http://pl.skuniv.ac.kr

컴파일러 입문

제 5 장

Context-Free 문법

목 차

5.1 서론

- **regular expression**: the lexical structure of tokens
  - recognizer : FA(⇒ scanner)
  - id = (l+_)(l+d+_)* , sc = "(a+\c)*"
- **CFG**: the syntactic structure of programming languages
  - recognizer : PDA(⇒ parser)
- 프로그래밍 언어의 구문 구조를 CFG로 표현할 경우의 장점:
  1. *간단*하고 *이해*하기 쉽다.
  2. CFG로부터 인식기를 자동으로 구성할 수 있다.
  3. 프로그램의 구조를 생성규칙에 의해 구분할 수 있으므로 *번역시*에 유용하다.

- **CFG의 form : N. Chomsky의 type 2 grammar**

  $A \rightarrow \alpha$, where $A \in V_N$ and $\alpha \in V^*$.

- ***recursive* construction**

  ex) E → E OP E | (E) | -E | id

  OP → + | – | * | /

  VN = { E, OP }

  VT = { (,), –, id, +, *, /}

  ex) <if_statement> → '**if**' <condition> '**then**' <statement>

  V_N : **<와 >사이에 기술된 symbol.**

  V_T : **' 와 ' 사이에 기술된 symbol.**

5.2 유도와 유도 트리

Text p.172

- **Derivation** : $\alpha_1 \Rightarrow \alpha_2$

  start symbol로부터 sentence를 생성하는 과정에서 nonterminal을 이 nonterminal로 시작되는 생성 규칙의 right hand side로 대치하는 과정.

  (1) $\Rightarrow$: derives in one step.

  $$\text{if } A \rightarrow \gamma \in P, \quad \alpha, \beta \in V^* \text{ then } \alpha A \beta \Rightarrow \alpha\gamma\beta.$$

  (2) $\overset{*}{\Rightarrow}$: derives in **zero or more** steps.

  1. $\forall \alpha \in V^*, \ \alpha \overset{*}{\Rightarrow} \alpha$
  2. $\text{if } \alpha \overset{*}{\Rightarrow} \beta \text{ and } \beta \Rightarrow \gamma \text{ then } \alpha \overset{*}{\Rightarrow} \gamma$

  (3) $\overset{+}{\Rightarrow}$: derives in **one or more** steps.

- **L(G) : the language generated by G**

  $= \{\omega \mid S \overset{*}{\Rightarrow} \omega, \omega \in V_T^*\}$

- **definition :**

  sentence : $S \overset{*}{\Rightarrow} \omega, \quad \omega \in V_T^*$ ♣ 모두 terminal로만 구성.

  sentential form : $S \overset{*}{\Rightarrow} \omega, \quad \omega \in V^*$.

- **Choosing a nonterminal being *replaced***

  - sentential form에서 어느 nonterminal을 선택할 것인가 ?

    $A \rightarrow \alpha$, where $\alpha \in V^*$.

    leftmost derivation: 가장 왼쪽에 있는 nonterminal을 대치해 나가는 방법.

    rightmost derivation: 가장 오른쪽에 있는 nonterminal을 대치.

- A derivation sequence $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$ is called a **leftmost derivation** if and only if α_{i+1} is obtained from α_i by applying a production to the leftmost nonterminal in α_i for all i, $0 \le i \le n-1$.

  $\alpha_i \Rightarrow \alpha_{i+1}$: 가장 왼쪽에 있는 nonterminal을 차례로 대치.

- **parse** : parser의 ***출력 형태*** 중에 한가지.
  - ***left parse*** : leftmost derivation에서 적용된 생성 규칙 번호.
    - **top-down** parsing
    - start symbol로부터 sentence를 생성
  - ***right parse*** : rightmost derivation에서 적용된 생성 규칙 번호의 **역순**.
    - **bottom-up** parsing
    - sentence로부터 nonterminal로 reduce되어 결국엔 start symbol로 reduce.

유도 트리

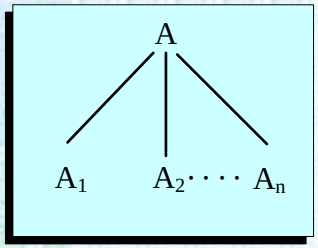
::= a graphical representation for derivations.

::= the hierarchical syntactic structure of sentences that is implied by the grammar.

- Definition : derivation tree

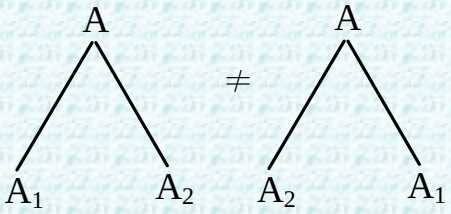
  CFG $G = (V_N, V_T, P, S)$ & $\omega \in V_T^*$ $\Rightarrow$ drawing a derivation tree.

  1. nodes: symbol of $V(V_N \cup V_T)$
  2. root node: S(start symbol)
  3. if $A \in V_N$, then a node A has at least one descendent.
  4. if $A \rightarrow A_1A_2...A_n \in P$, then A가 subtree의 root가 되고 좌로부터 $A_1, A_2, ..., A_n$가 A의 자 노드가 되도록 tree를 구성



- Nodes of derivation tree
  - internal(*nonterminal*) node ——— $\in V_N$
  - external(*terminal*) node ——— $\in V_T \cup \{\varepsilon\}$

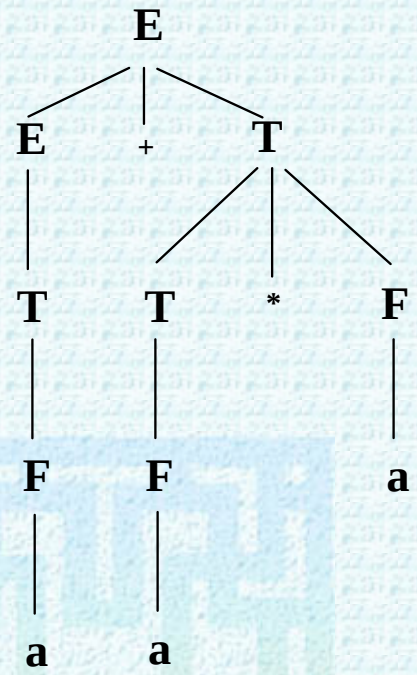
- ordered tree - child node들의 위치가 **순서**를 갖는 tree,
  따라서 derivation tree는 ordered tree이다.



예) G : E → E + T | T
T → T * F | F
F → (E) | a

ω : **a + a * a**

스트링 a + a * a**의 유도 트리:**



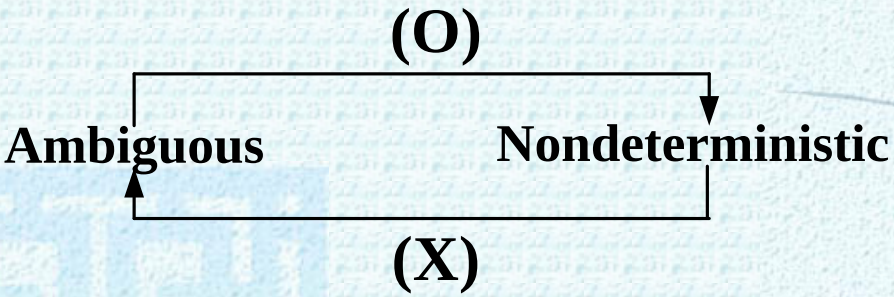
※ 각각의 유도 방법에 따라 derivation tree 모양은 변하지 않는다. 즉, 한 문장에 대한 tree 모양은 unique하다.

Ambiguous Grammar

- A context-free grammar G is ***ambiguous*** if and only if it produces more than one derivation trees for some sentence.

  → nondeterministic

- 설명: 같은 sentence를 생성하는 tree가 2개 이상 존재할 때 이 grammar를 ambiguous하다고 하며, 결정적인 파싱을 위해 nondeterministic한 grammar를 deterministic하게 변환해야 한다.



- "G: ambiguous 증명" → 하나의 sentence로 부터 2개 이상의 derivation tree 생성.

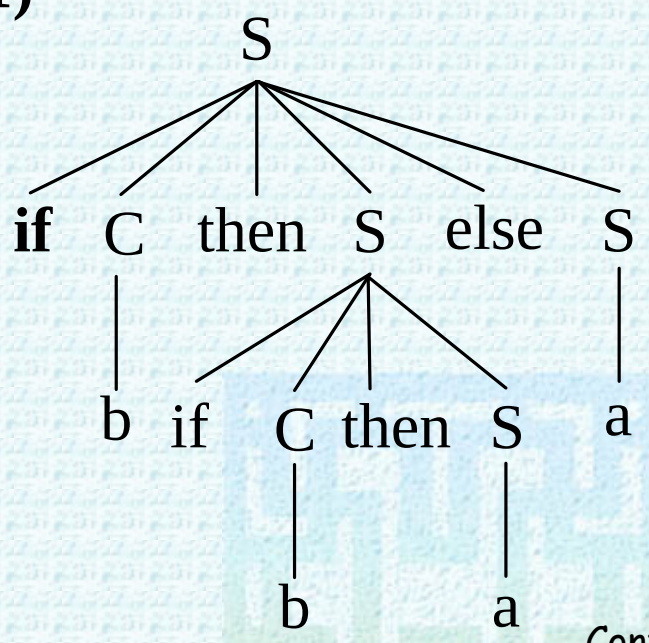
ex) **dangling else** problem:

G: S → if C then S else S | if C then S | a

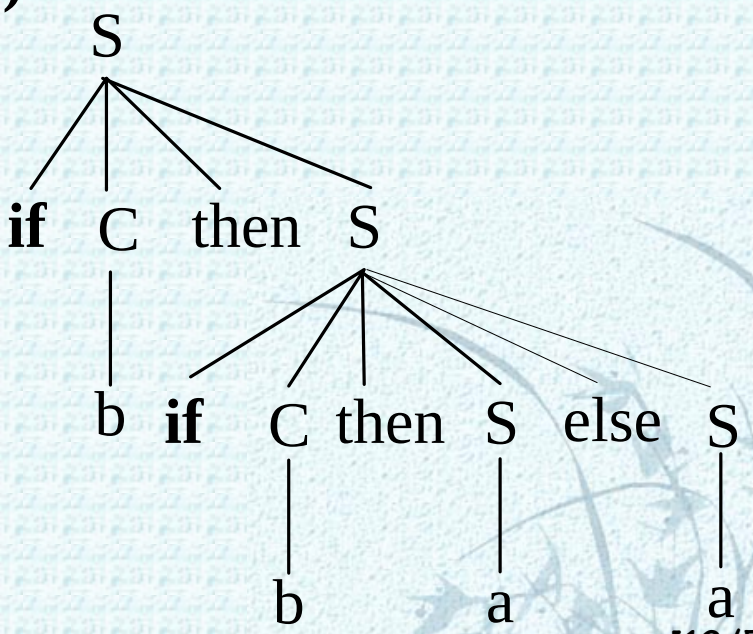
C → b

ω: **if b then if b then a else a**

1)

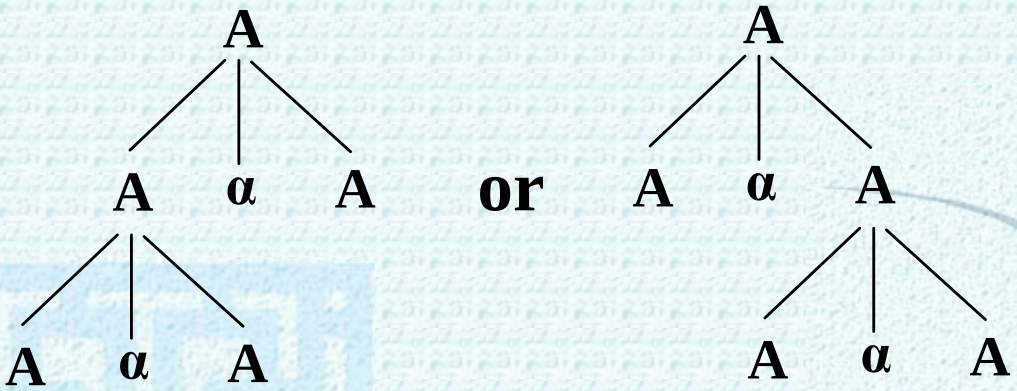


2)



※ **else** : 일반적으로 ***right*** associativity를 만족한다.
if 문장의 경우 자신과 가장 가까운 if와 결합함으로
두개의 트리 중 일반적으로 2)를 만족.

- In a more general form, the ambiguity appears when there is a production of the following form.
  - production form : $A \rightarrow A\alpha A$
  - sentential form : $A\alpha A\alpha A$
  - tree form :



ambiguous ⇒ unambiguous

1) 새로운 nonterminal을 도입해서 unambiguous grammar로 변환.

2) 이 과정에서, precedence & associativity 규칙을 이용.

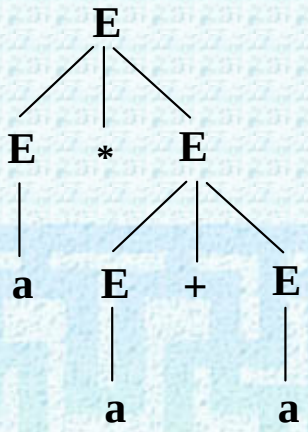
- nondeterministic ⇒ deterministic

예) $G: \; E \rightarrow E * E \mid E + E \mid a$

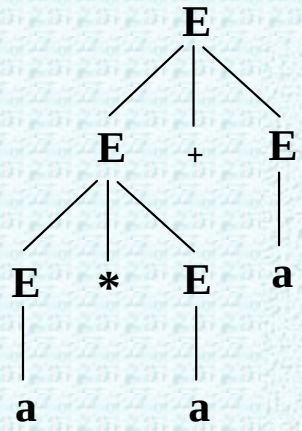
$\omega: \; a * a + a$

- precedence rule의 적용

1) + > *

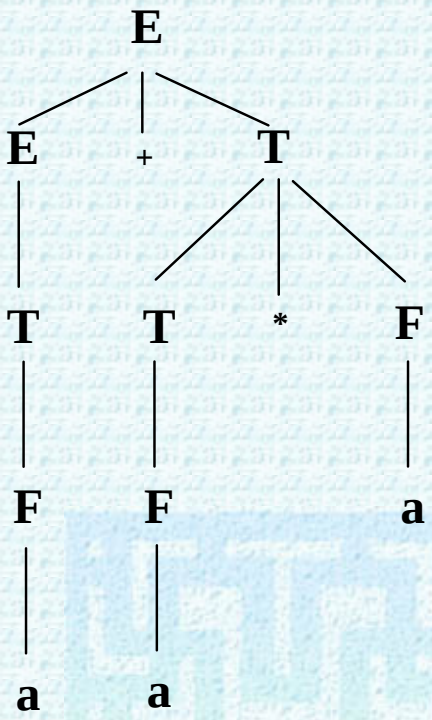


2) * > +



새로운 nonterminal의 도입

$$G: E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow a$$



※ 그런데, grammar의 ambiguity를 check할 수 있는 algorithm이 존재하지 않으며 unambiguous하게 바꾸는 formal한 방법도 존재하지 않는다.

- **unambiguous grammar로 바꾼 예:**

G : expression → expression + term
| expression - term
| term

term → term * factor
| term / factor
| factor

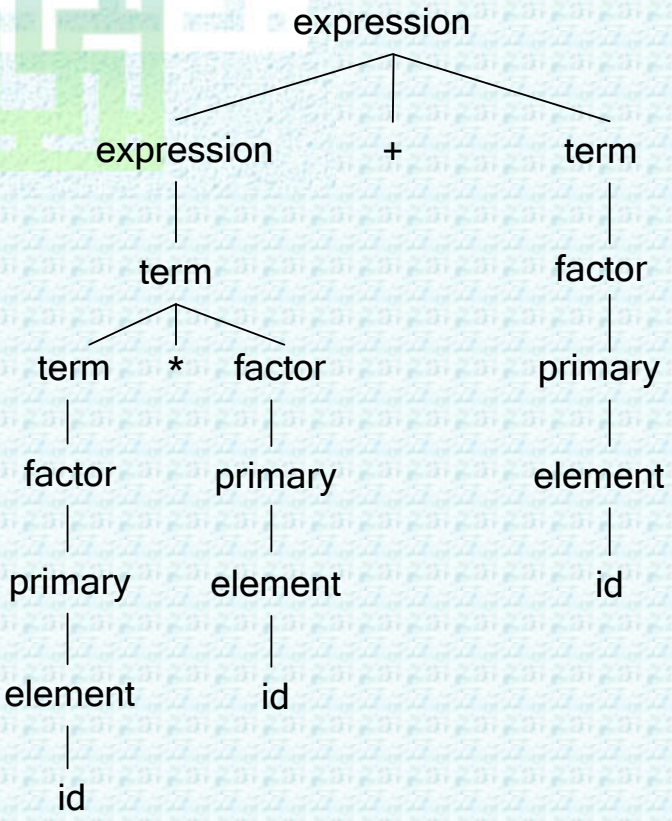
factor → primary ↑ factor
| primary

primary → - primary
| element

element → (exp)
| id

- **derivation tree가 하나이므로 위 grammar는 unambiguous하다.**

- id * id + id의 derivation tree:



- derivation tree가 하나 이므로 위 grammar는 unambiguous하다.

5.4 CFG 표기법

☞ BNF(Backus-Naur Form), EBNF(Extended BNF), Syntax Diagram

- **BNF**
  - 특수한 meta symbol을 사용하여 프로그래밍 언어의 구문을 명시하는 표기법.
  - **meta symbol** : 새로운 언어의 구문을 표기하기 위하여 도입된 심벌들.

nonterminal symbol	< >
→	::= (치환)
nonterminal symbol의 rewriting	¦ (또는)

  - terminal symbol : ' '
  - grammar symbol : $V_N \cup V_T$

예1) $V_N = \{S, A, B\}$, $V_T = \{a, b\}$

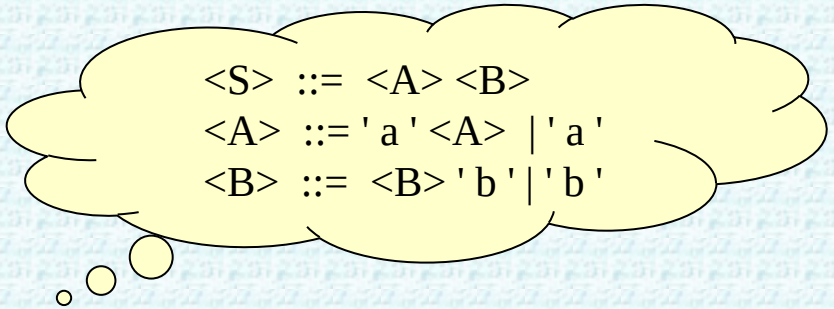
$P = \{S \rightarrow AB, A \rightarrow aA, A \rightarrow a, B \rightarrow Bb, B \rightarrow b\}$

BNF 표현:

<S> ::= <A> <B>

<A> ::= a <A> | a

<B> ::= <B> b | b



예2) Compound statement

BNF 표현:

<compound_statement> ::= '{'<statement_list> '}'

<statement_list> ::= <statement_list> <statement>

| <statement>

- Extended BNF(EBNF)
  - 특수한 의미를 갖는 meta symbol을 사용하여 반복되는 부분이나 선택적인 부분을 간결하게 표현.
  - meta symbol

반복되는 부분(repetitive part):	{ }
선택적인 부분(optional part):	[ ]
괄호와 택일 연산자(alternative):	(¦)

예1) <compound_statement> ::= '{' <statement> {<statement>} '}'

예2) <if-st> ::= 'if' '(' <expression> ')' <statement> ['else' <statement>]

예3) <exp> ::= <exp> + <exp> ¦ <exp> - <exp> ¦
<exp> * <exp> ¦ <exp> / <exp>

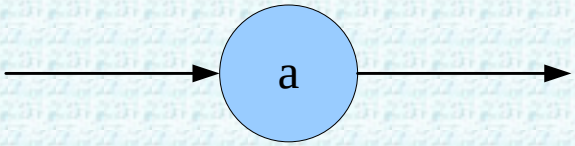
<exp> ::= <exp> (+ ¦ – ¦ * ¦ /) <exp>

- **Syntax diagram**
  - 초보자가 쉽게 이해할 수 있도록 구문 구조를 도식화하는 방법
  - syntax diagram에 사용하는 그래픽 아이템:

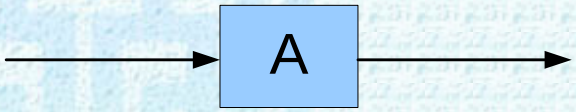
원	:	terminal symbol
사각형	:	nonterminal symbol
화살표	:	흐름 경로

- syntax diagram을 그리는 방법:

1. terminal a



2. nonterminal A

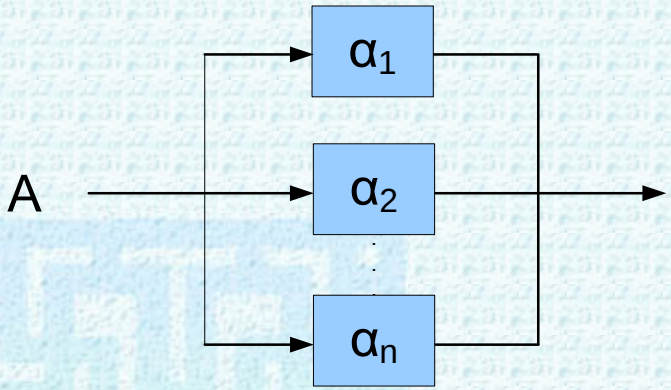


3. A ::= $X_1X_2 \ldots X_n$

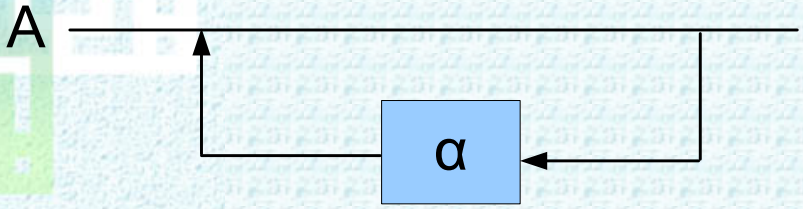
(1) X_i가 nonterminal인 경우:

(2) X_i가 terminal인 경우:

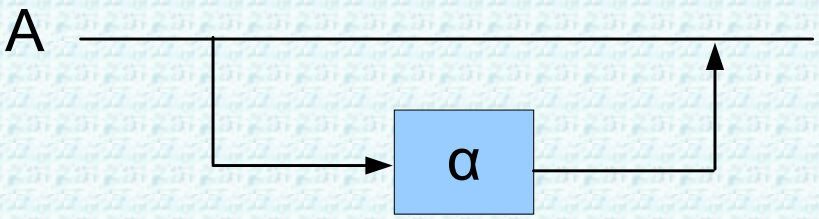
4. A ::= $\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$



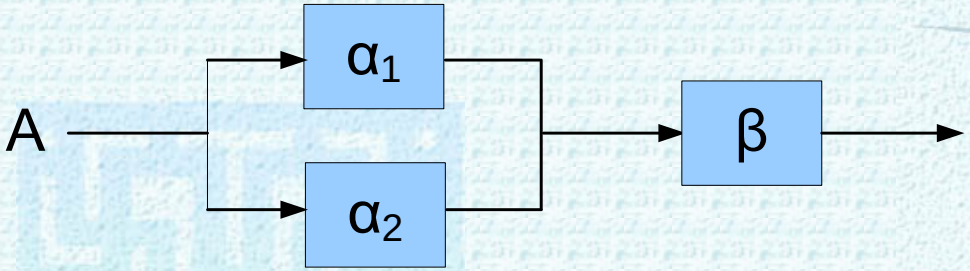
5. EBNF A ::= {α}



6. EBNF A ::= [α]



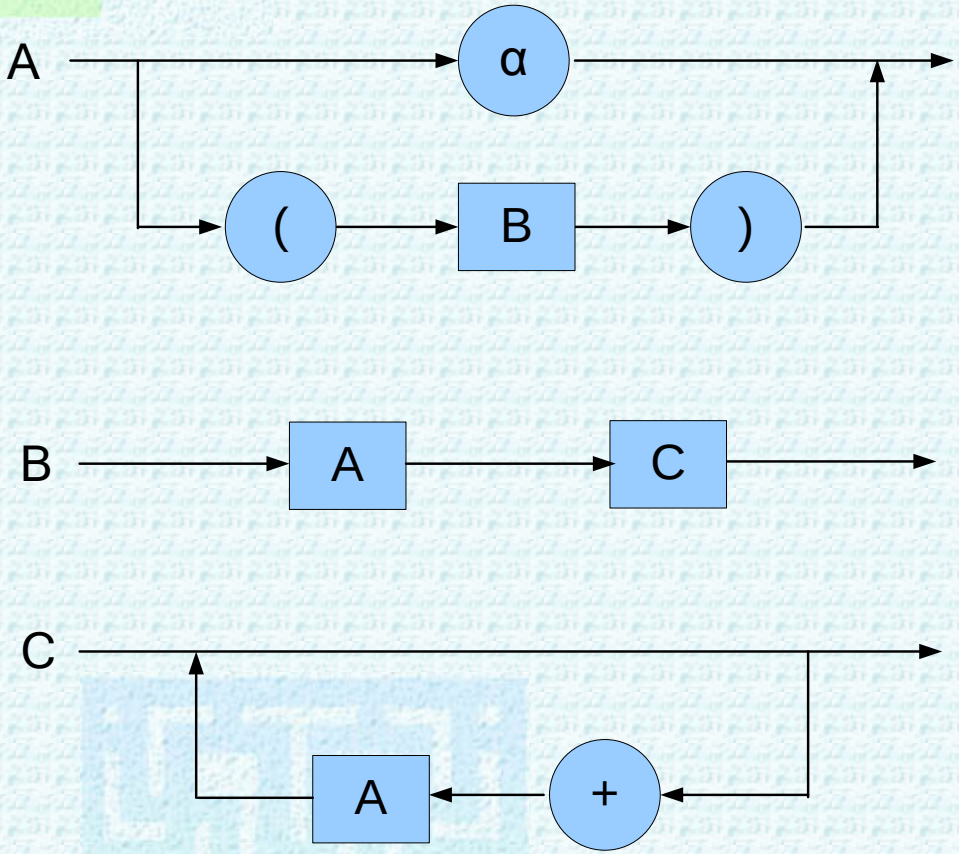
7. EBNF A ::= $(\alpha_1 \mid \alpha_2)\beta$

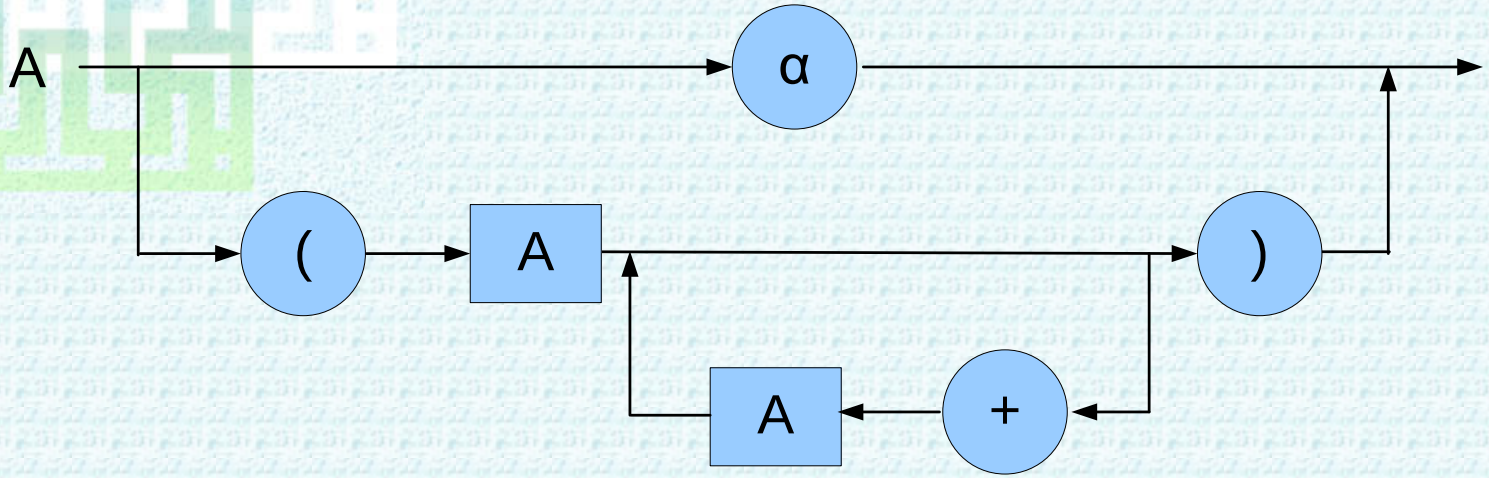


(예) A ::= a ¦ (B)

B ::= AC

C ::= {+A}




A
α
(
A
)
A
+