Chapter 5

문맥 자유 문법과 파싱 알고리즘 - **Part II**

목차

Table of Contents

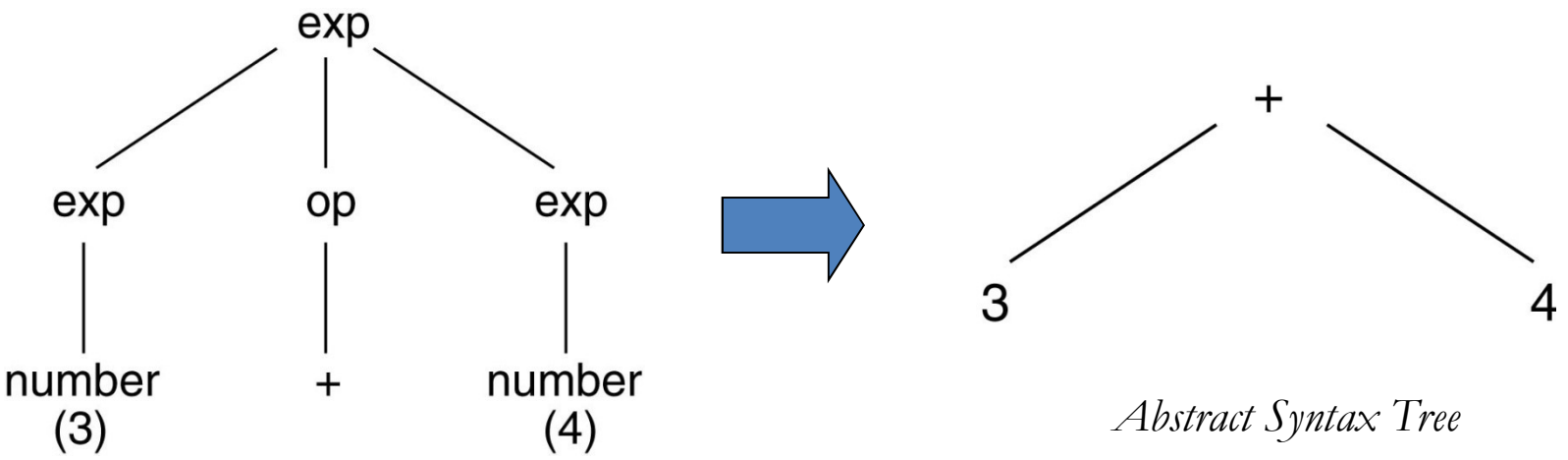
Abstract Syntax Tree (AST)

parse tree는 지나치게 많은 정보를 표현

- Tree 탐색에 시간이 많이 걸림
- Tree 와 같은 동적 구조를 구현하기 위해 많은 메모리가 필요

AST(추상 구문 트리)

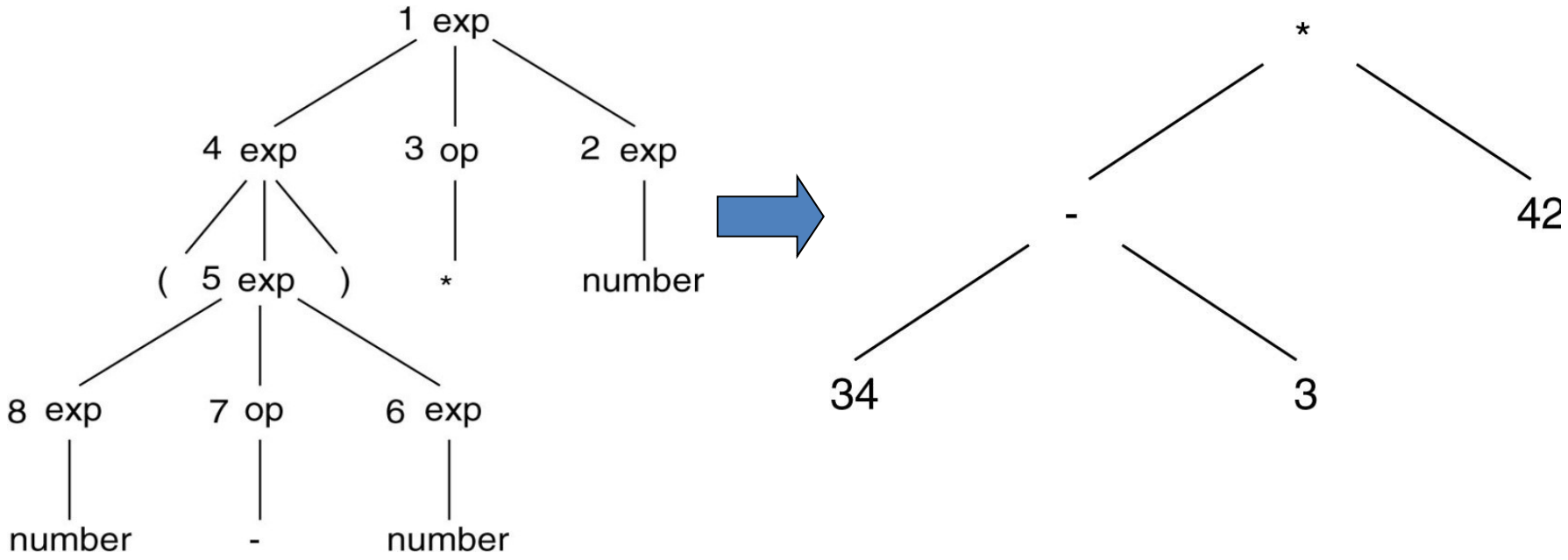
- contains all the information needed for translation
  - in a more *efficient form* than parse trees.



Concrete (Ordinary) Syntax Tree

Abstract Syntax Tree

AST for (34 – 3) * 42



단점 : 원래 *source code sequence*로 되돌릴 수 없음
(reverse compile이 불가능)

Table of Contents

Ambiguous grammar(1/2)

- **예: Grammar 의 생성 규칙이 다음과 같을 때**

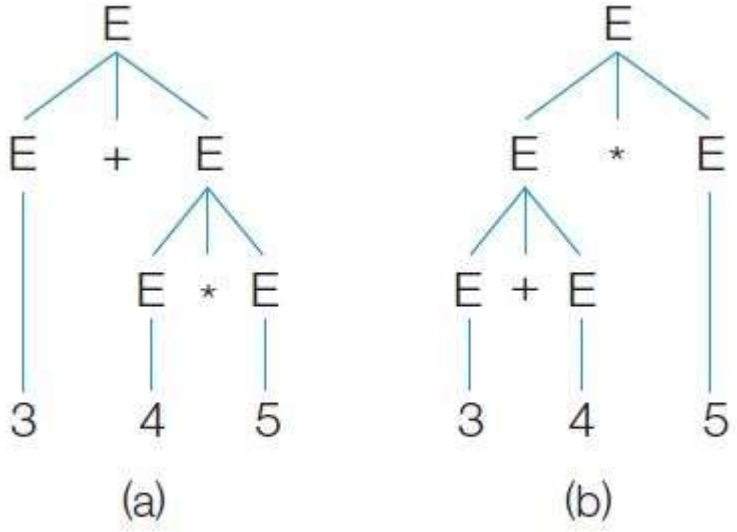
  $E \rightarrow$ E + E | E - E | E * E | E / E | (E)

  $E \rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  입력 3 + 4 * 5 에 대해 구문 분석을 하시오.

E ⇒ E + E
⇒ 3 + E
⇒ 3 + E * E
⇒ 3 + 4 * E
⇒ 3 + 4 * 5

E ⇒ E * E
⇒ E + E * E
⇒ 3 + E * E
⇒ 3 + 4 * E
⇒ 3 + 4 * 5



3+(4*5)=23 (3+4)*5=35

Ambiguous grammar(2/2)

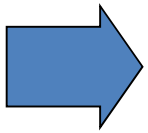
- **하나의 입력에 대해 두 개 이상의 parse tree를 생성할 수 있으면, 문법 *G* 는 모호하다(ambiguous).**
  - 전혀 다른 machine code 생성 → 똑같은 코딩인데 실행 결과가 다름

- **모호한 문법을 해결하는 방법**
  - **문법 변환** : 모호한 문법 ➔ **모호하지 않은 동등한 문법**
    - [예] 산술 연산 : 우선 순위와 결합 법칙을 생성 규칙에 반영
  - 모호하지 않은 문법으로 변환할 수 없는 경우
    - 적용 가능한 생성 규칙 중 **일정한 기준에 따라** 하나를 선택
    - [예] Dangling-else : else는 가장 가까이 있는 if 와 짝이 된다!

문법 변환 : Precedence and Associative rule

- 우선 순위가 같은 연산기호들은 한데 묶음
  ➔ 하나의 rule에 함께 포함
- 우선 순위가 낮을수록 parse tree의 root에 가깝게 배치!

- 좌 결합법칙을 갖는 연산기호는 ***left-recursive*** rule로!
- 우 결합법칙을 갖는 연산기호는 ***right-recursive*** rule로!



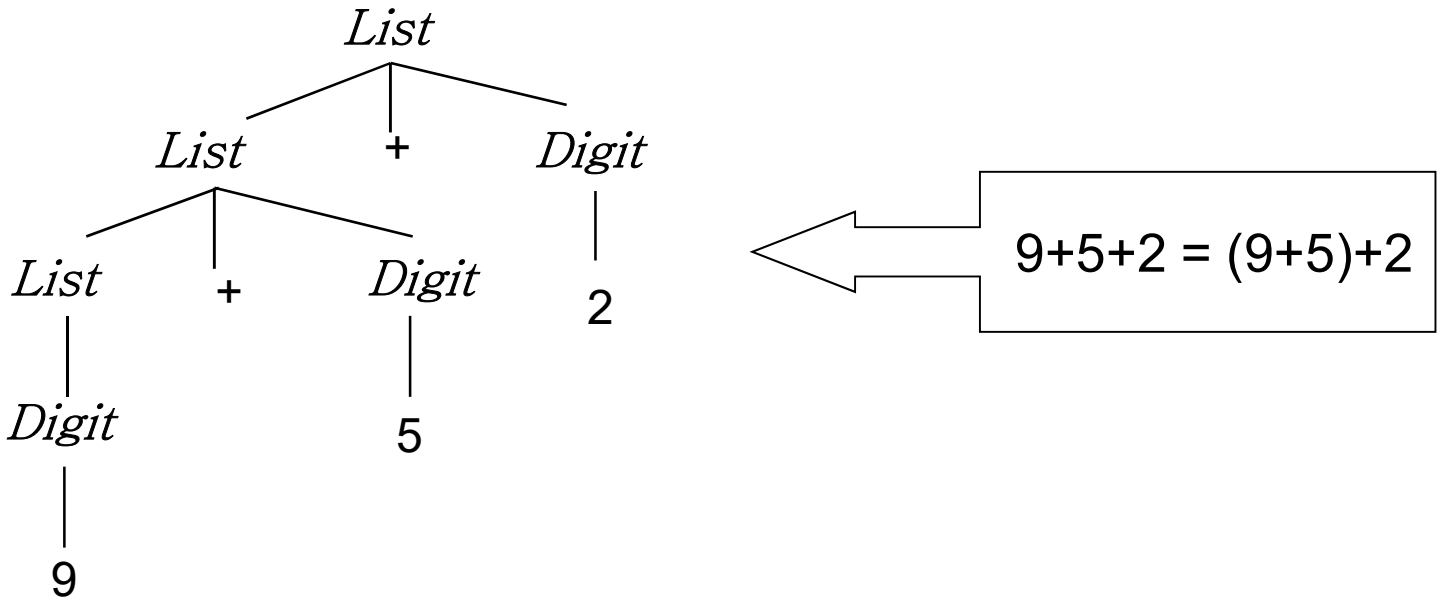
E → **E** + T | **E** - T | T
T → **T** * F | **T** / F | F
F → **(** E **)** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

생성 규칙에서 좌 결합법칙 표현하기

$List \rightarrow List + Digit \mid Digit$

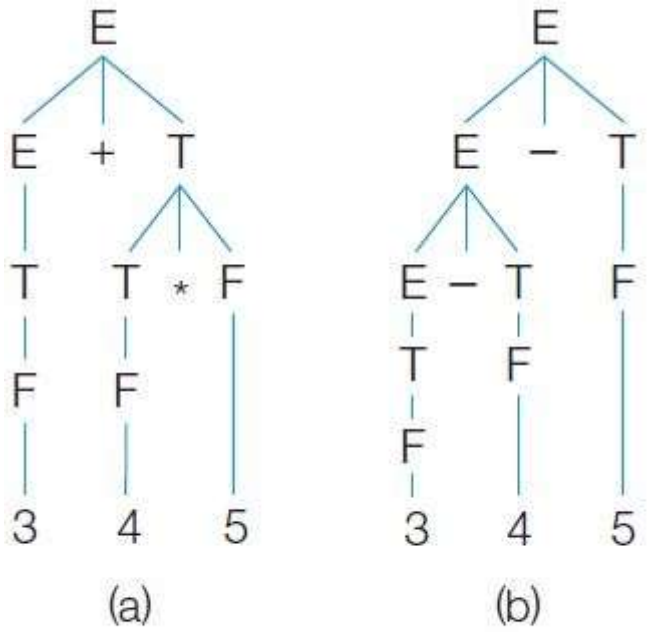
$Digit \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$

Left Associative Rule (좌결합 법칙 = 좌순환 규칙)



예 10: 아래 입력에 대해 구문 분석하시오.

E → **E** + T | **E** - T | T
T → **T** * F | **T** / F | F
F → **(** E **)** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



3+4*5　　3-4-5

Quiz #4

E → E + T | E - T | T

T → T * F | T / F | F

F → P ^ F | P // ^는 거듭 제곱: 2 ^ 3 ➔ 2^3

P → -P | H // - 는 단항 연산자. 마이너스 부호.

H → (E) | 0 | 1 | … | 9

a. 연산기호간 우선 순위는?

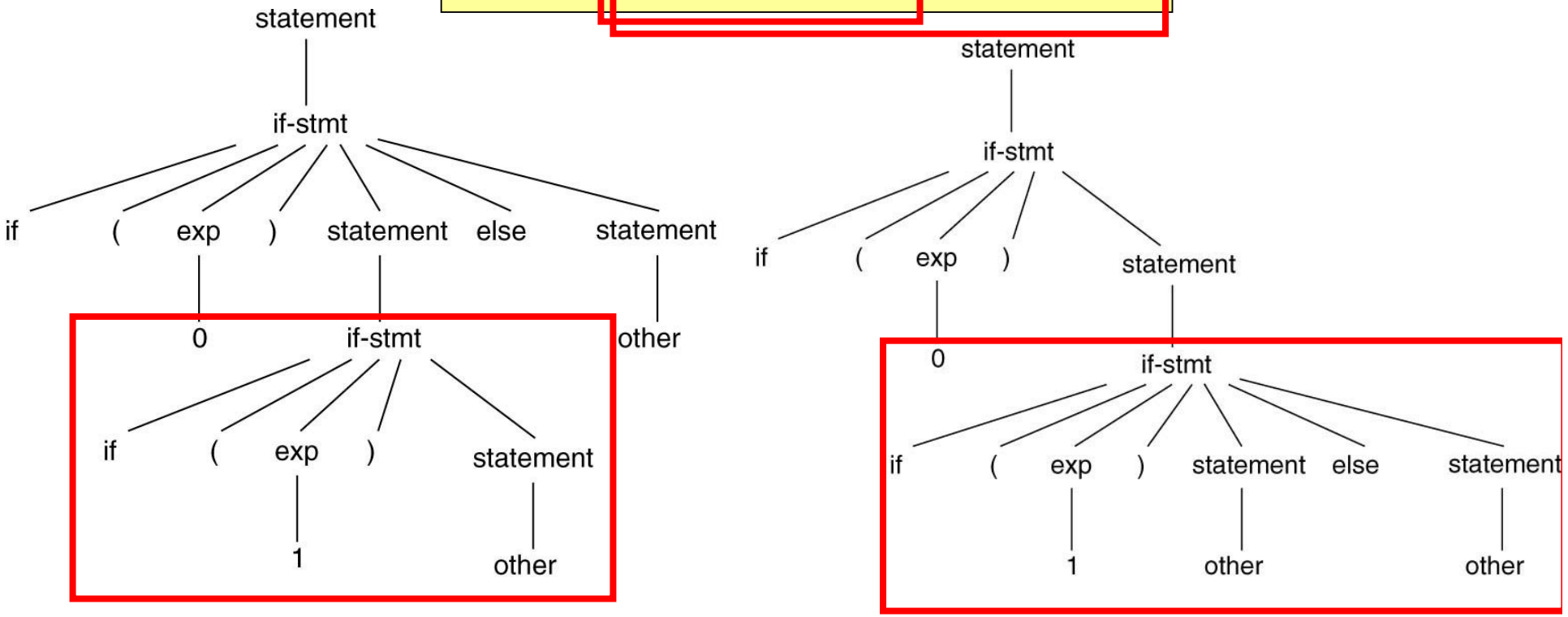
b. 각 연산기호는 어떤 결합 법칙이 적용되는가?

c. 입력 **-2 + 3 * (4 ^ 5)** 에 대해 구문 분석하시오.

The Dangling Else Problem

statement → *if-stmt* | `other`
if-stmt → `if (` *exp* `)` *statement*
| `if (` *exp* `)` *statement* `else` *statement*
exp → `0` | `1`

`if (0) if (1) other else other`



Most closely nested rule

- **`else` 와 가장 가까이에 있는 `if` 와 짝을 이룸**
  - 2번째 parse tree 로 해석

```
if (x != 0)
  if (y == 1/x) ok = TRUE;
  else x = 1/x;
```

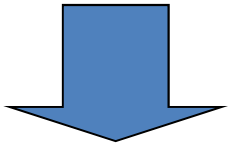
- **첫 번째 `if` 와 짝을 이루려면 괄호를 사용**

```
if (x != 0)
  { if (y == 1/x) ok = TRUE; }
else x = 1/x;
```

a bracketing keyword for the *if* statement

```
if x/=0 then
  if y=1/x then ok := true;
  else z := 1/x;
  end if;
end if;
```

```
if x/=0 then
  if y=1/x then ok := true;
  end if;
else z := 1/x;
end if;
```



statement → *if-stmt* | *other*
if-stmt → if (*exp*) *statement* **endif**
| if (*exp*) *statement* else *statement* **endif**
exp → 0 | 1

Table of Contents

EBNF (*Extended* BNF) : Repetition { }(1/2)

Generic rules

- Left recursive : $A \rightarrow A\ \alpha \mid \beta$

  ➔ **$A \rightarrow \beta\ \{\alpha\}$**

- Right recursive : $A \rightarrow \alpha\ A \mid \beta$

  ➔ **$A \rightarrow \{\alpha\}\ \beta$**

EBNF로 변환하면 문제는 없을까?

- 문장 구조의 특징이 사라짐
  - *Parse tree*가 어떻게 만들어졌는지 알기 어려움

EBNF : *Repetition* { } (2/2)

stmt-sequence → *stmt-sequence* ; *stmt* | *stmt*

stmt-sequence → stmt { ; stmt } (*left* recursive)

stmt-sequence → *stmt* ; *stmt-sequence* | *stmt*

stmt-sequence → { stmt ; } stmt (*right* recursive)

exp → *exp* *addop* *term* | *term*

exp → term {addop term}

exp → *term* *addop* *exp* | *term*

exp → {term addop} term

EBNF : *Optional* [ ] (1/2)

- **Rules for *if*-statement with optional *else-parts***

if-stmt → `if (` *exp* `)` *statement*
| `if (` *exp* `)` *statement* `else` *statement*

if-stmt → if (exp) statement [else statement]

EBNF : *Optional* [ ] (2/2)

순환이 그대로 살아 있음!!!

- 우순환 규칙을 [ ]로 표기

  stmt-sequence → *stmt* ; *stmt-sequence* | *stmt*

  ⇒ **stmt-sequence → stmt [; stmt-sequence]**

  ♣ 참고 : 반복 기호 { } 사용

  stmt-sequence → { stmt ; } stmt

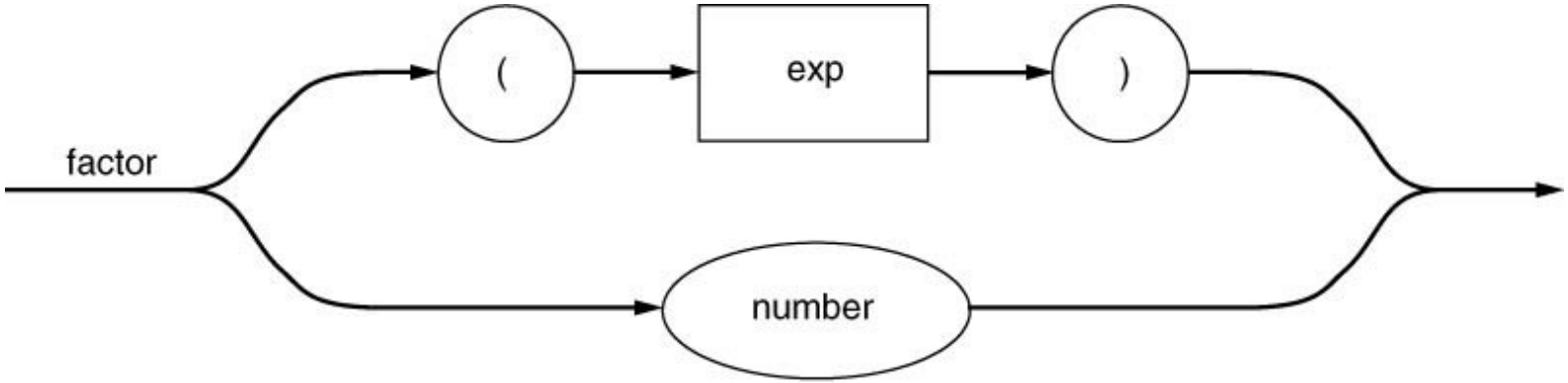
- 우결합 형태를 [ ]로 표기

  exp → *term addop exp* | *term*

  ⇒ **exp → term [addop exp]**

Syntax Diagram (1/2)

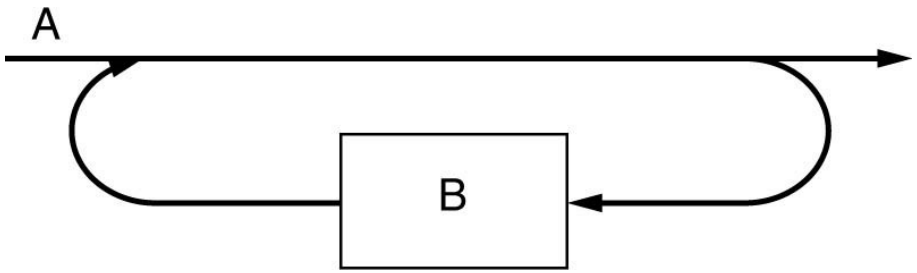
factor → (*exp*) | **`number`**



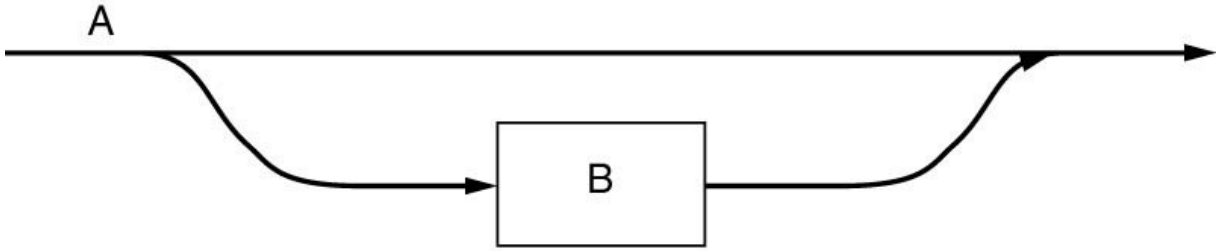
원, 타원 → terminal
사각형 → nonterminal

Syntax Diagram (2/2)

$A \rightarrow \{ B \}$

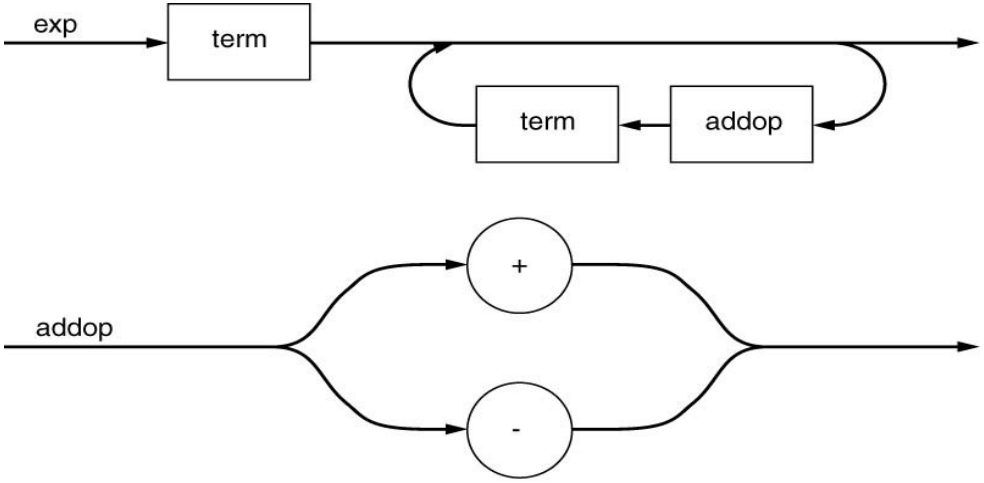


$A \rightarrow [B]$



예 11: Syntax diagram (1)

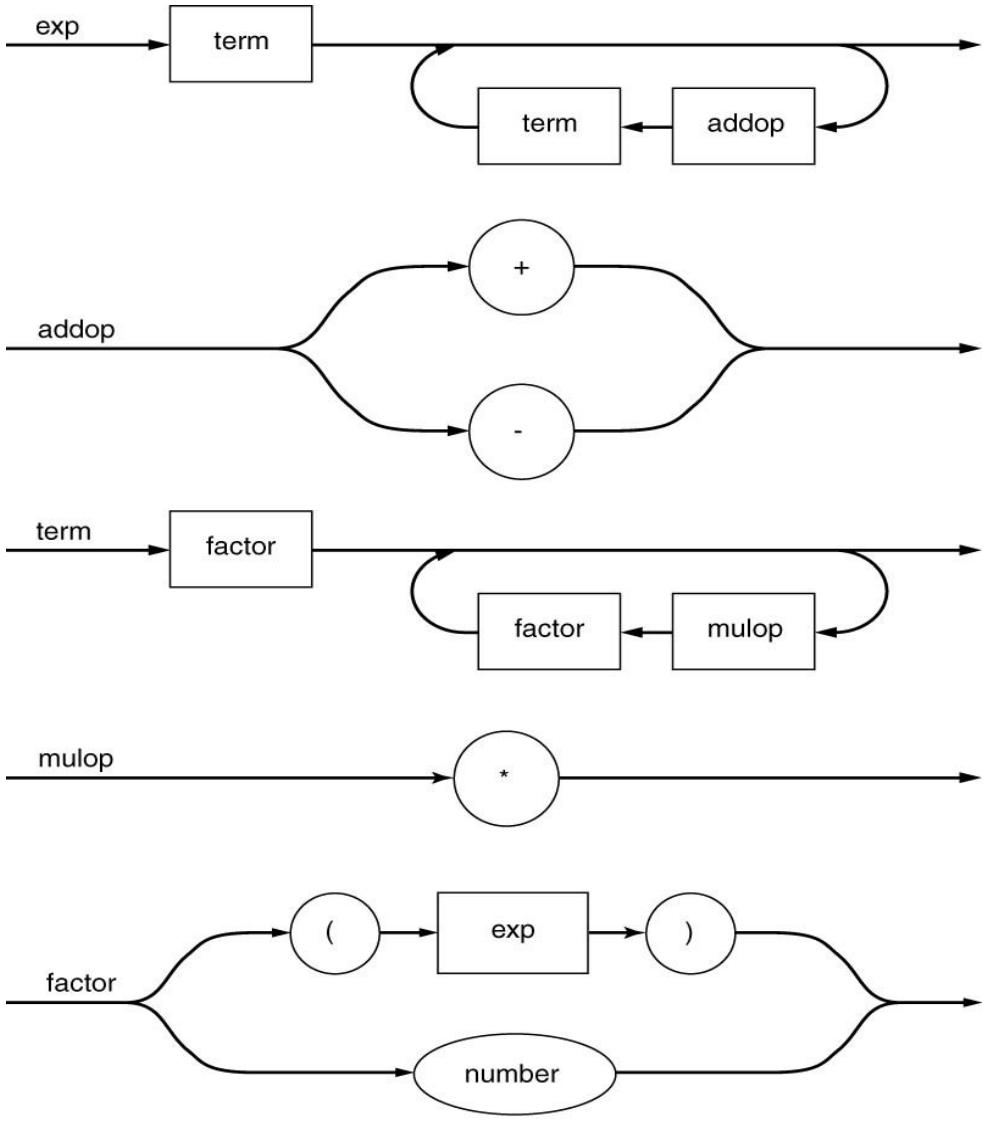
exp → *exp addop term* | *term*
addop → + | -
term → *term mulop factor* | *factor*
mulop → *
factor → (*exp*) | *number*



exp → *term* { *addop term* }
addop → + | -
term → *factor* {*mulop factor* }
mulop → *
factor → (*exp*) | *number*

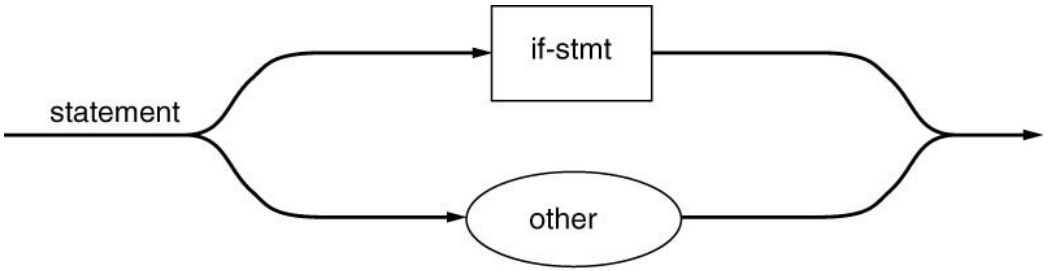
예 11: Syntax diagram (2)

exp → *term* { *addop term* }
addop → + | -
term → *factor* {*mulop factor* }
mulop → *
factor → (*exp*) | *number*

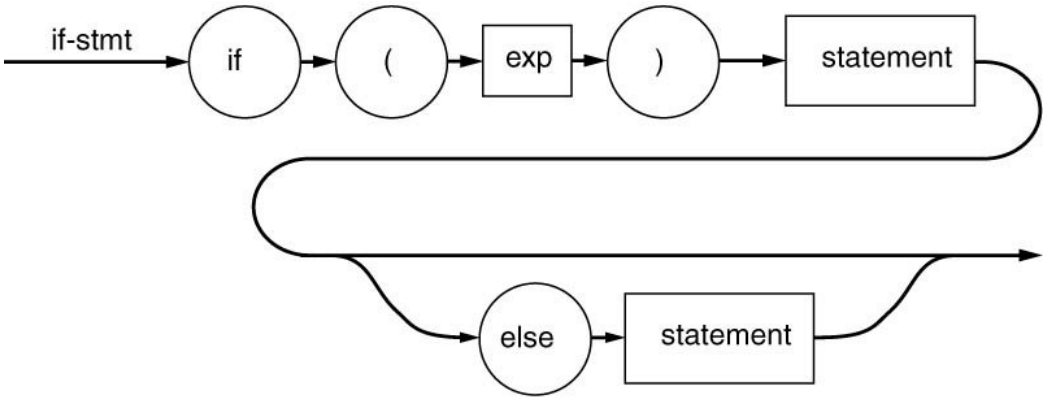


예 11: Syntax diagram (3)

statement → *if-stmt* | *other*
if-stmt → if (*exp*) *statement*
| if (*exp*) *statement* else *statement*
exp → 0 | 1



statement → *if-stmt* | *other*
if-stmt → if (*exp*) *statement*
[else *statement*]
exp → 0 | 1



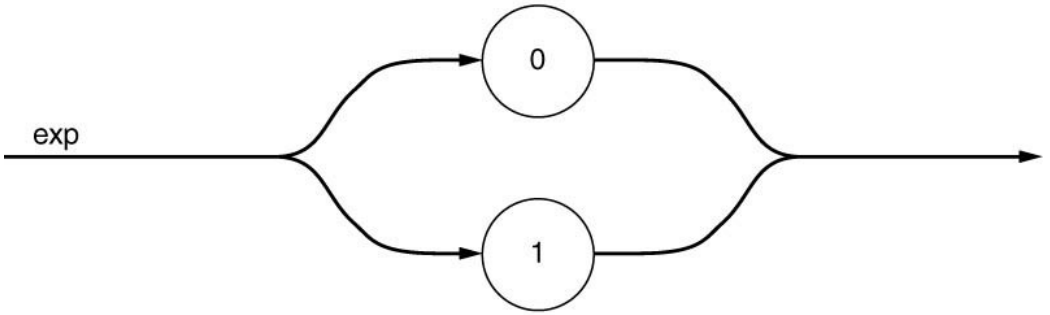


Table of Contents

A formal definition of CFG

- **4 *tuple* notation** $G = (T, N, P, S), 단 S \in N$
  - A set ***T*** of *Terminals* (*input symbols*)
  - A set ***N*** of *Nonterminals* ($T \cap N = \phi$)
  - A set ***P*** of *productions or grammar rules*
    - 생성 규칙의 왼쪽에는 non-terminal 한 개만 있다.

      $$A \rightarrow \alpha,\ \alpha \in (T \cup N)*$$
  - *A start symbol S.* $S \in N$

Classes of Grammars

0. An Unrestricted Grammar : no *restrictions* on the rewriting rules

$$\text{예} : SaB \rightarrow cS$$

1. A Context-Sensitive Grammar(CSG)

$$\alpha A \gamma \rightarrow \alpha \beta \gamma$$

$$\text{예} : SaB \rightarrow caB$$

2. A Context-Free Grammar(CFG)

$$A \rightarrow \alpha$$

$$\text{예} : A \rightarrow aABb$$

3. Regular Grammar

$$A \rightarrow aB \text{ 또는 } A \rightarrow a$$

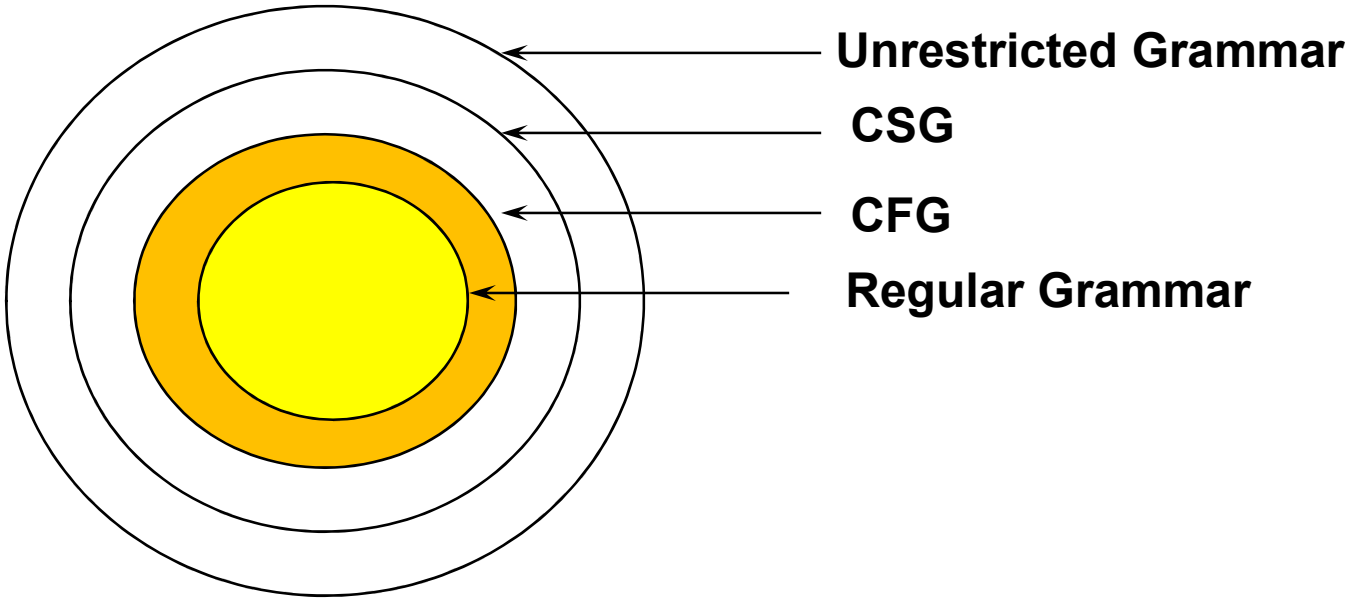
$$A \rightarrow Ba \text{ 또는 } A \rightarrow a$$

α, β, γ : 임의의 문자열
A, B, S : 비단말 기호
a, b, c : 단말기호

Right linear regular grammar

Left linear regular grammar

Classes of Grammars (*by N. Chomsky*)



Type 0 : Unrestricted
Type 1 : Context-Sensitive
Type 2 : Context-Free
Type 3 : Regular Grammar

정규 표현은 정규 문법

정규 표현(*Regular Expression*)

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
number = *digit* *digit* *

정규 문법 (*Regular Grammar*)

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
number → *number* *digit* | *digit*

A → A *a* | *a*

예 12: CSG, *not CFG*

P :

1. S → aSBC
2. S → ε
3. aB → ab
4. bB → bb
5. C → c
6. CB → CX
7. CX → BX
8. BX → BC

S => a***S***BC => aa***S***BCBC => aaB***CB***C
=> aaB***CX***C => aaB***BX***C => a***aB***BCC
=> aa***bB***CC => aabbC***C*** => aabb***C***c
=> aabbcc

$$L(G) = \{\varepsilon, abc, aabbcc, \cdots\}$$
$$= \{a^n b^n c^n, n \geq 0\}$$

예 13: 문법 종류

1. aSb → aAcBb	(Type 1. Context-Sensitive)
2. B → aA	(Type 3. Right Linear)
3. a → ABC	(Type 0. Unrestricted)
4. S → aBc	(Type 2. Context-Free)
5. Ab → b	(Type 1. Context-Sensitive)
6. AB → BA	(Type 0. Unrestricted)