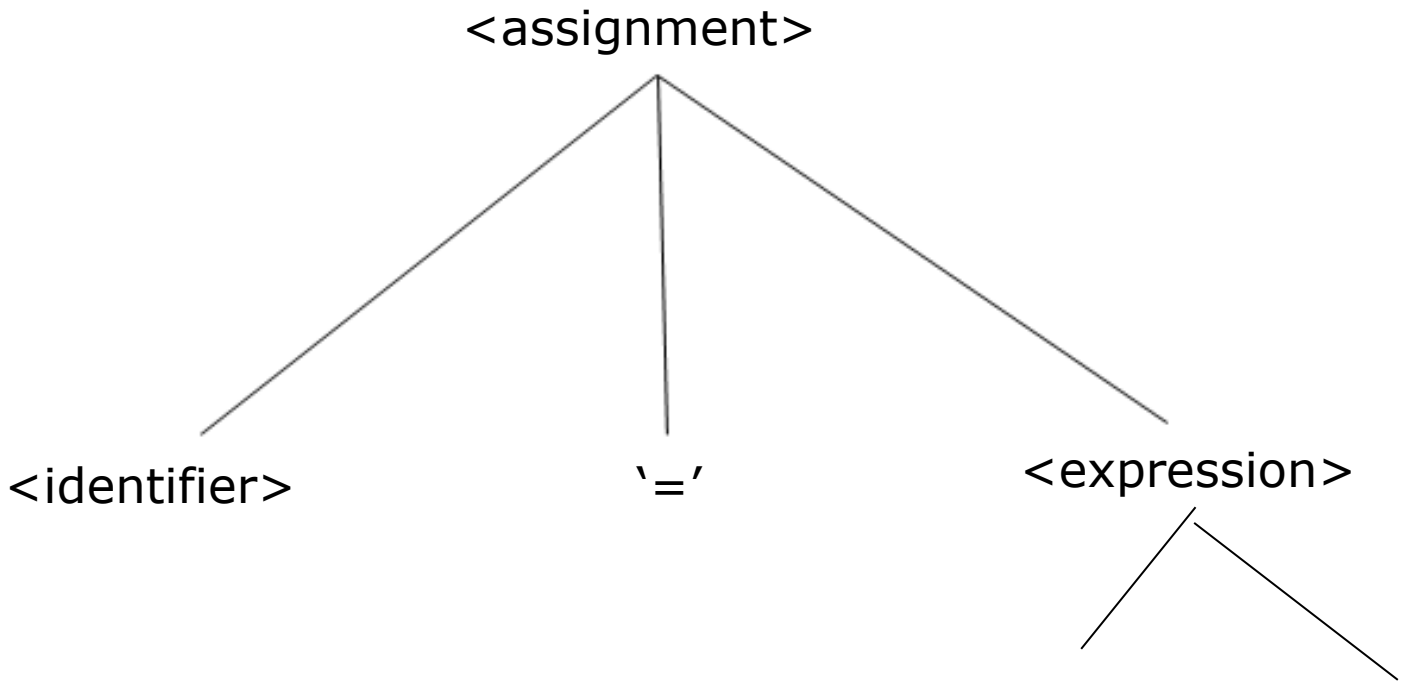
<expression> ::= <factor> | <factor> <operator> <expression>

<factor> ::= <number> | <identifier> | "(" <expression> ")"

<operator> ::= "+" | "-" | "*" | "/"

문법 – tree 관계

<할당문> ::= <심볼> = <수식>;



생성 혹은 생성규칙

BNF 규칙: **생성 규칙**

터미널 개체 : 어휘 분석단계에서 토큰으로 분류된 것

넌터미널 개체

- 터미널과 넌터미널 개체 모두 생성 규칙의 오른쪽에 표기 가능
- 터미널은 오른 쪽에만
- <non terminal> 개체, “터미널 개체“

할당문 예제

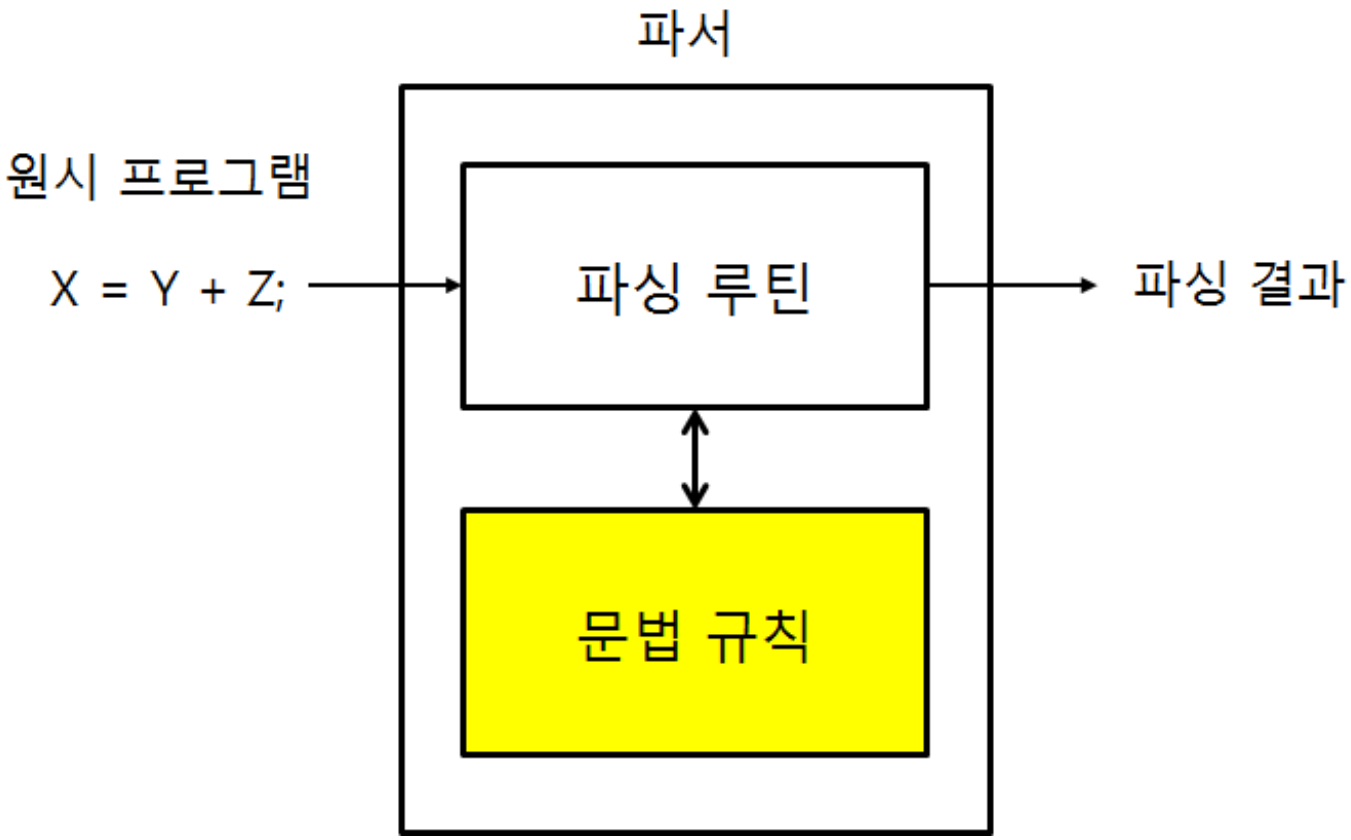
<assignment> ::= <identifier> "=" <expression> ";"

<expression> ::= <factor> | <factor> <operator> <expression>

<factor> ::= <number> | <identifier> | "(" <expression> ")"

<operator> ::= "+" | "-" | "*" | "/"

4.2.2.3 파싱 개념과 기법



❙그림 4.3❙ 파서의 구조와 작동 개념

예

연속 번호	생성 규칙
1	<할당문> ::= <변수> = <수식> ;
2	<수식> ::= <수식> + <수식>
3	<변수> ::= a|b|c|x|y|z

- 변수도 단지, a, b, c, x, y, z만 사용할 수 있다.

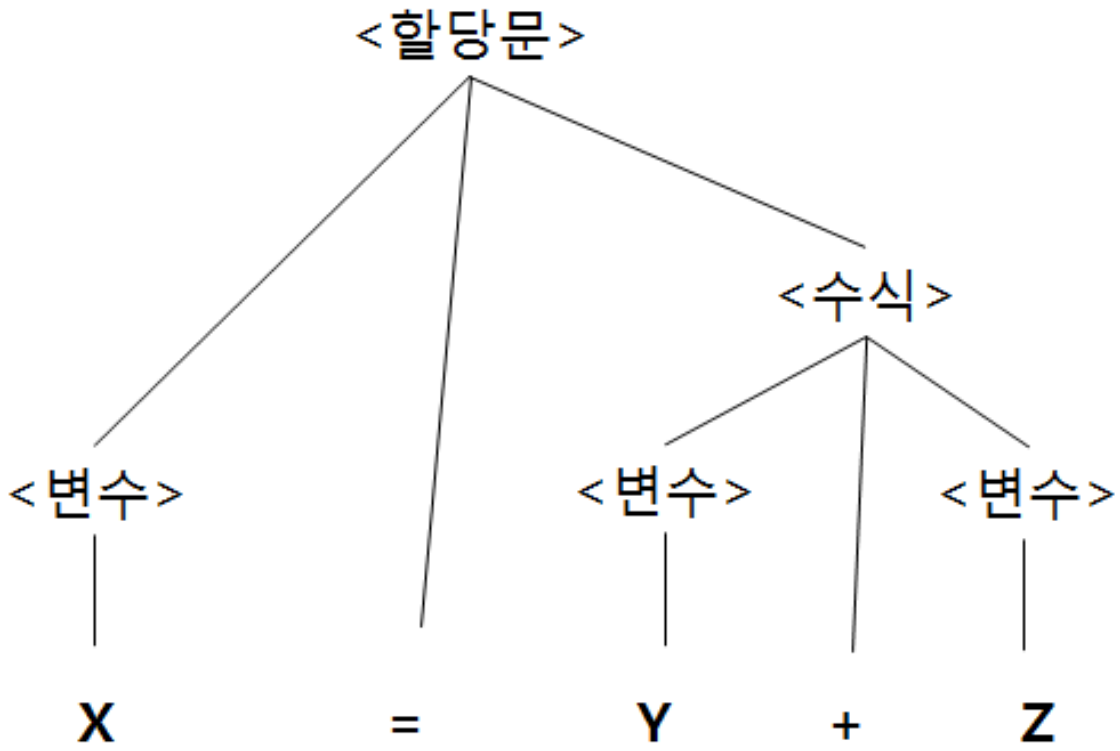
<assignment> ::= <identifier> "=" <expression> ";"

<expression> ::= <factor> | <factor> <operator> <expression>

<factor> ::= <number> | <identifier> | "(" <expression> ")"

<operator> ::= "+" | "-" | "*" | "/"

<수식> ::= <수식> + <수식>



❙그림 4.5❙ 파서에 의해 생성된 파스 트리 – 문법 검사

parsing - 문법체크

<assignment> ::= <identifier> "=" <expression> ";"

<expression> ::= <factor> | <factor> <operator> <expression>

<factor> ::= <number> | <identifier> | "(" <expression> ")"

<operator> ::= "+" | "-" | "*" | "/“

```
assignment() {
        match(token,identifier())
        match( operator, ‘=‘);
        expression();
}
```

```
expression() {
        factor()
        if(match(tok,operator()),
        expression()) return TRUE
        return FAlse
}
```

```
factor() {
        if( match(tok,number) or match(tok, identifier) )  return TRUE
        else {match(‘(‘), expression(), match(‘)’) } return TRUE
        else return FALSE
}
```

4.2.3 의미 분석과 코드 생성

각 변수 x, y는 컴퓨터 내부에 **실수형** 값을 저장할 수 있도록 메모리 공간을 확보하라는 **의미??**

y = x * y;

실수형 자료가 저장된 공간 x 주소의 값과 실수형 자료가 저장된 공간 y 주소의 값을 곱하여, 실수형 자료가 저장될 수 있는 공간 y 주소에 결과

float x, y;

아래 프로그램 일부를 살펴보자.

a = 13; b = 40; s = 0; s = a + b;

의미 분석 단계를 통하여 기계어 코드를 생성

⋮

load a, r1 // 1번 레지스터 r1에 a 번지의 정수형 값을 탑재(저장)하라 //

load b, r2 // 2번 레지스터 r2에 b 번지의 정수형 값을 탑재(저장)하라 //

add r1, r2 // 레지스터 r1과 r2의 값을 더하여 r1에 결과를 저장하라 //

store r1, s // r1 레지스터의 내용(값)을 메모리의 정수형 s 번지에 저장//

⋮

```
a: .data 13          // a = 13;     //
b: .data 40          // b = 40;     //
s: .data 0           // s = 0;      //
```

아래 프로그램 일부를 살펴보자.

a = 13; b = 40; s = 0; s = a + b;

```
assignment() {
      id()
      “=“
      expression()
}
expression()  {
      factor();
      (tok,operator)
      expression();
      }
```

```
factor() {
      (tok, identifier) or
      (tok, number) or
      (tok,’(‘)), expression(), (tok,’)’)
```

4.2.4 코드 최적화

문장 a = b + b + b;는 곱하기(*) 연산 대신 더하기 연산 세 번을 수행

실행 시간을 단축 코드로 최적화

실행 시간 단축 최적화

tail recursive 함수를 반복문으로 변경 등

```
load    b, r1
add     r1, b
add     r1, b
add     r1, b
store   r1, a
   ⋮
```