4장: 시스템 소프트웨어

4.1 가상기계

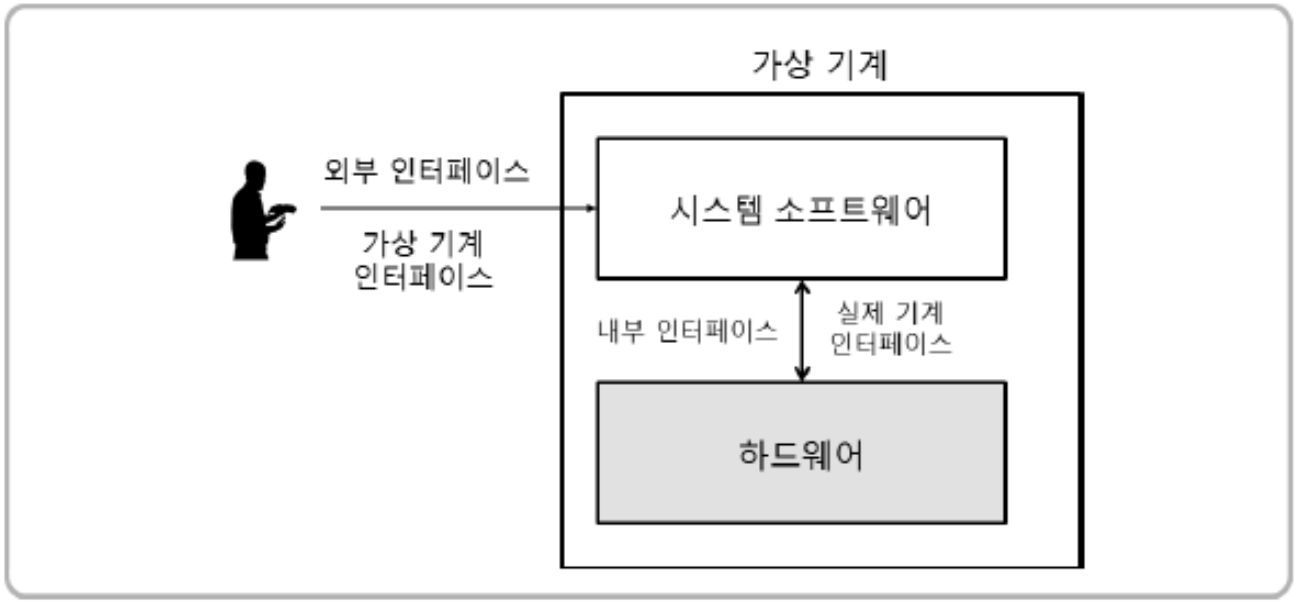
4.2 시스템 소프트웨어

4.3 컴파일러

4.1 가상기계

시스템 소프트웨어가 만드는 가상기계

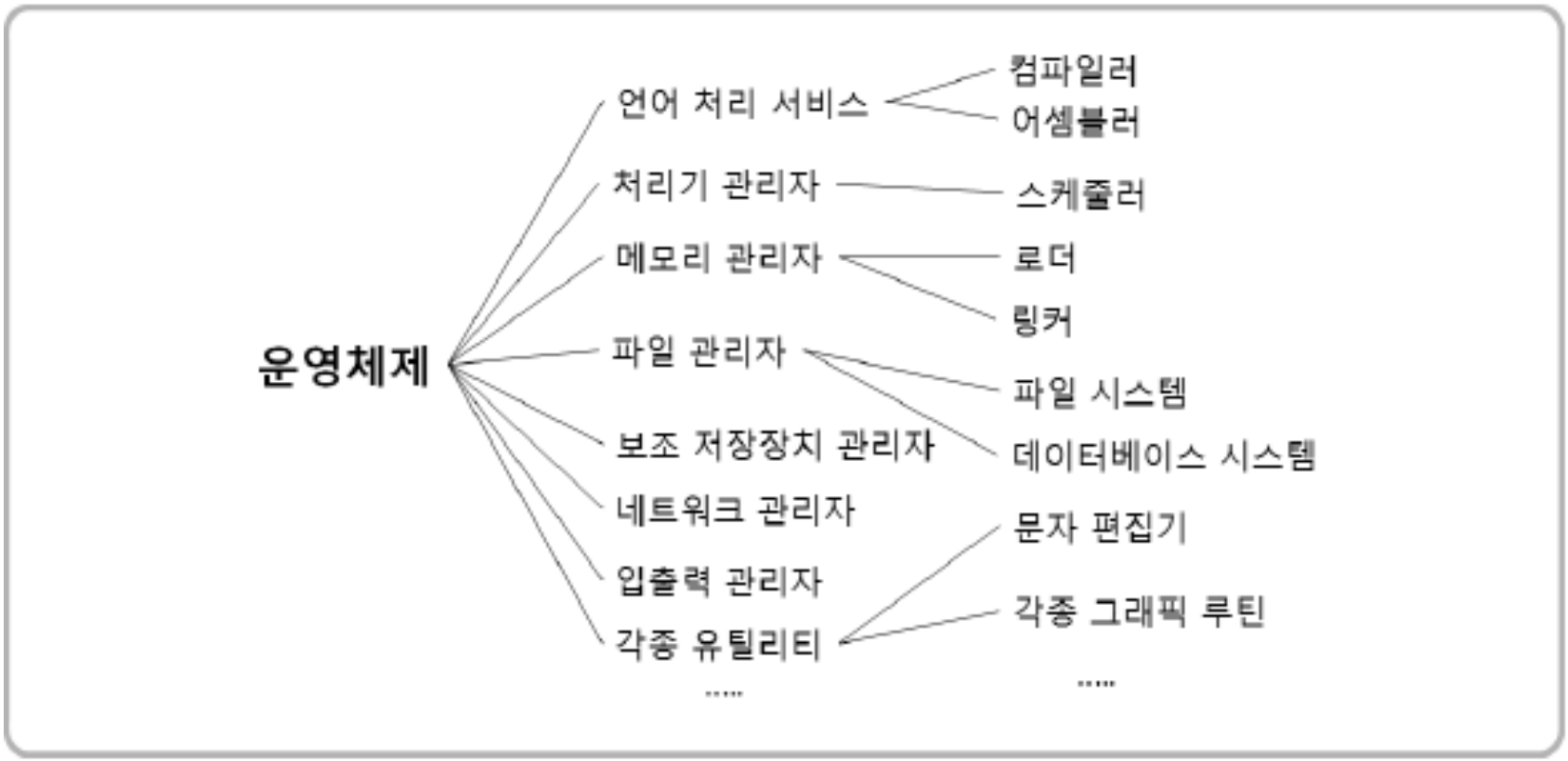
컴퓨터 HW 내부구조를 잘 몰라도 필요한 자원에 쉽게 접근하고, 작업을 안전하게 수행하며, 필요한 정보를 알 수 있다.



[그림 4.1] 시스템 소프트웨어 역할

4.2 시스템 소프트웨어

시스템 소프트웨어



[그림 4.2] 시스템 소프트웨어의 종류

4.3 컴파일러

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

- 주소 X → 000010
- 주소 A → 010111

결국 어셈블리어 기계어

- ADD X, A → **0101**000010010111

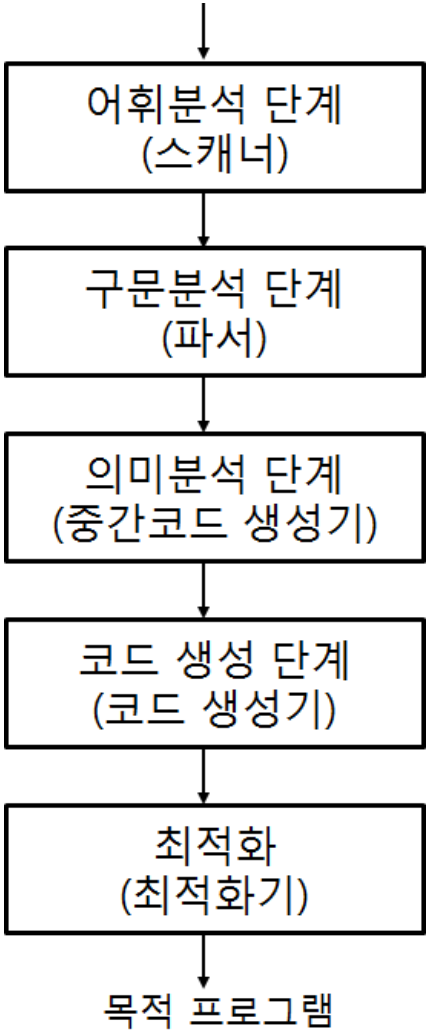
고급 언어의 문장을 예를 들어 보자

a = b + c – d;

어셈블리어

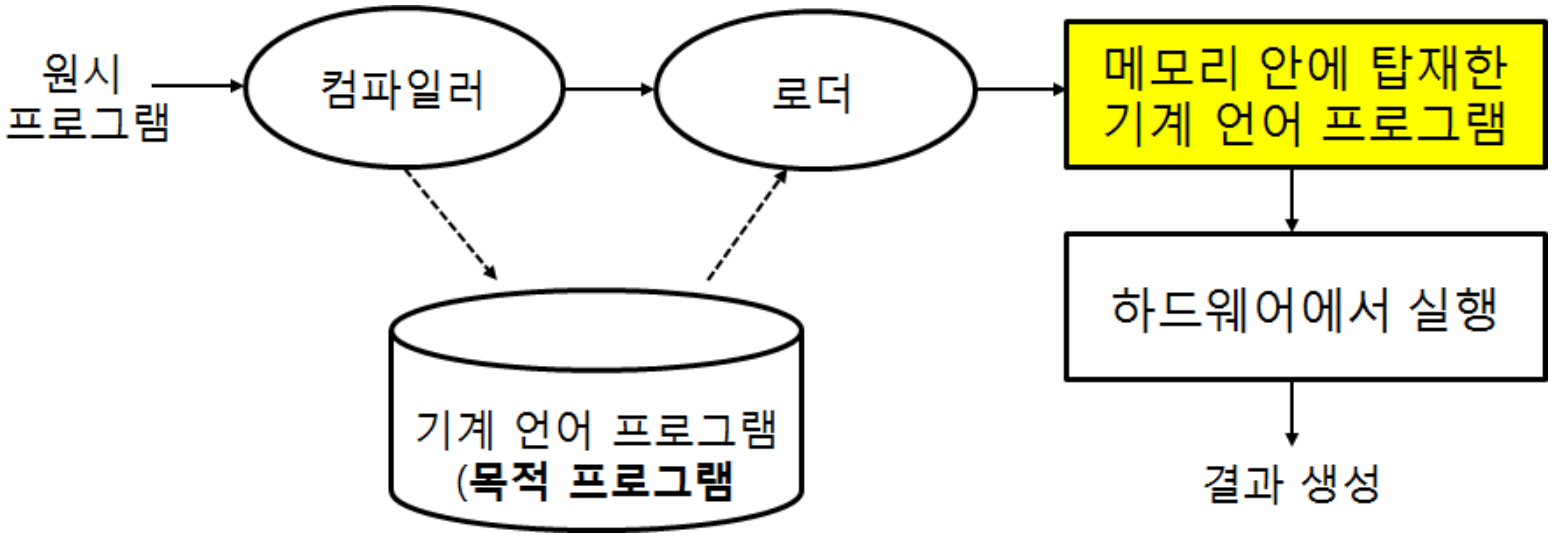
```
LOD  B
ADD  C
SUB  D
STO  A
```

컴파일 단계



❙그림 4.1❙ 간략한 컴파일러 단계

목적 프로그램



❙그림 4.2❙ 고급 언어 프로그램이 실행되는 과정

4.2.1 어휘 분석(단어분석)

어휘 분석을 수행하는 모듈을 **어휘 분석기**(혹은 **스캐너**)라 하는데, 입력된 문자열을 토큰이라는 단위로 쪼갠다. 즉, 토큰은 구조적인 단위로 번역되어 질 때 더 이상 쪼갤 수 없는 한 개의 단어(어휘)다. 예를 들어, 아래의 할당문을 살펴보자.

area = b + 3.14 * radius; lexical analyze.

이 할당문에 사용한 심볼인문자열이나 문자는 area, b, radius, ;이고, 숫자는 상수로 3.14이며, 연산자는 =, +, *이다. 그러나 실제로는 이 한 문장에 아래와 같이 22개의 문자가 존재한다.

a, 빈칸, =, 빈칸, b, 빈칸, +, 빈칸, 3, ., 1, 4, 빈칸, *, 빈칸, r, a, d, i, u, s, ;

어휘정의

Regular expression 을 이용한 definition

$$letter\_ \longrightarrow A \mid B \mid C \mid \ldots \mid Z \mid a \mid b \mid c \mid \ldots \mid z \mid \_$$

$$digit \longrightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

$$id \longrightarrow letter\_ \ (letter\_ \mid digit)^*$$

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

입력된 원시 C 프로그래밍 언어 문장

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

4.2.2 구문 분석 단계

parse.

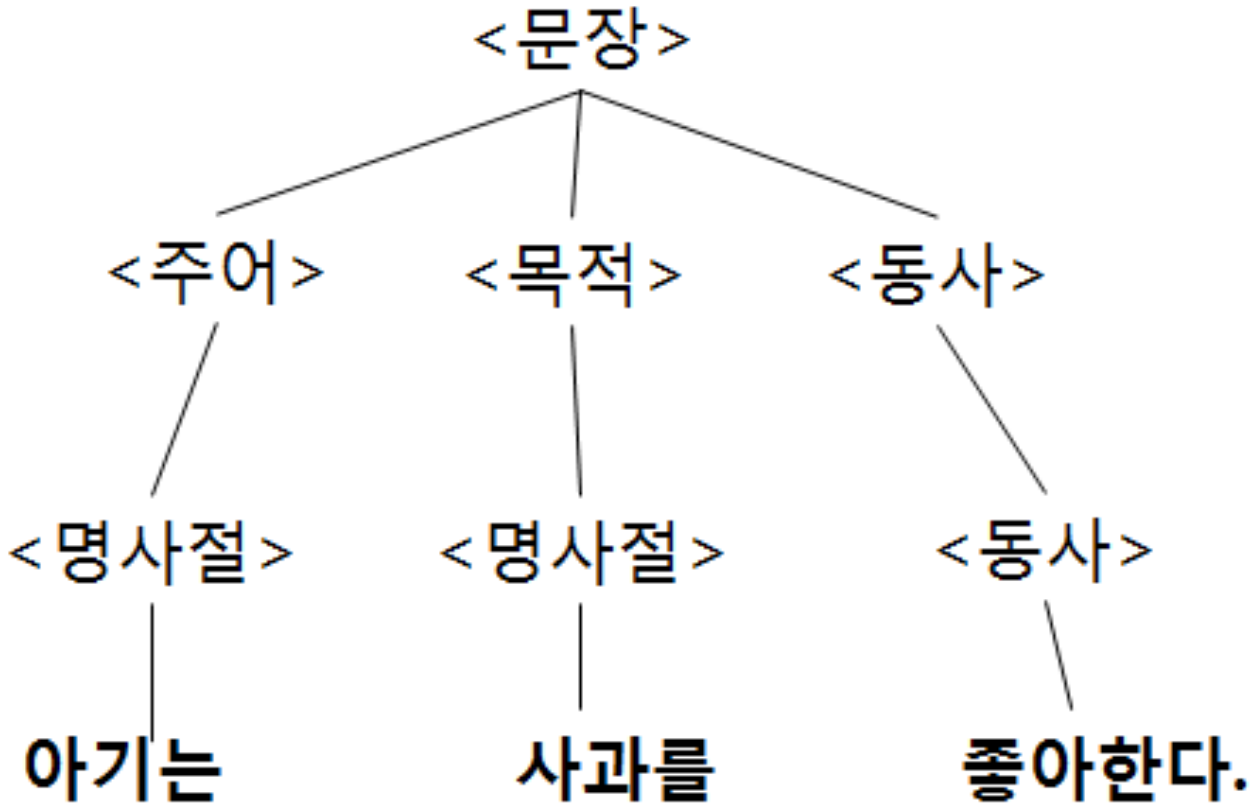
단계에서는 앞 단계의 어휘 분석 단계서 생성한 토큰에 대하여 문법적으로 맞는지 아닌지를 분석하고 검사한다. 즉, 고급 프로그래밍 언어에서 제공하는 문법 규칙에 따라 프로그램 문장이 잘 작성되었는지를 파악하고 문법적으로 옳은지 아닌지를 결정한다.

이 단계에서 수행하는 과정을 아래 예로 살펴보자.

아기는 사과를 좋아한다.

4.2.2.1 파싱(parsing)

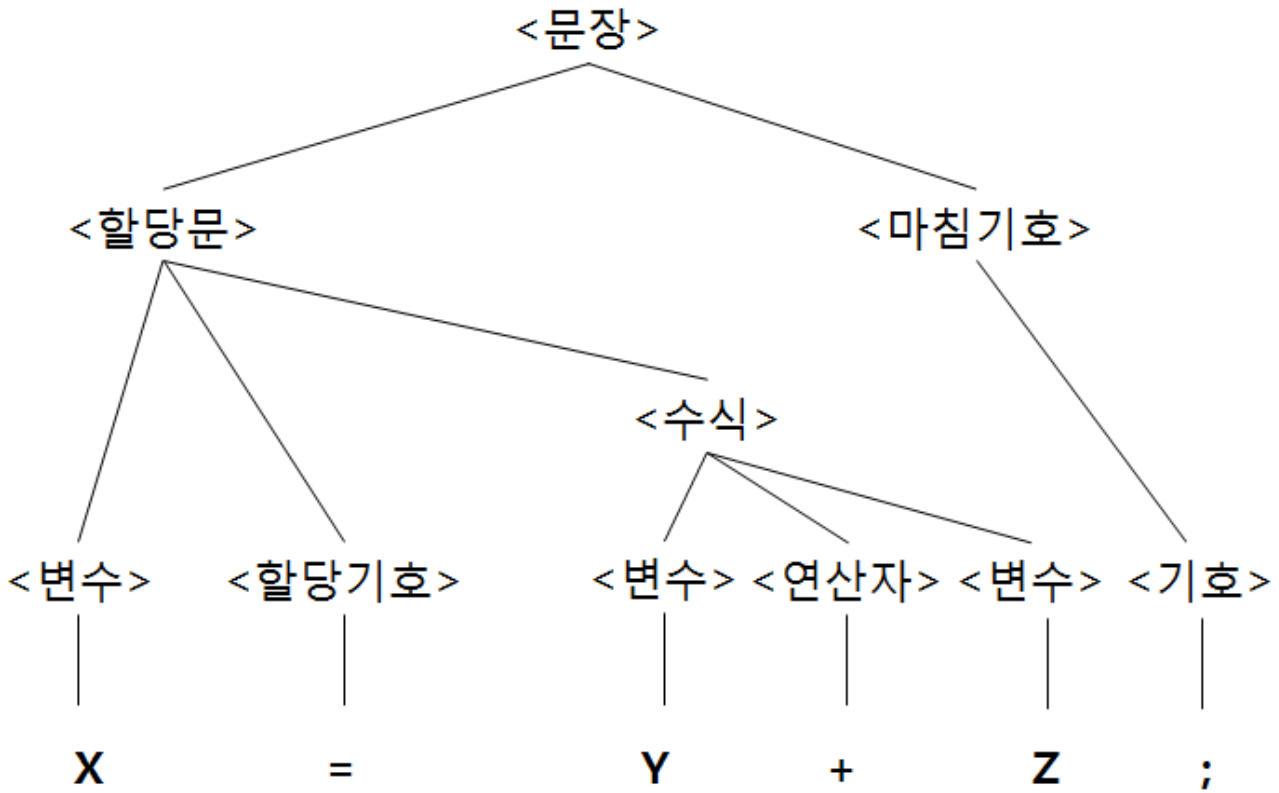
아기는 사과를 좋아한다.



고급 프로그래밍 언어 X = Y + Z;

이 문장은 아래와 같이 문법적으로 맞는다고 판단한다.

이와 같이 어떤 문장이 문법에 맞는지를 검사하기 위해 문법을 대조하는 구조를 **파스 트리(parse tree)**라 한다.



4.2.2.2 문법과 언어

일단 파서는 프로그래밍 언어의 **구문 구조**(신택스)라는 문법을 수학적으로 표현하는 형식을 가지고 있어야 한다. 그러면 파서는 이 문법으로 토큰으로 분해된 문장을 분석하게 된다. 문법을 표현하는 방법은 다양한데 그 중에서 한 방법이 **BNF** 표기법이다.

BNF에서는 프로그래밍 언어의 구문 구조를 규칙의 집합으로 표기하며 **생성**이라 한다. 이렇게 특정 프로그래밍 언어에 대해 문법 규칙을 모아 놓은 것을 **문법**이라 한다. 이 문법에서 BNF 규칙 한 개는 아래처럼 표기한다.

왼쪽 ::= “정의”

BNF(John Backus와 Peter Naur) Form

<할당문> ::= <심볼> <할당기호> <수식>;

<할당문>가 기호 ::= 의 오른쪽에 있는 <심볼> = <수식>;를 정의
이 때 <심볼> 바로 다음에 할당 심볼인 =가 따라 나오고,
그 다음에는 <수식>와 ;가 나타난다.

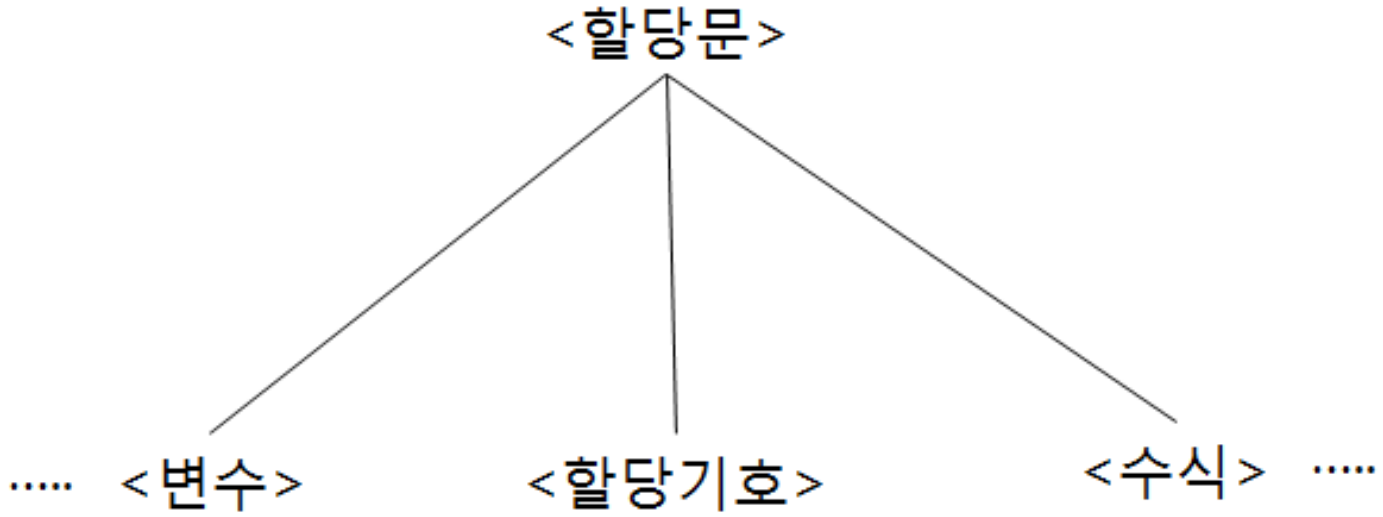
<더하기 연산자> ::= +

문법 구조 <더하기 연산자>는 연산자가 한 개인 +를 정의

할당문

<할당문> ::= <심볼><할당기호><수식>;

<할당기호> ::= =



<할당문> ::= <변수> = <수식>

<수식> ::= <변수> + <변수>

<변수> ::= a | b | c | X | Y | Z

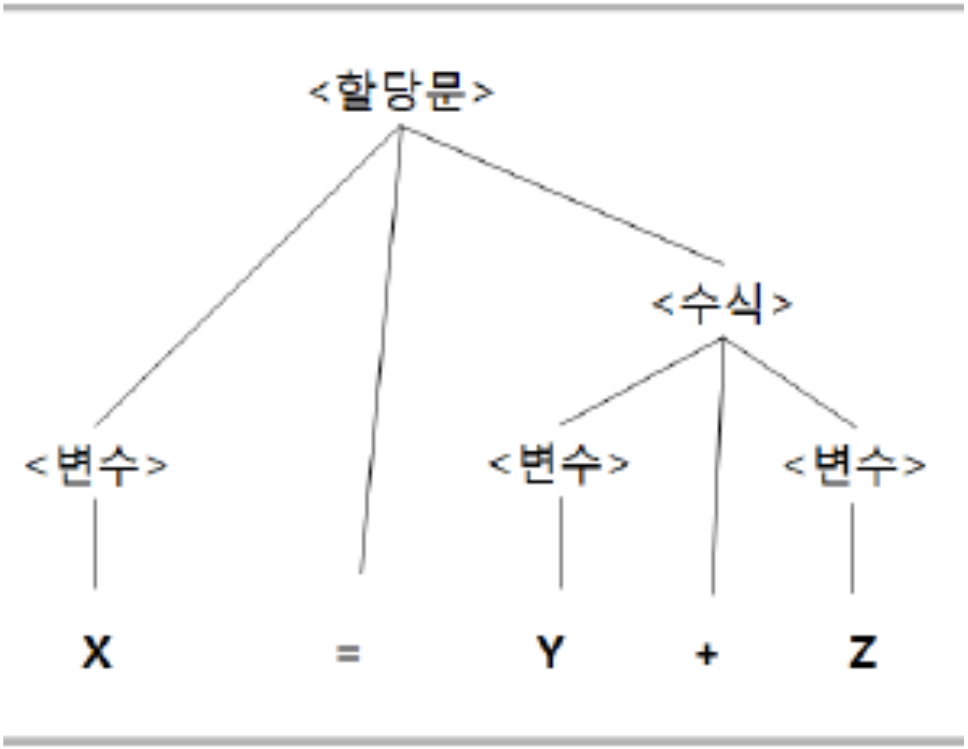
X = Y + Z

⇒ X = <변수> + <변수>

⇒ <변수> = <변수> + <변수>

⇒ <변수> = <수식>

⇒ <할당문>



[그림 4.6] 파서에 의해 생성된 파스 트리

생성 규칙

<문장> ::= <할당문> <마침기호>

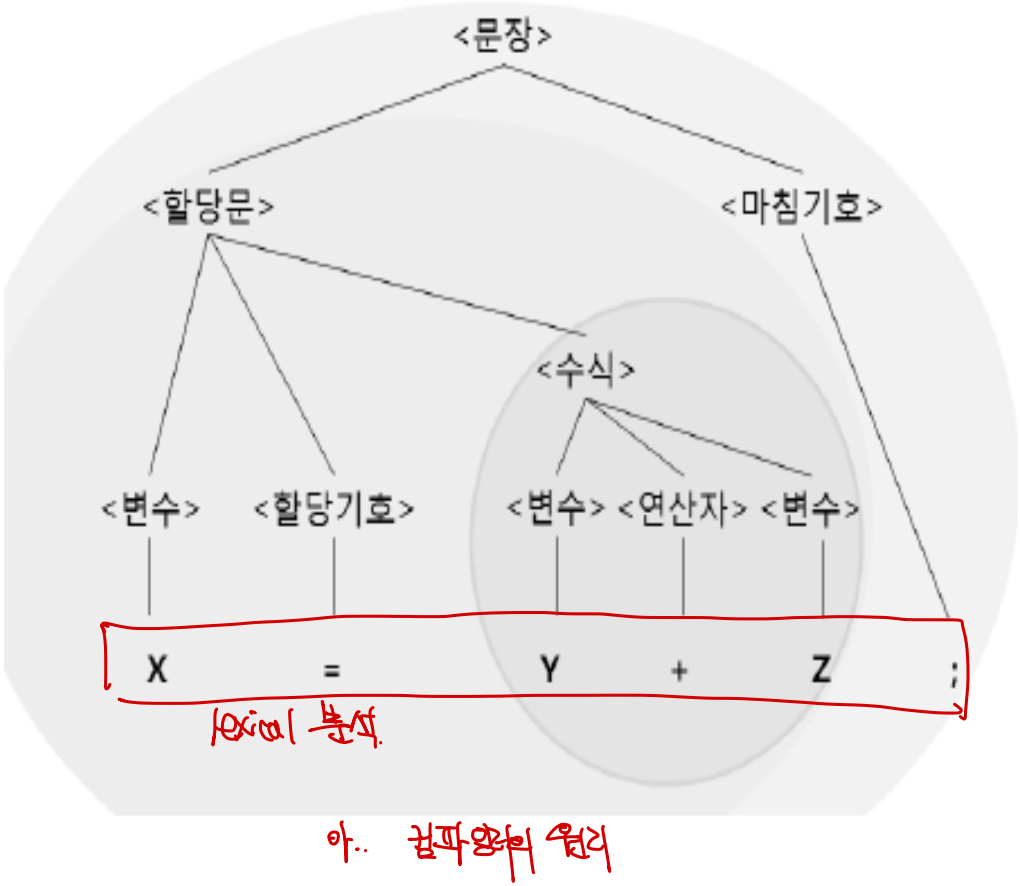
<할당문> ::= <변수> = <수식>

<수식> ::= <변수> + <변수>

<변수> ::= a | b | c | X | Y | Z

<마침기호> ::= ;

X = Y + Z;



생성 혹은 생성규칙

<심볼>, =, <수식>, ;는 <할당문>이라는 문법 영역에서 생성

BNF 규칙이 **생성** 혹은 **생성 규칙**이기 때문

터미널 개체

넌터미널 개체

- 터미널과 넌터미널 개체 모두 생성 규칙의 오른쪽에 표기가능
- 터미널은 오른 쪽에만

터미널 예

- if else area x y 3.14 = + * == ( ) ;

<수> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

만약 수를 정의하는 넌터미널인 <digit>를 I 없이 정의

<digit> ::= 0

<digit> ::= 1

<digit> ::= 2

<digit> ::= 3

<digit> ::= 4

<digit> ::= 5

<digit> ::= 6

<digit> ::= 7

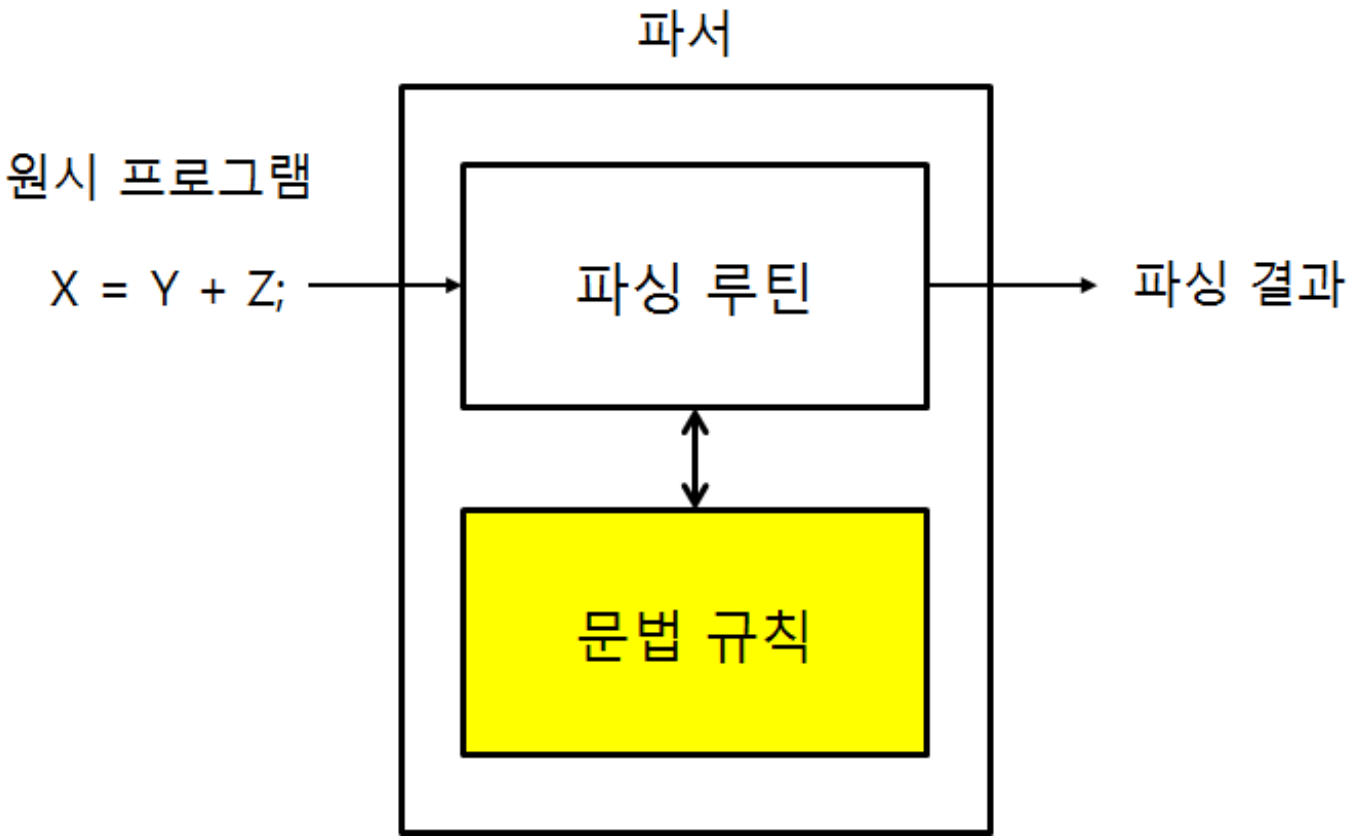
<digit> ::= 8

<digit> ::= 9

P.116 숫자설명한 부분 삭제 7-16까지 삭제

이유 어휘분석에서 숫자로 인식하여 구문 분석단계로 넘어옮

4.2.2.3 파싱 개념과 기법



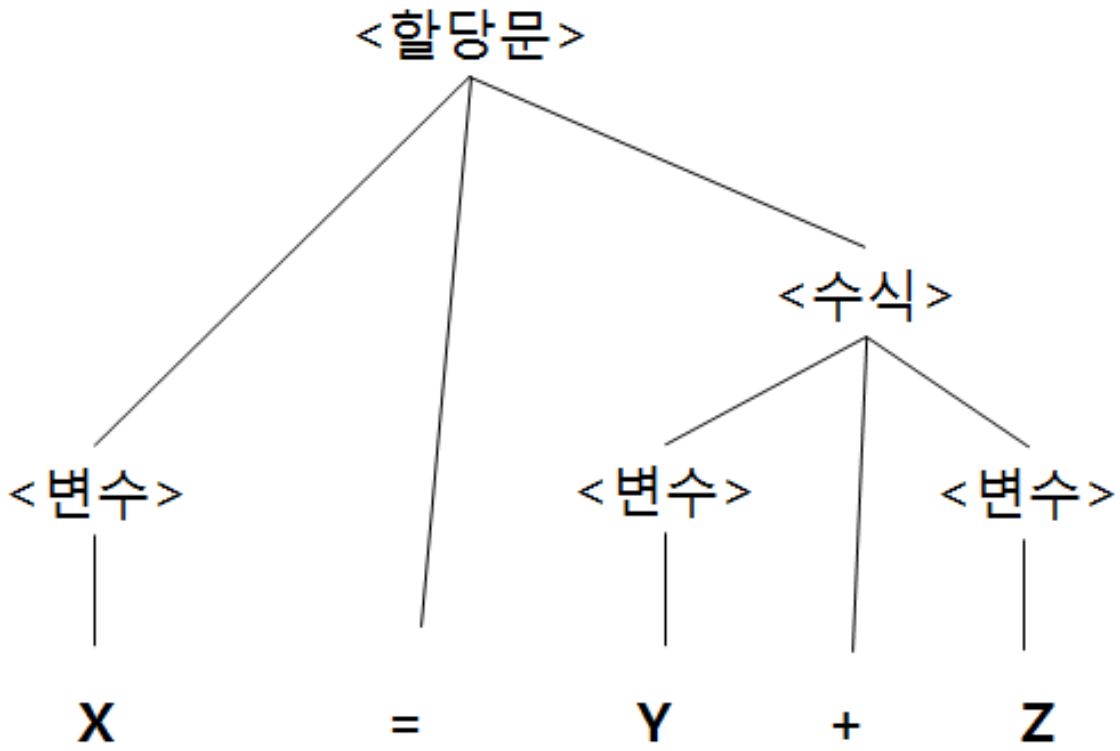
❙그림 4.3❙ 파서의 구조와 작동 개념

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.

<수식> ::= <수식> + <수식>



❙그림 4.5❙ 파서에 의해 생성된 파스 트리

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;      //
```

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

a = b × 3;

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

4.4 기계어프로그램의 실행

기계어프로그램

메모리 주소	프로그램(32비트)
00 ···· 0000100	10101010101010111100010101010101
00 ···· 0001000	00000000111111111100001111100000
	.....
00 ···· 0001100	11101010111111111000000000111111
00 ···· 0010000	00010101111100000111111101010101
	.....
00 ···· 0011010	11111000000101011111111101010110

어셈블리어 프로그램

```
      ADD    a
      JMP    go
      ....
go:   ADD    b
      ....
a:    DATA   1
b:    DATA   20
```

어셈블리 명령어의 일부 예시

기계어	어셈블리어		의미(M(X):메모리의 주소 X)
0000	LOD	X	R ← M(X)
0001	STO	X	M(X) ← R
0010	CLR	X	M(X) ← 0
0011	ADD	X	R ← R + M(X)
0100	INC	X	M(X) ← M(X) + 1
0101	SUB	X	R ← R − M(X)
0110	DEC	X	M(X) ← M(X) − 1
0111	COM	X	만약 M(X) > R GT = 1, 아니면 0
			만약 M(X) = R EQ = 1, 아니면 0
			만약 M(X) < R LT = 1, 아니면 0
1000	JMP	X	PC ← X, 다음 실행 순서는 X번지
1001	JGT	X	GT = 1이면, 메모리 위치 X부터 실행
1010	JEQ	X	EQ = 1이면, 메모리 위치 X부터 실행
1011	JLT	X	LT = 1이면, 메모리 위치 X부터 실행
1100	JNQ	X	EQ = 0이면, 메모리 위치 X부터 실행
1101	INP	X	정수형 값을 입력받아 메모리 위치 X에 저장
1110	OUT	X	메모리 위치 X에 저장된 값을 십진 값으로 출력
1111	HAL		프로그램 실행을 종료

어셈블리 프로그램

```
1  TOTAL ← 0
2  COUNT ← 0
3  read  data
4  while data ≠ 0  do
5        TOTAL ← TOTAL + 1
6        COUNT ← COUNT + data
7        read  data
8  endwhile
9  write COUNT, TOTAL
10 stop
```

```
        BEGIN
LOOP:   INP     data      //정수 값을 읽어서 data에 저장한다.
        LOD     TOTAL     //레지스터 R에 TOTAL 값 0을 넣어라.
        ADD     data      //레지스터 R 값과 data의 값을 더하여 R에 넣어라.
        STO     TOTAL     //레지스터 R 값을 TOTAL에 저장하라.
        LOD     COUNT     //레지스터 R에 COUNT 값 0을 넣어라.
        ADD     1         //레지스터 R 값과 1을 더하여 R에 넣어라.
        STO     COUNT     //레지스터 R 값을 COUNT에 저장하라.
        LOD     ZERO
        COM     data      //레지스터 R의 0과 읽은 data를 비교한다.
                JEQ     PRINT     //같으면(0이면) PRINT로 가서 실행한다.
                JMP     LOOP      //레이블 LOOP로 가서 I 값을 다시 읽는다.
PRINT:          OUT     COUNT     //COUNT를 출력한다.
                OUT     TOTAL     //TOTAL을 출력한다.
STOP:           HAL               //프로그램 정지한다.
   I:           DATA    0         //입력 정수 값 I를 위한 저장 공간
COUNT:          DATA    0         //횟수 COUNT를 저장하기 위한 공간
TOTAL:          DATA    0         //합계 TOTAL을 저장하기 위한 공간
ZERO:           DATA    0
                END
```