제 3 장

구문과 의미론

제 3 장 구문과 의미론

❖ 목적 :

- 구문(syntax)과 의미(semantics)
- 언어의 문법적 요소
- 형식 언어
- 형식 문법
- 문법의 표현 방법
- 유도, 파스 트리, 모호성, 연산자 우선순위
- 속성 문법(attribute grammar)
- 의미론

3.1 서론(구문과 의미)

- **언어 설계자, 언어 구현자, 언어 사용자**
  - **언어는 만드는 방법** – 문법을 디자인(문법 안에는 구문과 의미 포함), 언어를 구현, 언어를 사용.

- **구문(syntax) –** 문장에 있는 원소들의 관계를 보여주기 위한 것으로 언어의 표현식, 문장, 프로그램 단위에 대한 형식이나 구조
  - **구문의 목적 :** 프로그래머와 프로그래밍 언어 프로세서 사이에 의사소통을 하기 위한 notation
- **의미(semantics) -** 언어의 표현식, 문장, 프로그램 단위의 의미
  - 예) C의 **if** 문:
  - 구문: **if** (<식>) <문장>
  - 의미: 식에 대한 조건을 만족하면 문장을 수행하고 맞지 않으면 빠져 나오라는 의미.
  - 예) Java의 **while** 문:
  - 구문: **while** (불리안 표현식) <문장>
  - 의미: 불리안 표현식이 참이면 계속 문장을 수행하다가 거짓이 되면 while문을 빠져 나오라는 의미.
  - 같은 방법으로 배정문, 조건문, 반복문 등의 의미

3.1 서론(구문과 의미)

- 문법 예
  - 1) <한글> ::= <초성><중성>
  - <초성> ::= ㄱ | ㄴ
  - <중성> ::= ㅏ | ㅑ
  - 예) 가
  - 2) <program> ::= <stmts>
  - <stmts> ::= <stmt> | <stat>;<stmts>
  - <stmt> ::= <var> = <expr>
  - <var> ::= a | b | c | d
  - <expr> ::= <expr> + <term> | <expr> - <term>
  - <term> ::= <var> | <const>
  - <const> ::= 1 | 2 | 3 | 4 | 5
  - 예) b = 2 + 3 – 4; c = 3 – 2 + 5

3.2 구문(언어의 문법적 요소)

❖ 언어의 문법적 요소

- 언어(language): 알파벳 문자 집합들의 부분 집합 즉, 문장들의 집합
- 문장(sentence): 알파벳의 문자들로 구성된 문자열
- 토큰(token) : 문법적으로 의미 있는 최소 단위
  - ✓ 예약어(keyword: begin, end, for), 연산자(+,-,*), 구분자(delimeters: ,, ., ;), 식별자(identifier), 상수(constant)
  - ✓ 어휘 항목(렉심; lexeme): basic language element appeared in input sentence(예 : total, index)
  - ✓ 패턴(Pattern) : 토큰을 서술하기 위한 규칙
- **예 125페이지)** Java 나 C 문장: "index = 2 * count + 17;"

어휘 항목	토큰	패턴
index	식별자	첫자가 영문자나 언더바이고 둘째자부터는 영문자나 숫자, 언더바
=	equal_sign 연산자	치환연산자(=)
2	int_literal 상수	정수형 숫자상수
*	mult_op 연산자	곱셈 산술연산자(*)
count	식별자	첫자가 영문자나 언더바이고 둘째자부터는 영문자나 숫자, 언더바
+	add_op 연산자	덧셈 산술연산자(+)
17	int_literal 상수	정수형 숫자상수
;	세미콜론 구분자	문장 사이를 구분해주는 구분자

3.2 구문(언어의 문법적 요소)

- 언어는 인식(recognition)과 생성(generation) 방법으로 형식적으로 정의
  - **3.2.1 언어 인식기(Recognizers)**
    - ✓ 문자들의 알파벳 Σ 로 부터 구성된 문자열을 인식하는 장치.
    - ✓ 인식 장치는 문자열을 수락(accept)하거나 거부(reject)한다.
    - ✓ 컴파일러의 구문 분석기가 언어에 대한 인식기
    - ✓ - 구문 분석기에 대해서는 4장에서 다룸.
  - **3.2.2 언어 생성기(Generators)**
    - ✓ 언어 생성기는 언어의 문장들을 생성하는데 사용되는 장치
    - ✓ 일반적으로 문법으로 언어들은 생성

추가(형식문법)

- 형식 문법은 크게 두 가지 방법으로 정의할 수 있다. 생성 규칙(production rule)만을 가지고 표현 하거나 다음과 같이 네 가지 항목으로 정의하는 것이다
- **<정의 >** 문법 $G=(V_N, V_T, P, S)$ 정의
  1. V_N : 논터미널 기호들의 유한 집합
  2. V_T : 터미널 기호들의 유한 집합

     $V_N \cap V_T = \phi$, $V_N \cup V_T = V$
     - 터미널 기호와 논터미널 기호를 문법 기호(***grammar symbol***)라 하며 보통 V(vocabulary)로 표시
  3. P : 생성 규칙의 집합으로 다음과 같다.

     $\alpha \rightarrow \beta,\ \alpha \in V^+,\ \beta \in V^*$

     α를 왼쪽 부분(left-hand side), β를 오른쪽 부분(right-hand side)

     → 는 왼쪽 부분에 있는 기호가 오른쪽 부분에 있는 기호로 단순히 대체
  4. S : V_N에 속하는 기호로서 다른 논터미널 기호들과 구별하여 시작 기호(start symbol)

3.3.1.5 문법과 유도

- 정의된 문법으로 부터 어떤 언어가 생성되는지, 언어가 문법에 맞는지를 알기 위해 유도를 설명한다
- $\Rightarrow$: 유도(derivation)
  - 만약 $\alpha \rightarrow \beta$ 가 존재 하고, γ, $\delta \in V^*$ 이면

    $\gamma \alpha \delta \Rightarrow \gamma \beta \delta$ 로 표시
    - ✓ 즉, 한 문자열에서 생성 규칙을 한 번 적용해서 다른 문자열로 바꾸는 것을 나타낸다.
- $\overset{*}{\Rightarrow}$: 영 번 이상의 유도(zero or more derivation)
- $\overset{+}{\Rightarrow}$: 한 번 이상의 유도(one or more derivation)
  - $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$: $\alpha_1, \alpha_2, \alpha_3, \cdots, \alpha_n$ 이 V^*에 속하고 $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \cdots \Rightarrow \alpha_n$ 이 존재
    - ✓ α_1 은 α_n을 **생성(produce)** 혹은 **유도(derivation)**한다고 함
    - ✓ α_n 은 α_1 으로 감축(**reduce**)된다고 함.
    - ✓ $\rightarrow$ 는 생성 규칙에서 사용되는 기호로 단일 화살표(single arrow)라 하고, $\Rightarrow$ 는 유도 과정을 나타내는 기호로 이중 화살표(double arrow)라 한다.

3.3.1.5 문법과 유도

❖ 문장과 문장 형태

- $S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \cdots \Rightarrow w_n \Rightarrow w$와 같은 유도과정이 있을 때 $S \overset{*}{\Rightarrow} w$ 이고 w가 V^*에 속하면 w를 문장 형태(sentential form)라 하고, w가 V_T^*에 속할 경우, w를 문장(sentence)이라 한다.
- 즉, 문장 형태는 터미널 기호와 논터미널 기호들의 조합으로 구성되고 문장은 터미널 기호들로만 구성
- 우리가 프로그램을 작성할 때 사용하는 터미널 기호로만 구성된 문장을 언어라 한다.
- **예** : 문법 $G = (\{S, A\}, \{0, 1\}, P, S)$

  생성규칙 P: $S \rightarrow 0AS$ $S \rightarrow 0$

  $A \rightarrow S1A$ $A \rightarrow 10$ $A \rightarrow SS$

  ➔ $S \Rightarrow 0AS$ (생성 규칙 $S \rightarrow 0AS$ 적용)

  $\Rightarrow 010S$ (생성 규칙 $A \rightarrow 10$ 적용)

  $\Rightarrow 0100$ (생성 규칙 $S \rightarrow 0$ 적용) 의 유도가 적용

  $S \overset{*}{\Rightarrow} 0100$이라고 쓸 수 있고 S, $0AS$, $010S$, 0100은 문장 형태, 0100을 문장 , 문장 0100은 시작 기호 S로 부터 생성된다고 한다.

3.3.1.5 문법과 유도

- 좌단 유도(leftmost derivation)
  - 유도 과정의 각 단계에서 문장 형태의 가장 왼쪽에 있는 논터미널 기호를 계속해서 대체(replacement)하는 경우
  - $\underset{lm}{\Rightarrow}$ 으로 표시
  - 좌 문장 형태(left - sentential form) : 좌단 유도 시 나타나는 문장 형태
- 우단 유도(rightmost derivation)
  - 유도 과정의 각 단계에서 문장 형태의 가장 오른쪽에 있는 논터미널 기호를 계속해서 대체(replacement)하는 경우
  - $\underset{rm}{\Rightarrow}$ 으로 표시
  - 우 문장 형태(right - sentential form) : 우단 유도 시 나타나는 문장 형태
  - **예** : 문법 $G = (\{S, A\}, \{a, b\}, P, S)$

    단, $P : S \rightarrow aAS \mid a$

    $A \rightarrow SbA \mid SS \mid ba$

    이 경우 문장 $aabbaa$를 유도하는 과정에 대해서 좌단 유도와 우단 유도

    ➔ 1) 좌단 유도의 경우

    $$S \underset{lm}{\Rightarrow} aAS \underset{lm}{\Rightarrow} aSbAS \underset{lm}{\Rightarrow} aabAS \underset{lm}{\Rightarrow} aabbaS \underset{lm}{\Rightarrow} aabbaa$$

    2) 우단 유도의 경우

    $$S \underset{rm}{\Rightarrow} aAS \underset{rm}{\Rightarrow} aAa \underset{rm}{\Rightarrow} aSbAa \underset{rm}{\Rightarrow} aSbbaa \underset{rm}{\Rightarrow} aabbaa$$

보통의 경우 $\underset{lm}{\Rightarrow}$, $\underset{rm}{\Rightarrow}$ 대신에 $\Rightarrow$ 로 표시하더라도 대체되는 기호에 의해서 좌단 유도, 우단 유도 임을 알 수 있다.

3.3.1.5 문법과 유도

➢ **예제3.1]** 어떤 작은 언어를 위한 문법이 다음과 같다.

<program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt>

| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>

| <var> - <var>

| <var>

프로그램 **begin** A = B + C ; B = C **end** 는 문법에 맞는 프로그램인지 확인

➔ 우리는 좌단 유도를 통해서 확인한다.

<program> ⇒ **begin** <stmt_list> **end**

⇒ **begin** <stmt> ; <stmt_list> **end**

⇒ **begin** <var> = <expression> ; <stmt_list> **end**

⇒ **begin** A = <expression> ; <stmt_list> **end**

⇒ **begin** A = <var> + <var> ; <stmt_list> **end**

⇒ **begin** A = B + <var> ; <stmt_list> **end**

⇒ **begin** A = B + C ; <stmt_list> **end**

⇒ **begin** A = B + C ; <stmt> **end**

3.3.1.5 문법과 유도

⇒ **begin** A = B + C ; <var> = <expression> **end**
⇒ **begin** A = B + C ; B = <expression> **end**
⇒ **begin** A = B + C ; B = <var> **end**
⇒ **begin** A = B + C ; B = C **end**

- 프로그램 **begin** A = B + C ; B = C **end** 는 문법에 맞는 프로그램이라는 것을 확인하였다.
- **우리는 우단 유도를 통해서도 같은 결과를 얻을 수 있다.**

- **예제3.2]** 전형적인 프로그래밍 언어의 일부분에 대한 문법이 다음과 같다.

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

이 문법은 곱셈과 덧셈의 연산자와 괄호를 포함하는 산술식을 우변으로 갖는 배정문이다. 프로그램 A = B * (A + C) 가 문법에 맞는 프로그램인지 확인해보자.

3.3.1.5 문법과 유도

➔ 우리는 좌단 유도를 통해서 확인한다.

<assign> ⇒ <id> = <expr>
⇒ A = <expr>
⇒ A = <id> * <expr>
⇒ A = B * <expr>
⇒ A = B * (<expr>)
⇒ A = B * (<id> + <expr>)
⇒ A = B * (A + <expr>)
⇒ A = B * (A + <id>)
⇒ A = B * (A + C)

➔ 여기서 연산자를 4칙 연산자로 확장하면 문법은 어떻게 되겠는가?

3.3.1.6 파스 트리

❖ 파스 트리(parse tree)

- ➢ 유도 과정을 이해하기 쉬운 트리 형태의 계층적 구조로 나타낸 것.

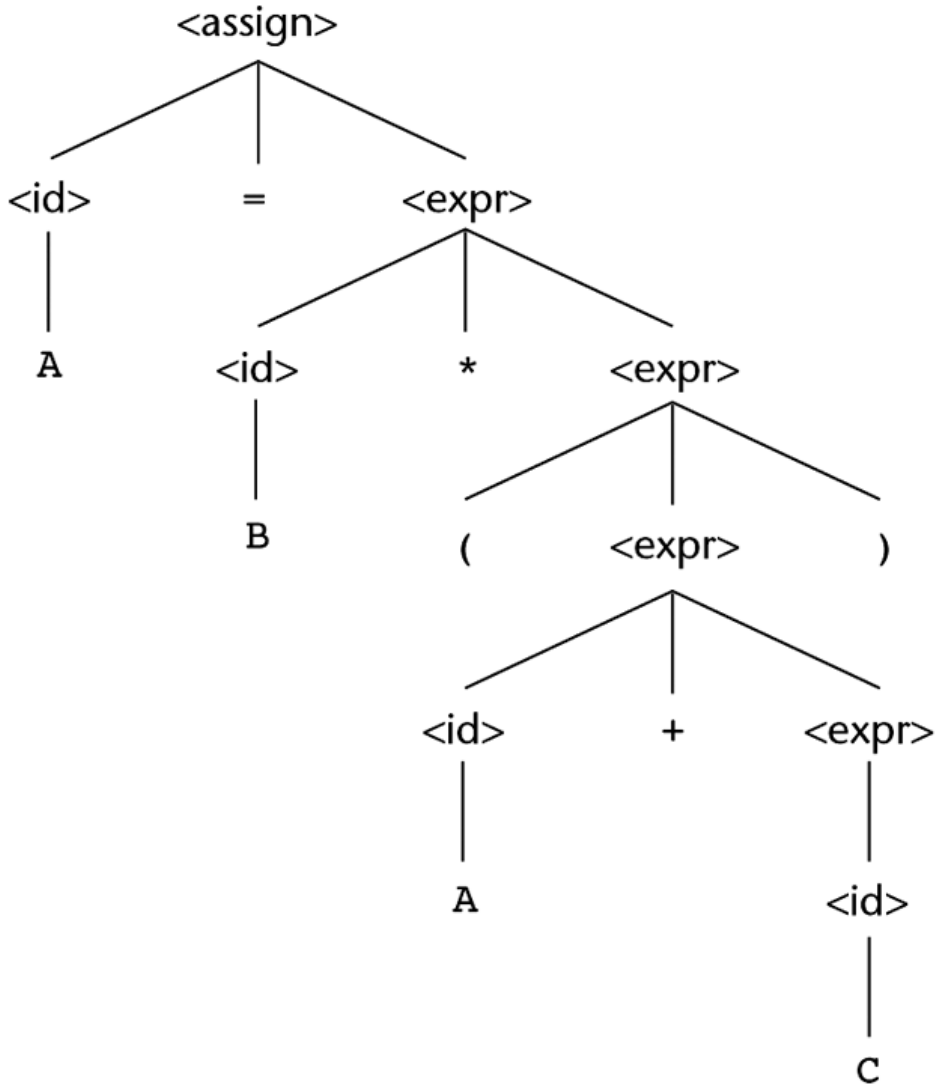
- ➢ CFG(Context - free 문법) G = (V_N, V_T, P, S)에 대한 파스 트리의 정의
  - ✓ 다음을 만족하는 트리이다.
  - ✓ 1) 모든 정점(vertex)의 이름은 문법기호이다.
  - ✓ 2) 루트(root) 정점의 이름은 시작 기호 S이다.
  - ✓ 3) 만약 어떤 정점이 하나 이상의 자식(child)을 갖는다면 이 정점의 이름은 논터미널 기호이다.
  - ✓ 4) 왼쪽부터 순서적으로 X_1, X_2, …, X_n의 n개의 자식을 갖는 어떤 정점 A가 존재한다면 생성 규칙 $A \rightarrow X_1\ X_2 \cdots X_n$ 가 존재한다.
  - ✓ 5) 만약, 어떤 정점이 자식을 하나도 가지고 있지 않다면, 이 정점을 잎(leaf) 노드라 하고 잎의 이름은 터미널 기호이다.

3.3.1.6 파스 트리

Figure 3.1

A parse tree for the simple statement

`A = B * (A + C)`



3.3.1.7 모호성

❖ 문법 *G*에 의해 생성되는 어떤 문장이 두 개 이상의 파스 트리를 갖는다면 문법 *G*는 **모호하다**(ambiguous)고 한다.

➢ **예제3.3] 단순 배정문에 대한 모호한 문법**

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

[풀이] 이 문법은 모호하다. 왜냐하면 배정문 A=B+C*A에 대해 좌단 유도를 하면 다음과 같이 두 개의 서로 다른 유도가 가능하다.

1) <assign> ⇒ <id> = <expr>
⇒ A = <expr>
⇒ A = <expr> + <expr>
⇒ A = B + <expr>
⇒ A = B + <expr> * <expr>

3.3.1.7 모호성

❖ ⇒ A = B + C * <expr>

⇒ A = B + C * A

2) <assign> ⇒ <id> = <expr>

⇒ A = <expr>

⇒ A = <expr> * <expr>

⇒ A = <expr>+<expr> * <expr>

⇒ A = B+<expr> * <expr>

⇒ A = B+C * <expr>

⇒ A = B+C * A

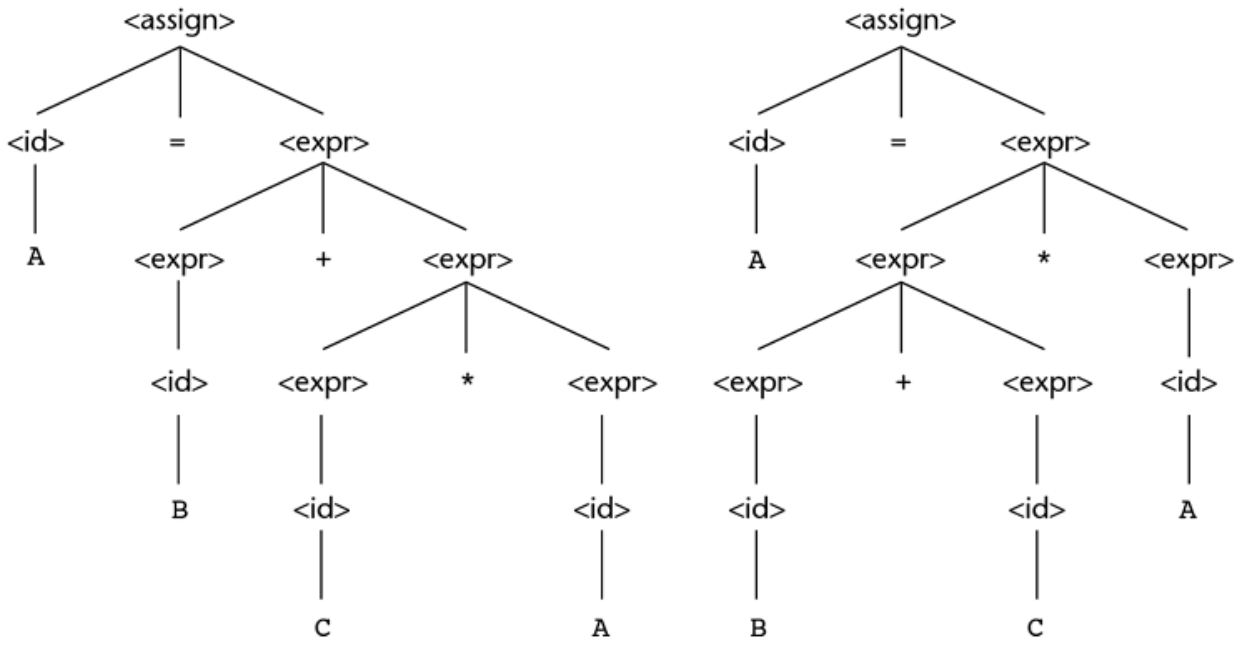
이 두 개의 유도를 파스 트리로 그리면 [그림 3.2]와 같다.

3.3.1.7 모호성

❖

Figure 3.2

Two distinct parse trees for the same sentence,
A = B + C * A



❖ 그래서 [예제 3.3]의 문법은 모호하다.

❖ 이 파스 트리를 보면 왼쪽은 * > +를 뜻하고 오른쪽은 + > *임을 보여 주고 있다.

3.3.1.8 연산자 우선순위

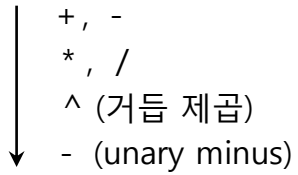
- **모호한 문법을 가지고 충돌을 발생시키지 않게 하기 위해서는 두 가지 방법이 사용**
  - ✓ 1. 모호한 문법을 모호하지 않은 문법으로 변환시킨 후 모호하지 않은 문법을 가지고 구문 분석기를 만드는 방법
  - ✓ 2. 모호한 문법을 가지고 구문 분석기를 만든 다음에 충돌이 발생한 구문 분석기에서 충돌을 없애는 방법

- **모호한 문법은 모호하지 않은 동등한 문법으로 변환 가능**
  - ✓ 그러나 모호한 모든 문법을 동등한 모호하지 않은 문법으로 변환할 수 있는 것은 아님
  - ✓ 산술식에 대한 모호한 문법은 연산자의 우선 순위(precedence)와 결합 법칙(associativity)을 이용하여 모호하지 않은 문법으로 변환 가능

- 문장 3 + 4 * 5를 계산하는데 [예제 3.3]의 문법을 가지고
  - ✓ [그림 3.2]의 왼쪽과 같이 계산하면 * 를 + 보다 연산자의 우선 순위를 높게 준 경우로서 계산된 결과 값이 23
  - ✓ [그림 3.2]의 오른쪽과 같이 계산하면 + 를 * 보다 연산자의 우선 순위를 높게 준 경우로서 계산된 결과 값이 35

3.3.1.8 연산자 우선순위

- 연산자 우선 순위를 어떻게 주느냐에 따라 예의 문법은 문장 3 + 4 * 5에 대한 유도 트리를 유일하게 만들 수 있다.
  - ✓ 통상의 경우 연산자 우선 순위를 순위가 낮은 것부터 나타내면 다음과 같다.



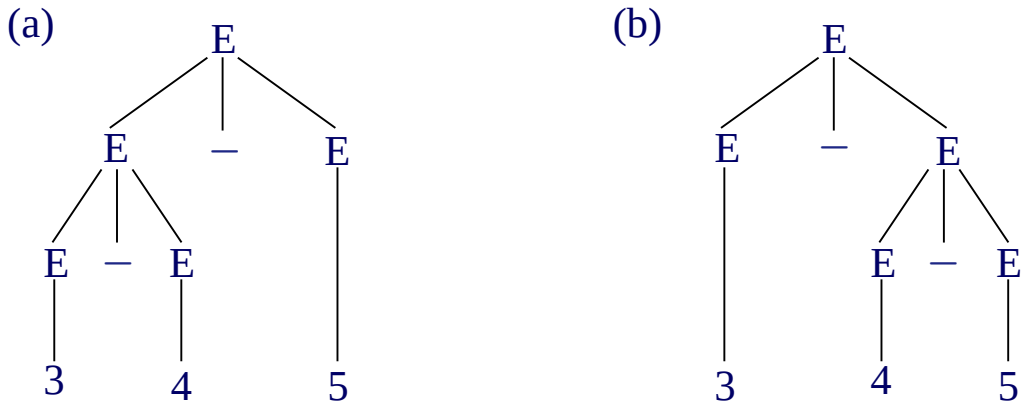
  - ✓ 연산자 우선 순위만을 가지고는 모호한 문법들이 모두 모호하지 않은 문법으로 변환되지는 않는다.

- 예제 : [예제 3.3]의 문법을 조금 수정해보자. 그리고 문장 3 - 4 - 5를 좌단 유도로 나타내면 다음과 같이 두개의 파스 트리가 생성

➔

```
<E> → <E> + <E>
      | <E> * <E>
      | <E> - <E>
      | <E> / <E>
      | (<expr>)
      | 3 | 4 | 5
```

3.3.1.8 연산자 우선순위



- 이런 경우에도 두 개의 서로 다른 결과 값이 생긴다. (a)의 경우는 유도 트리에서 보면 3-4를 먼저 계산하므로 결과 값은 -6이 생긴다. (b)의 경우에는 4-5를 먼저 계산하므로 결과 값이 4이다. 여기에서는 4-5를 계산하지 -4-5를 계산하지 않는다는 것이다. 왜냐하면 -4의 -는 이항 연산자이기 때문이다.

3.3.1.8 연산자 우선순위

- 결합 법칙(associativity)
  - ✓ 연산자 우선 순위가 같은 경우에 어느 것을 먼저 계산하느냐 하는 것을 해결하는 방법
  - ✓ 연산자의 우선 순위가 같은 경우에 왼쪽으로부터 오른쪽으로 계산할 것이냐, 아니면 오른쪽으로 부터 왼쪽으로 계산하느냐 하는 것
  - ✓ 좌측 결합(left associative) : 왼쪽으로부터 오른쪽으로 계산하는 것
  - ✓ 우측 결합(right associative) : 오른쪽으로부터 왼쪽으로 계산하는 것
  - ✓ 통상의 경우 결합 법칙은 좌측 결합을 취함
    - ♣ 그러나 거듭 제곱의 경우는 우측 결합 법칙을 취한다. 그리고 고급 언어 중 인터프리터 언어인 APL(A Programming Language)은 일반적인 연산자들은 우측 결합 법칙을 취하고 거듭 제곱의 경우는 좌측 결합 법칙을 취한다.

3.3.1.8 연산자 우선순위

- **[예제 3.3]을 가지고 모호한 문법을 모호하지 않은 문법으로 변환. 단 연산자 우선순위는 * > + 이고 *와 +가 좌측 결합 법칙을 갖도록 변환**
  1. 가장 기초적인 피연산자를 먼저 처리하고 이 생성 규칙이 생성 규칙 중에서 가장 아래에 온다. 가장 기초적인 피연산자를 <factor>라 하면 다음과 같다.

     <factor> → (<expr>) | <id>
  2. 다음으로는 연산자 순위가 높은 것부터 취한다. *가 가장 높고, 좌측 결합을 취하므로 다음 생성 규칙과 같이 재귀적으로 구성

     <term> → <term> * <factor> | <factor>
     - ♣ 만약 * 연산자가 우측 결합 법칙을 취할 경우는 다음과 같다.
       - <term> → <factor> * <term> | <factor>
     - ♣ 만약 연산자 우선 순위가 같은 것들이 있다면 같은 줄에 표기하면 된다. 만일 연산자 /가 있어서 * 연산자와 같은 우선순위를 갖는다면 다음과 같다.
       - <term> → <factor> * <term> | <factor> / <term> | <factor>
  3. 같은 방법으로 다음 우선 순위가 +이고 좌측 결합을 취하므로 다음 생성 규칙과 같이 재귀적으로 구성

     <expr> → <expr> + <term> | <term>

❖ **이 생성 규칙들을 모아보면 [예제 3.4]와 같다.**

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → (<expr>) | <id>

3.3.1.8 연산자 우선순위

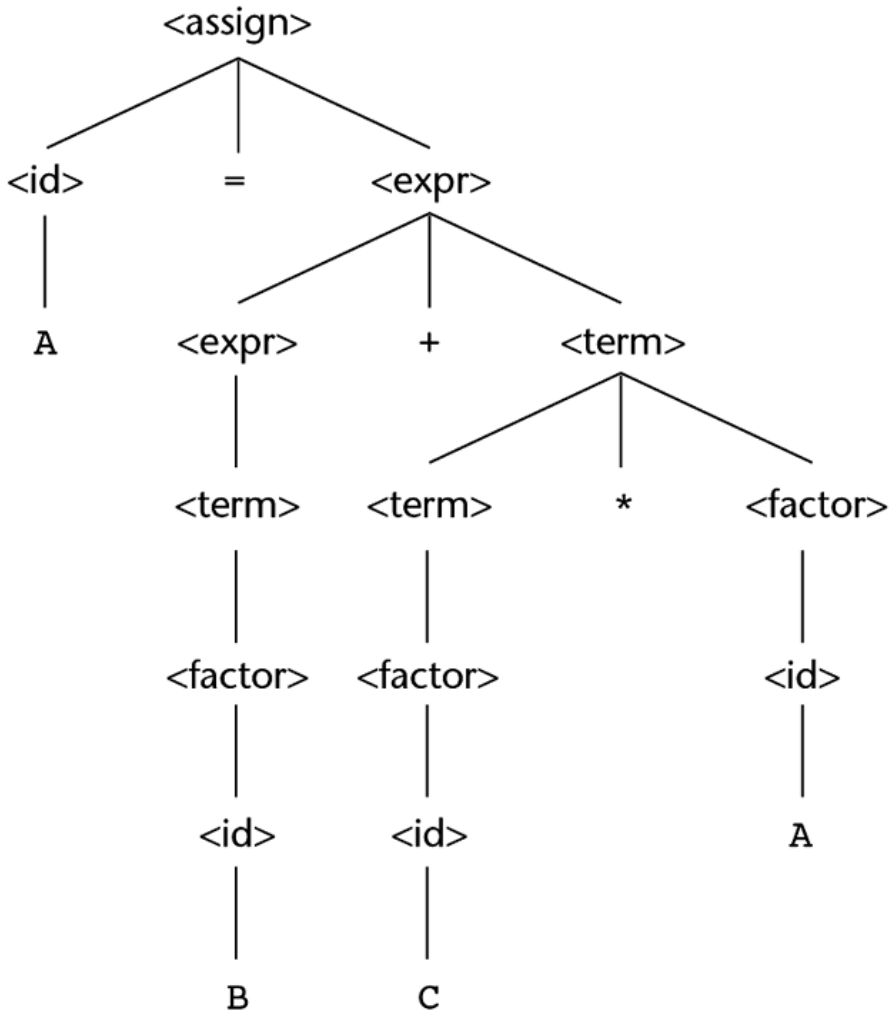
- **[예제 3.4]를 가지고 문장 A=B+C*A에 좌담 유도를 하면 다음과 같다.**
  - ✓ <assign> ⇒ <id> = <expr>
    ⇒ A = <expr>
    ⇒ A = <expr> + <term>
    ⇒ A = <term>+ <term>
    ⇒ A = <factor>+ <term>
    ⇒ A = <id>+ <term>
    ⇒ A = B + <term>
    ⇒ A = B + <term> * <factor>
    ⇒ A = B + <factor> * <factor>
    ⇒ A = B + <id> * <factor>
    ⇒ A = B + C * <factor>
    ⇒ A = B + C * <id>
    ⇒ A = B + C * A

이에 대한 파스 트리는 [그림 3.3]과 같다.

3.3.1.8 연산자 우선순위

Figure 3.3

The unique parse tree for `A = B + C * A` using an unambiguous grammar



3.3.1.8 연산자 우선순위

- **[예제 3.4]를 가지고 문장 A=B+C*A에 우단 유도를 하면 다음과 같다.**
  - ✓ <assign> ⇒ <id> = <expr>
    ⇒ <id> = <expr> + <term>
    ⇒ <id> = <expr> + <term> * <factor>
    ⇒ <id> = <expr> + <term> * <id>
    ⇒ <id> = <expr> + <term> * A
    ⇒ <id> = <expr> + <factor> * A
    ⇒ <id> = <expr> + <id> * A
    ⇒ <id> = <expr> + C * A
    ⇒ <id> = <term> + C * A
    ⇒ <id> = <factor> + C * A
    ⇒ <id> = <id> + C * A
    ⇒ <id> = B + C * A
    ⇒ A = B + C * A

3.3.1.8 연산자 우선순위

- **예 :** 다음의 모호한 문법을 모호하지 않은 문법으로 만들어보자. 연산자 우선순위와 결합 법칙은 통상적인 것을 따른다.

  P : E → E + E | E - E | E * E | E / E | E ^ E | -E | (E) | id

  ➔ 가장 기초적인 피 연산자는 (E)와 id이다.

  element → (E) | id

  다음으로 연산자 우선 순위가 가장 높은 것은 단항 연산자(unary operator) - 이므로

  primary → - primary | element

  다음은 ^ (여기서 연산자 ^는 우측 결합 법칙을 취함)

  factor → primary ^ factor | primary

  같은 방법으로 다음과 같은 생성 규칙을 만들 수 있다.

  term → term * factor | term / factor | factor

  E → E + term | E - term | term

  이를 정리하면 다음과 같은 모호하지 않은 문법이 만들어진다.

  E → E + term | E - term | term

  term → term * factor | term / factor | factor

  factor → primary ^ factor | primary

  primary → - primary | element

  element → (E) | id

3.3.1.10 현수 else

❖ 현수 else(dangling else)

- 중첩된 if 문장에서 else가 어떤 if 문에 걸려있느냐 하는 것
- 다음과 같은 문법을 고려해보자.

stat → **if** *expr* then *stat*
| **if** *expr* then *stat* **else** *stat*
| other

"other"는 임의의 다른 문장을 뜻한다. 이 문법에 따른 다음과 같은 조건 문장을 있다.

if E_1 then if E_2 then S_1 else S_2

이 문장이 가지는 두 개의 서로 다른 의미를 괄호를 이용하여 표시하면

if E_1 then (if E_2 then S_1 else S_2)

if E_1 then (if E_2 then S_1) else S_2

if ~ then 문장과 if ~ then ~ else 문장이므로 주어진 문법은 모호한 문법

- 모호한 문법을 모호하지 않는 문법으로 변환하기 위한 방법
  - ✓ 일반적인 프로그래밍 언어에서는 이런 경우에 else를 그 앞에 있는 제일 가까운 if와 연결
    문장 if E_1 then if E_2 then S_1 else S_2는 문장 if E_1 then (if E_2 then S_1 else S_2) 으로 해석
    이와 같은 의미가 되도록 하기 위해서는 다음과 같은 모호하지 않는 문법으로 변환할 수 있다.

stat → *matched-stat* | *unmatched-stat*
matched-stat → **if** *expr* then *matched-stat* **else** *matched-stat*
| other
unmatched-stat → **if** *expr* then *stat*
| **if** *expr* then *matched-stat* **else** *unmatched-stat*

3.3.1.2 문법의 표기법(BNF)

❖ BNF(Backus - Naur Form) 표기법

- 프로그래밍 언어의 formal definition을 위해 가장 널리 사용되는 방법
- 이 표기법은 ALGOL 60의 문법을 표현하기 위해 가장 먼저 사용
- 현재에는 대부분의 언어들을 표현하는데 가장 많이 사용
- 이 표기법은 메타 기호(메타 기호는 표현하려는 언어의 일부분이 아니라, 그 언어를 표현하려고 사용된 특수기호)로서 세 가지 기호를 사용
  - ✓ 논터미널 기호는 <와 >로 묶어 표현
  - ✓ 대체(replacement)를 나타내기 위해서 ::=를 사용
  - ✓ 양자 택일을 나타내기 위해서 | 를 사용

- 예 : 우리가 보통 사용하는 식별자(identifier)는 첫 자가 영문자로 시작하고, 두 번째 자부터 영문자나 숫자 오면 된다. 이러한 식별자를 BNF 표기법으로 표시하면 다음과 같다.(무한 언어를 표현하기 위해서 재귀적으로 표현)

  ➔ <식별자> ::= <영문자> | <식별자><영문자> | <식별자><숫자>

  <영문자> ::= *a* | *b* | *c* | … | *z*

  <숫자> ::= 0 | 1 | 2 | … | 9

  ➔ <ident_list> ::= <identifier> | <identifier>, <ident_list>

  <identifier> ::= A | B | C

3.3.1.2 문법의 표기법(BNF)

- 예 : 앞의 예에서 정의한 식별자의 길이가 길어야 8이라고 할 때 BNF를 이용해서 문법을 정의해 보자.
  - ✓ BNF를 이용해서 표현하기에는 다음과 같은 어려움이 따름

    <식별자> ::= <영문자> | <영문자><영문자> | <영문자><숫자>

    | <영문자><영문자><영문자>|<영문자><영문자><숫자>

    | <영문자><숫자><영문자> | <영문자><숫자><숫자> |

    …

    | <영문자><숫자><숫자><숫자><숫자><숫자><숫자><숫자>

    <영문자> ::= *a* | *b* | *c* | … | *z*

    <숫자> ::= 0 | 1 | 2 | … | 9
  - ✓ 위와 같이 나열해야 되는데 중간에 엄청난 가지 수의 생성 규칙이 존재
  - ✓ 이와 같이 BNF는 반복되는 부분을 표시하는데 어려움을 가짐
- 반복되는 부분을 쉽게 표시하면서 BNF로 표시하는 방법이 EBNF이다.

3.3.2 문법의 표기법(EBNF)

❖ EBNF(Extended BNF)

- BNF 표기법은 반복되는 부분을 표시하는데 어려움을 가짐
- 반복되는 부분을 BNF 표기법보다 읽기 쉽고 간결하게 표현
- 반복되는 부분을 나타내기 위해서 메타 기호로 { }와 < >를 사용
  - ✓ {a}는 a가 0번 이상 반복될 수 있다는 것을 의미
    - ♣ 정규표현 a^*와 같은 의미로 생각
  - ✓ 또한 선택적인 부분을 표시할 때는 [ ]로 표현
    - ♣ [x]는 x가 나타나지 않거나 한 번 나타날 수 있음을 의미
    - ♣ [x]는 $\{x\}_0^1$과 같은 의미
- 메타 기호를 터미널 기호로 사용하는 경우에는 그 기호를 ' 와 '로 묶어 표현
  - ✓ { }, [ ], |, < >), ::= 와 같이 EBNF에서 사용되어지는 메타 기호를 터미널 기호로 사용하는 경우 발생하는 혼돈을 피하기 위해서 사용

3.3.2 문법의 표기법(EBNF)

- 예 : 식별자의 길이가 최대 8자이라고 할 때 이를 EBNF로 표현
  - ➔ <식별자> ::= <영문자>{<영숫자>}$_{0}^{7}$
    <영숫자> ::= <영문자> | <숫자>
    <영문자> ::= *a* | *b* | *c* | … | *z*
    <숫자> ::= 0 | 1 | 2 | … | 9
- **예** : if 문장에서 else 부분이 선택적으로 나타날 수 있다면 다음과 같이 표현
  - ➔ <if_st> ::= if <condition> then <statement> [else <statement>]
- **예** : <ident_list> ::= <identifier> | <identifier>, <ident_list>
  - ➔ <ident_list> ::= <identifier> {, <identifier>}
- **예** : 메타 기호를 터미널 기호로 사용하는 예
  - ➔ <BNF_rule> ::= <left_part>'::='<right_part>
    <right_part> ::= <right_part_element>{'|'<right_part_element>}
    ※ 여기서 메타 기호 ::=과 |가 터미널 기호로 사용

3.3.2 문법의 표기법(EBNF)

- **예제3.5] 표현식 문법의 BNF와 EBNF**
- BNF :

```
<expr> → <expr> + <term>
         | <expr> - <term>
         | <term>
<term> → <term> * <factor>
         | <term> / <factor>
         | <factor>
<factor> → <exp> ** <factor>
         | <exp>
<exp> → (<expr>)
         | <id>
```

EBNF :

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor>{(* | /)  <factor>}
<factor> → <exp>{ ** <exp>}
<exp> → (<expr>)
         | <id>
```

3.4 속성 문법(내용 끝임)

❖ 속성 문법(attribute grammar)

- 속성 문법은 속성과 의미 규칙을 사용하여 문맥자유 문법을 확장한 것으로, 각각의 논터미널 기호 또는 터미널 기호의 속성을 결합하여 의미 분석에 이용한다.
- 속성이란 어떤 프로그래밍 언어 구문에 관한 특성이다. 예를 들어 컴퓨터 분야가 아닌 다른 분야에서 어떤 사람이나 객체의 속성은 그 사람의 유머 감각이나 객체의 색깔처럼 그 사람이나 객체를 묘사하는 특성을 말한다.
- 속성 문법은 다양한 응용 분야에 사용.
  - ✓ 프로그래밍 언어의 구문과 정적 의미론에 대해 완벽한 기술을 제공
  - ✓ 컴파일러 생성 시스템의 입력으로 사용될 수 있는 언어의 형식적 정의로 사용
  - ✓ 다양한 구문 지시적 편집 시스템(syntax-directed editing system)의 기반으로 사용
  - ✓ 자연어 처리 시스템에서 사용

3.4 속성 문법(내용 끝임)

- 속성은 프로그래밍 언어의 정의 시간으로부터 번역 시간 및 실행 시간까지 해당 특성이 결정 되는 시간에 따라 다양한 형태로 나타날 수 있다.
  - ✓ 속성의 대표적인 예는 변수의 자료형, 식의 값, 메모리에서 변수의 위치, 어떤 숫자의 유효 자릿수 등이다.
- 이러한 속성을 확정하는 것을 **바인딩(binding)**이라 하고, 바인딩이 결정되는 시간을 **바인딩 시간(binding time)**이라 한다. 또한 바인딩 시간이 프로그램 실행 이전인 경우는 **정적 바인딩(static binding)**, 바인딩 시간이 프로그램 실행 중인 경우는 **동적 바인딩(dynamic binding)**이라 한다.
- 구문 지시적 정의(syntax-directed definition)는 문맥 자유 문법에 속성과 의미 규칙을 결합한 것이다. 속성은 문법 기호와 결합하고, 의미 규칙은 생성 규칙과 결합한다. 만약 X가 문법 기호이고 a가 속성이라면 X.a는 특정 파스 트리 노드 X에서의 a 값을 나타낸다.

3.4 속성 문법(내용 끝임)

- 속성 문법은 속성과 의미 규칙을 사용하여 문맥 자유 문법을 확장한 것으로, 각각의 논터미널 기호 또는 터미널 기호의 속성을 결합하여 의미 분석에 이용한다. 이때 속성이란 어떤 프로그래밍 언어 구문에 관한 특성이다. 예를 들어 컴퓨터 분야가 아닌 다른 분야에서 어떤 사람이나 객체의 속성은 그 사람의 유머 감각이나 객체의 색깔처럼 그 사람이나 객체를 묘사하는 특성을 말한다.
- **[예제 7-1] 구문 지시적 정의의 생성 규칙에 대한 의미 이해하기**
- 다음은 간단한 계산기에 대한 구문 지시적 정의이다. 각 생성 규칙에 대한 의미를 설명해보자

생성 규칙	의미 규칙
① S → E $	print(E.val)
② E → E1 + T	E.val = E1.val + T.val
③ E → T	E.val = T.val
④ T → T1 × F	T.val = T1.val × F.val
⑤ T → F	T.val = F.val
⑥ F → (E)	F.val = E.val
⑦ F → **digit**	F.val = **digit**.lexval

3.4 속성 문법(내용 끝임)

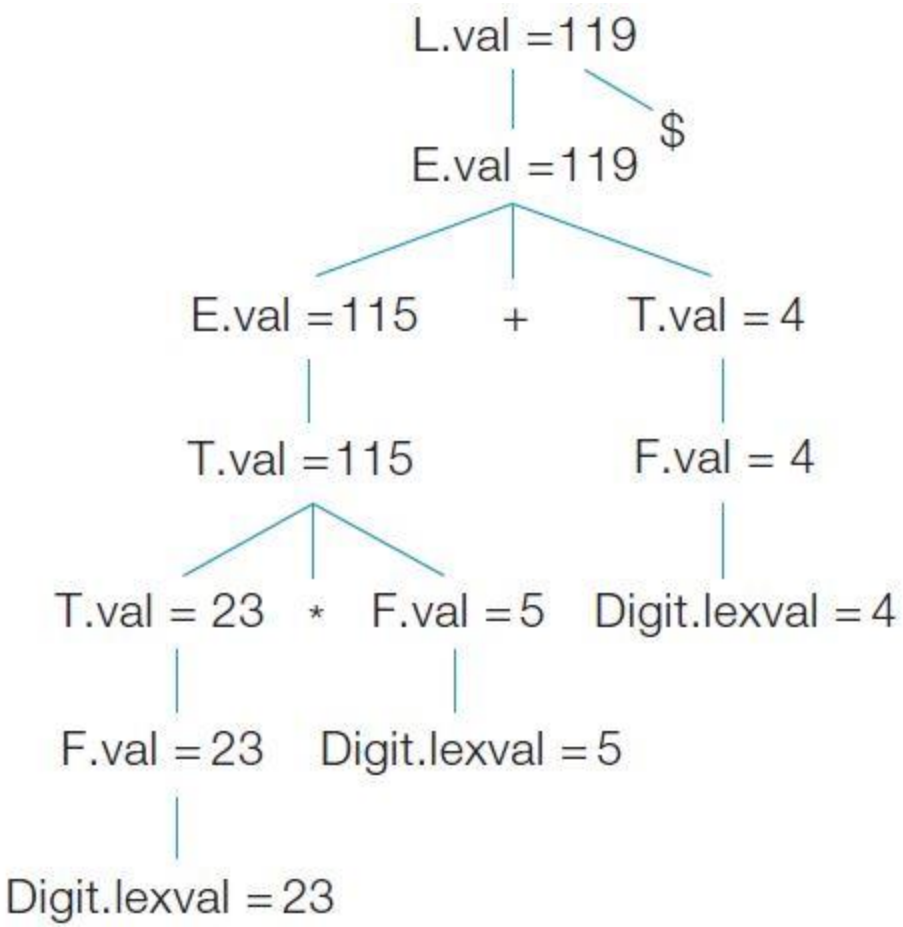
- **[풀이]**
  - ✓ 이 구문 지시적 정의는 연산자 +와 *를 가진 산술식 문법을 바탕으로 한다. 산술식의 끝은 $로 끝난다. 여기서 각 논터미널 기호 E, T, F는 val이라 불리는 단 하나의 속성을 가진다. 또한 터미널 기호 digit는 속성 lexval을 가지는데 lexval의 값은 어휘 분석기가 내주는 정수 값이다.
  - ✓ ① 생성 규칙 1 : S → E$에 대한 의미 규칙은 E.val을 출력하라는 뜻이다.
  - ✓ ② 생성 규칙 2 : E1과 T 속성 값의 합으로 E의 속성 val을 계산한다.
  - ✓ ③ 생성 규칙 3 : T의 속성 값은 E의 속성 val을 계산한다.
  - ✓ ④ 생성 규칙 4 : T1과 F 속성 값의 곱으로 T의 속성 val을 계산한다.
  - ✓ ⑤ 생성 규칙 5 : F의 속성 값은 T의 속성 val을 계산한다.
  - ✓ ⑥ 생성 규칙 6 : E의 속성 값은 F의 속성 val을 계산한다.
  - ✓ ⑦ 생성 규칙 7 : F.val에 하나의 숫자 값을 주는데 이것은 어휘 분석기가 내주는 토큰 digit의 수치 값이다.

3.4 속성 문법(내용 끝임)

- 생성 규칙에 해당하는 의미 규칙을 기술할 때 논터미널 기호는 목적에 따라 문자열, 숫자, 자료형 등을 나타내기 위한 다양한 속성을 표현하고, 이런 속성을 점(.) 연산자를 이용하여 나타낸다. 예를 들어 논터미널 기호 A의 속성으로 값을 나타내는 value와 기억 공간을 나타내는 addr 속성을 표현하기 위해 A.value와 A.addr로 표기한다. 속성 값은 구하는 위치에 따라 **합성 속성(synthesized attribute)**과 **상속 속성(inherited attribute)**으로 구분된다.
- 합성 속성은 생성 규칙의 왼쪽에 있는 논터미널 기호의 속성이 생성 규칙의 오른쪽에 있는 함수에 의해 결정되는 것을 말한다. 즉 생성 규칙 X → ABC에 대해 다음과 같다.
- X.attribute ::= func(A.attribute, B.attribute, C.attribute)
- 반면에 상속 속성은 생성 규칙의 왼쪽에 있는 논터미널 기호의 속성을 이용하여 생성 규칙의 오른쪽에 있는 논터미널 기호의 속성이 결정되는 것을 말한다. 즉 생성 규칙 X → ABC에 대해 다음과 같다.
- A.attribute ::= func(X.attribute, B.attribute, C.attribute)
- 이때 각 노드에서 속성 값을 주석으로 보여주는 **주석 파스 트리(annotated parse tree)**를 만들 수 있다.

3.4 속성 문법(내용 끝임)

- **[예제 7-2] 산술식에 대한 주석 파스 트리 만들기**
- 산술식 23 * 5 + 4$에 대한 주석 파스 트리를 만들어보자
- **[풀이]** 산술식 23 * 5 + 4$에 대한 주석 파스 트리는 다음과 같다.



Report 3

- 3장 복습 문제, 연습문제 풀이(173-177 페이지)
  - 연습문제 6, 8, 12, 28번 풀어서 제출하기
  - 기간 : 1주