Chapter 5

문맥 자유 문법과 파싱 알고리즘 - **Part III**

목차

01 문맥 자유문법

02 파스 트리

03 모호한 문법

04 문법 변환

05 푸시다운 오토마타

Parsers and Recognizers

Recognizer

- A compiler must determine if $x \in L(G)$
  - given a grammar G and an input string x

Parser

- We must determine not only ***the string's validity***, but also **its structure**(or *parse tree*).

Two approaches to parsing

Grammar

1.	Program	→	*begin*	Stmts	*end*	$
2.	Stmts	→	Stmt	;	Stmts	
3.		|	λ			
4.	Stmt	→	*simplestmt*			

Input string

begin simplestmt ; simplestmt ; end $

Top-Down Parsing(1/7)

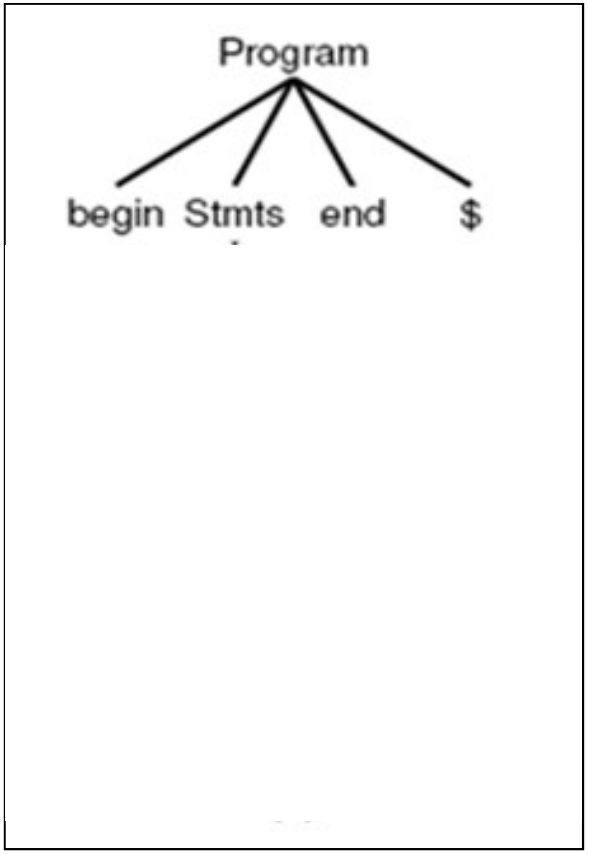
Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

Program
=> begin Stmts end $



Top-Down Parsing(2/7)

Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

Which one to select?

Program
=> begin Stmts end $
=> begin Stmt ; Stmts end $

Program
=> begin Stmts end $
=> begin λ end $

How do you know that?

Program
=> begin Stmts end $
⇒ begin *Stmt* …
⇒ begin *simplestmt* …

Top-Down Parsing(2/7)

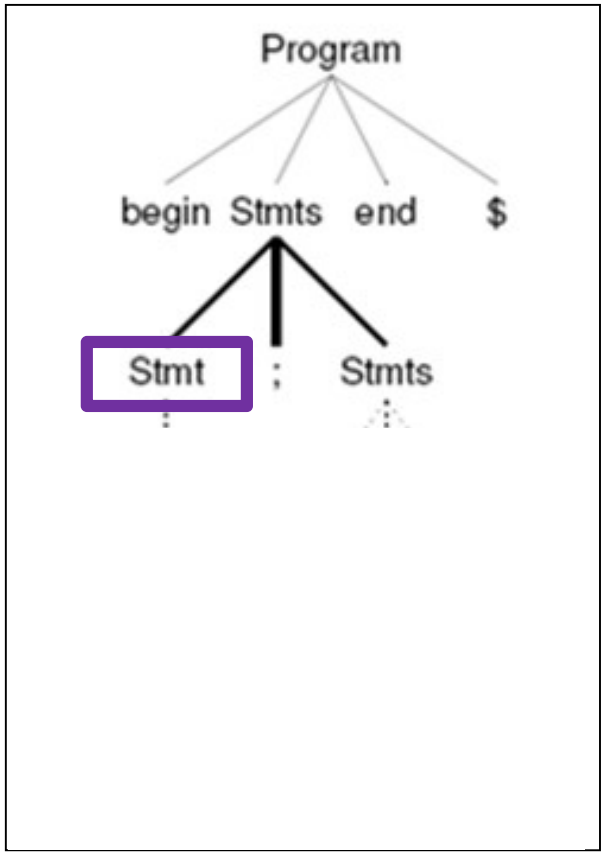
Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

Program
=> begin Stmts end $
=> begin Stmt ; Stmts end $



Top-Down Parsing(3/7)

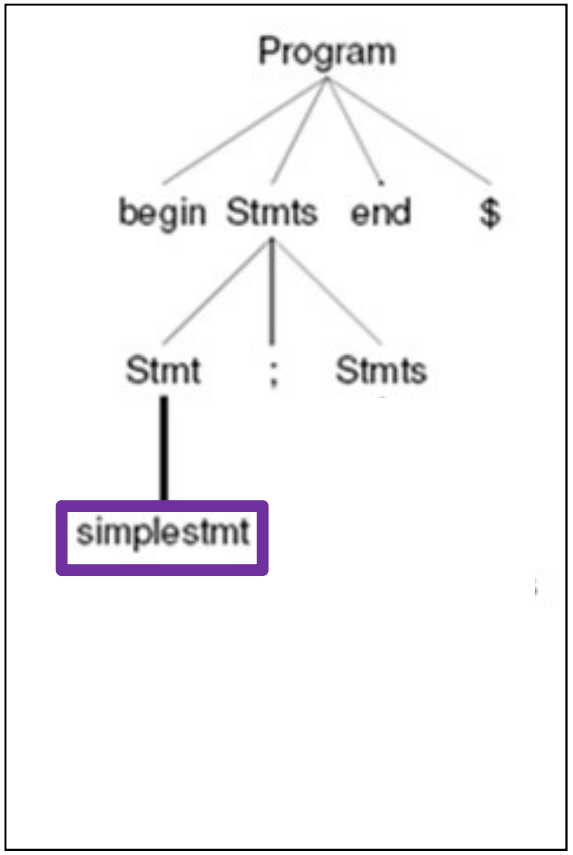
Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

Program
=> begin Stmts end $
=> begin Stmt ; Stmts end $
=> begin simplestmt ; Stmts end $



Top-Down Parsing(4/7)

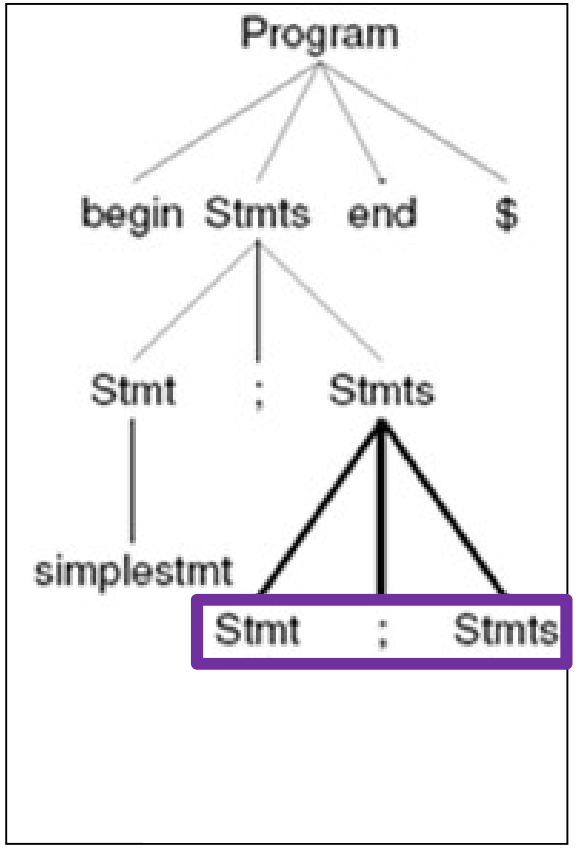
Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

Program
=> begin Stmts end $
=> begin Stmt ; Stmts end $
=> begin simplestmt ; Stmts end $
=> begin simplestmt ; Stmt ; Stmts end $



Top-Down Parsing(5/7)

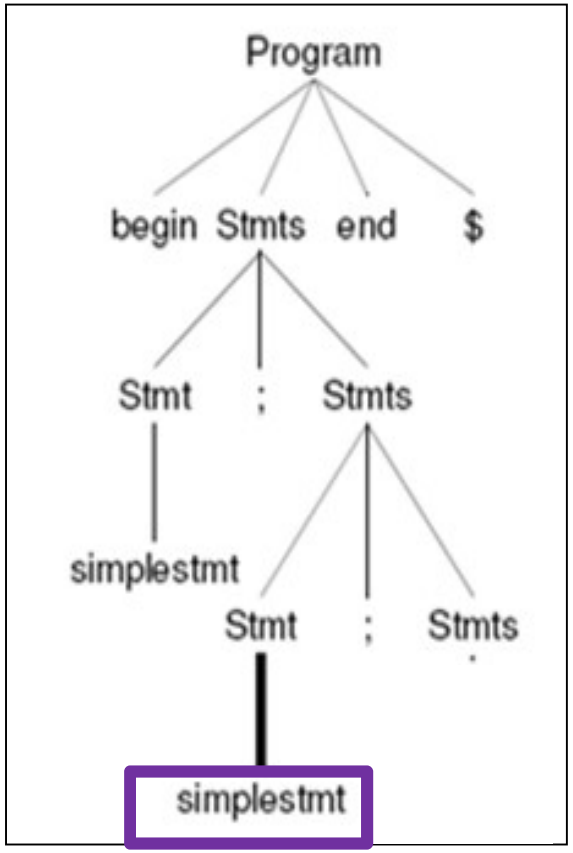
Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

Program
=> begin Stmts end $
=> begin Stmt ; Stmts end $
=> begin simplestmt ; Stmts end $
=> begin simplestmt ; Stmt ; Stmts end $
=> begin simplestmt ; simplestmt ; Stmts end $



Top-Down Parsing(6/7)

Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

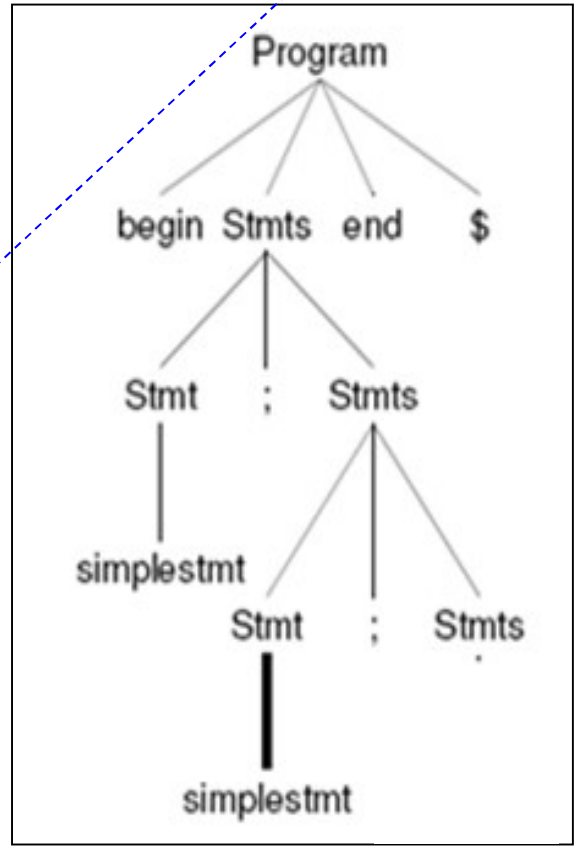
Input string

begin simplestmt ; simplestmt ; end $

Which one to select?

Program
=> begin Stmts end $
=> …
=> begin simplestmt ; simplestmt ; Stmts end $

Stmts 가 사라져야만(즉, empty string)
"end" 스트링이 나옴



Top-Down Parsing(7/7)

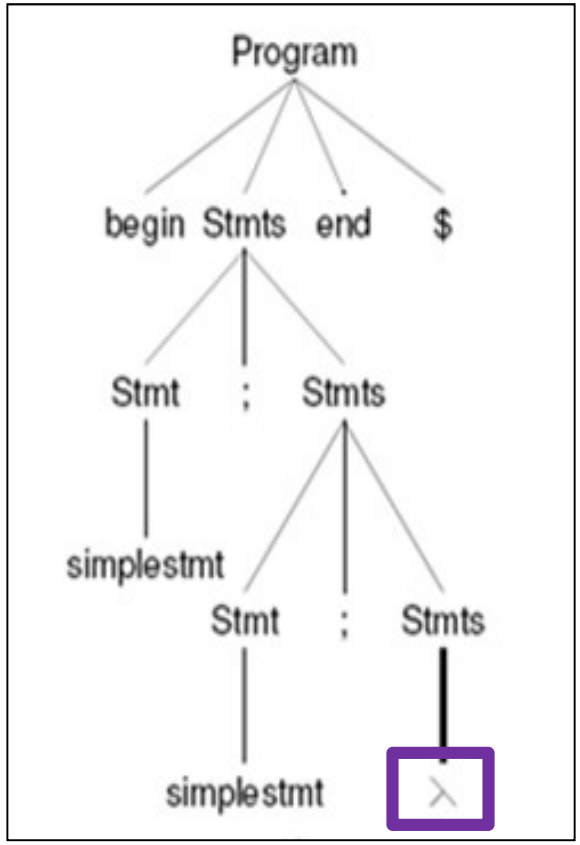
Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

Program
=> begin Stmts end $
=> begin Stmt ; Stmts end $
=> begin simplestmt ; Stmts end $
=> begin simplestmt ; Stmt ; Stmts end $
=> begin simplestmt ; simplestmt ; Stmts end $
=> begin simplestmt ; simplestmt ; end $



Bottom-Up Parsing(1/8)

Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

생성 규칙의 오른쪽에 놓인 기호들을
왼쪽 Nonterminal로 바꿈

Which one to select?

begin *simplestmt* ; simplestmt ; end $

begin simplestmt ; *simplestmt* ; end $

Bottom-Up Parsing(2/8)

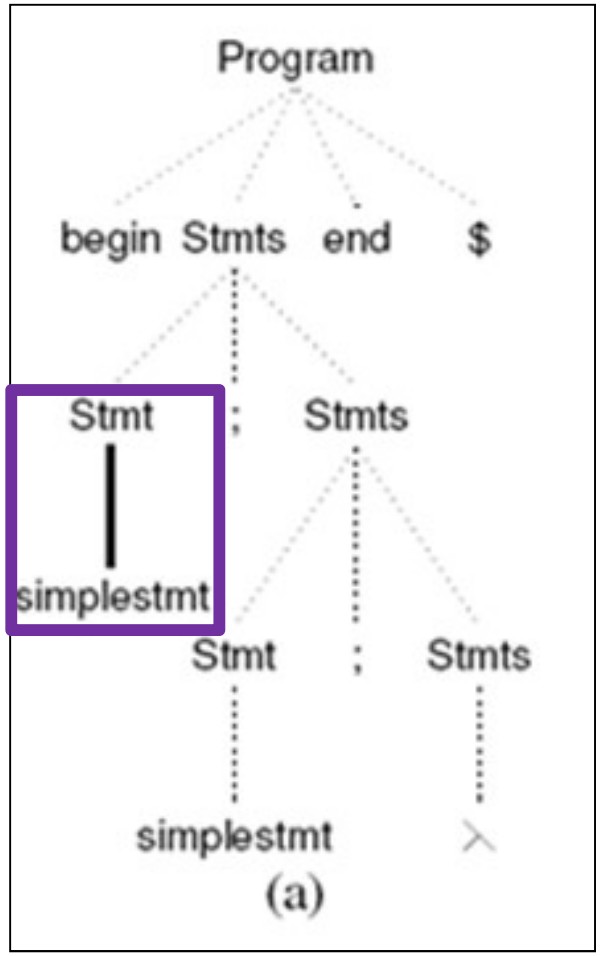
Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

begin simplestmt ; simplestmt ; end $
=> begin Stmt ; simplestmt ; end $



(a)

Bottom-Up Parsing(3/8)

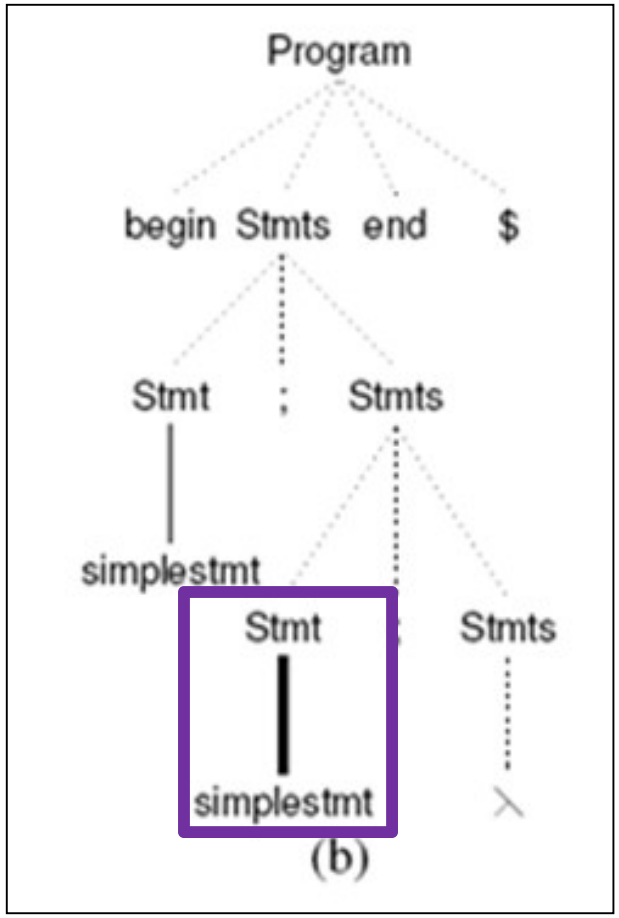
Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

begin simplestmt ; simplestmt ; end $
=> begin Stmt ; simplestmt ; end $
=> begin Stmt ; Stmt ; end $



Bottom-Up Parsing(4/8)

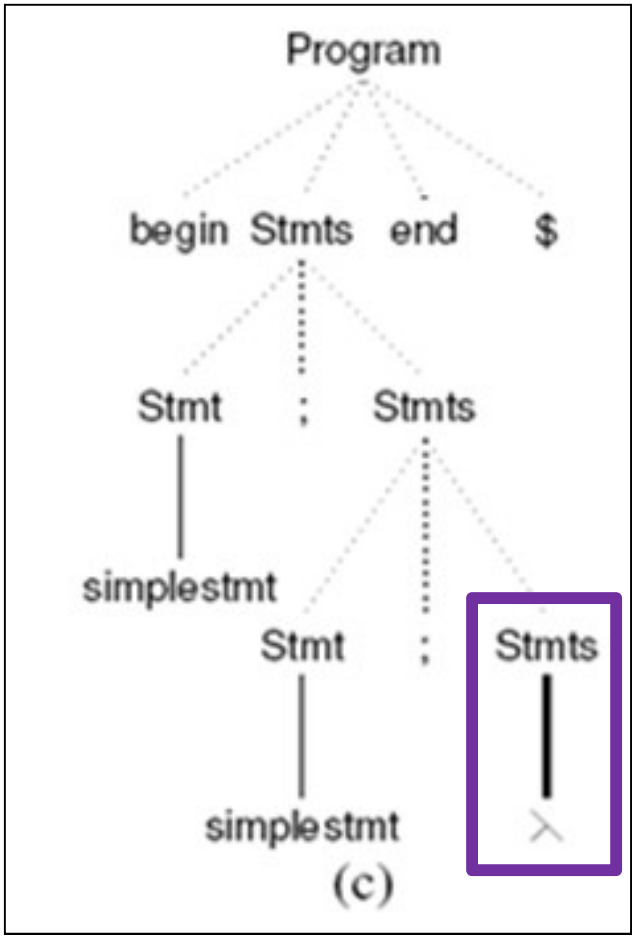
Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

begin simplestmt ; simplestmt ; end $
=> begin Stmt ; simplestmt ; end $
=> begin Stmt ; Stmt ; end $
=> begin Stmt ; Stmt ; Stmts end $



Bottom-Up Parsing(5/8)

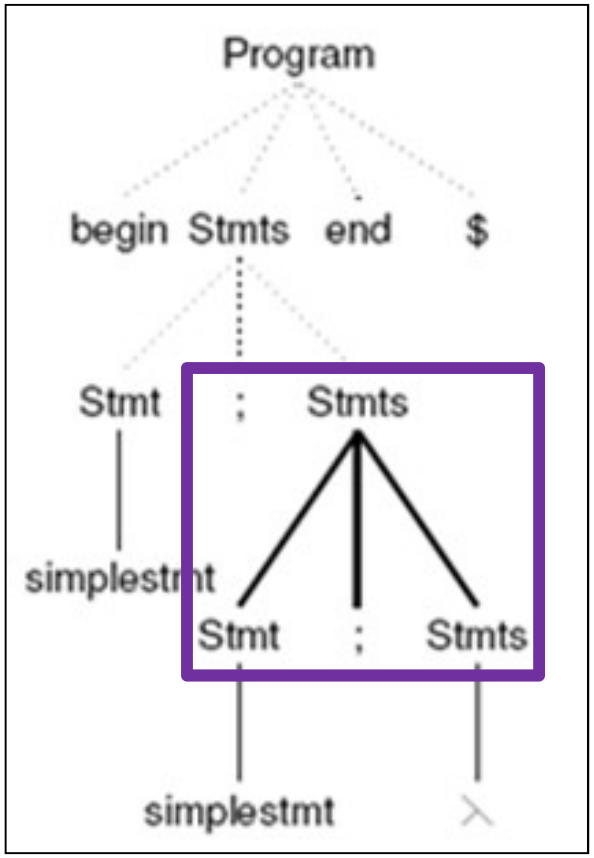
Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

begin simplestmt ; simplestmt ; end $
=> begin Stmt ; simplestmt ; end $
=> begin Stmt ; Stmt ; end $
=> begin Stmt ; Stmt ; Stmts end $
=> begin Stmt ; Stmts end $



Bottom-Up Parsing(6/8)

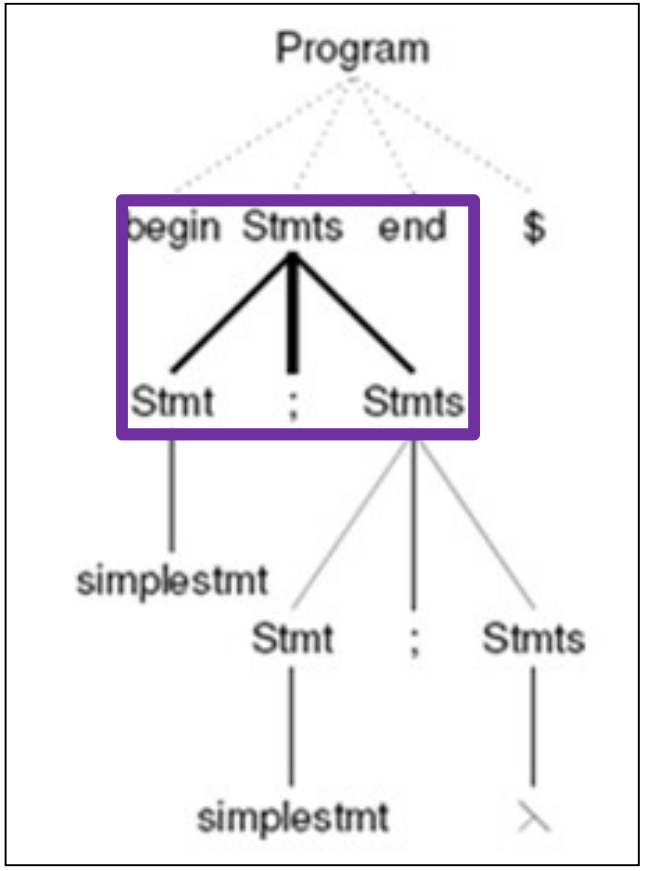
Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

begin simplestmt ; simplestmt ; end $
=> begin Stmt ; simplestmt ; end $
=> begin Stmt ; Stmt ; end $
=> begin Stmt ; Stmt ; Stmts end $
=> begin Stmt ; Stmts end $
=> begin Stmts end $



Bottom-Up Parsing(7/8)

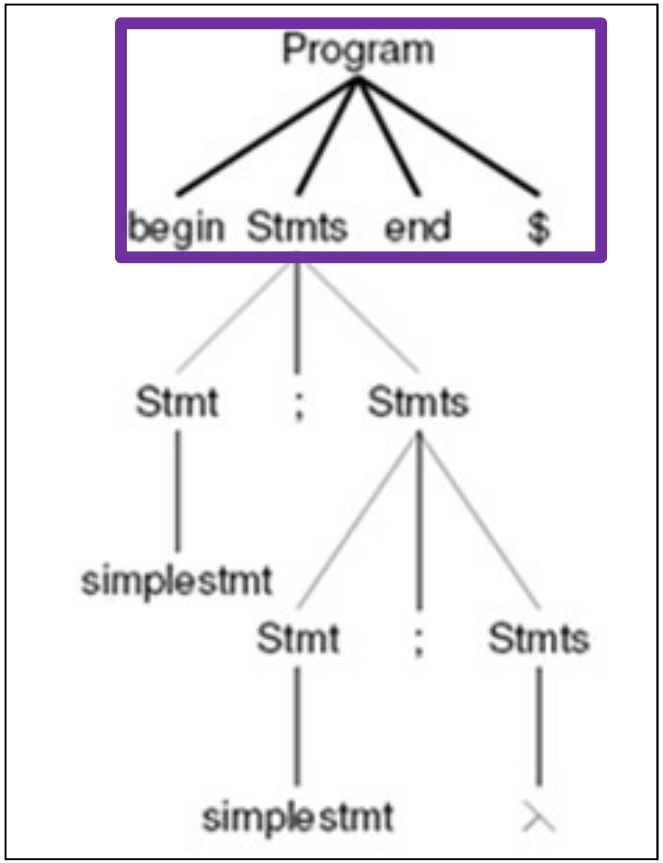
Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

Input string

begin simplestmt ; simplestmt ; end $

begin simplestmt ; simplestmt ; end $
=> begin Stmt ; simplestmt ; end $
=> begin Stmt ; Stmt ; end $
=> begin Stmt ; Stmt ; Stmts end $
=> begin Stmt ; Stmts end $
=> begin Stmts end $
=> Program



Bottom-Up Parsing(8/8)

Grammar

Program → *begin* Stmts *end* $
Stmts → Stmt ; Stmts
| λ
Stmt → *simplestmt*

begin simplestmt ; simplestmt ; end $
=> begin Stmt ; simplestmt ; end $
=> begin Stmt ; Stmt ; end $
=> begin Stmt ; Stmt ; Stmts end $
=> begin Stmt ; Stmts end $
=> begin Stmts end $
=> Program

Can you see that?

Program
⇒ begin Stmts end $
⇒ begin Stmt ; Stmts end $
⇒ begin Stmt ; Stmt ; Stmts end $
⇒ begin Stmt ; Stmt ; end $
⇒ begin Stmt ; simplestmt ; end $
⇒ begin simplestmt ; simplestmt ; end $

Two approaches to parsing

Top-Down

- *L*eftmost parse ; ***LL or LL(k) or LL(1)***
  - The 1st character *L* : the token sequence is processed from left to right
  - ***k*** : the number of **lookahead symbols** that the parser may consult to make parsing choices

Bottom-Up

- A post-order tree traversal of the parse tree
- *R*ightmost parse ; ***LR or LR(k) or LR(1)***

전처리(Preprocessing)

- 하향식(**Top-down**) 구문 분석을 위해서는 전처리가 필요
  - 좌 인수분해 (*factoring common prefixes*)
    - 공통 부분이 있는 생성규칙들은 공통 부분을 묶은 생성규칙으로 변환
  - 좌순환 제거(*eliminate left recursion*)
    - 좌순환 생성규칙은 우순환 생성규칙으로 변환

- 상향식(**Bottom-up**) 구문 분석에서는 위와 같은 전처리가 필요하지 않음.

Left Factoring(좌 인수분해)

stmt → `if` ***expr*** `then` ***stmt*** `else` ***stmt***
| `if` ***expr*** `then` ***stmt***

- **2개의 rule이 똑같은 내용으로 시작하기 때문에 *lookahead* 1개만으로는 어느 구문인지 알 수 없다.**
  - 공통 부분을 하나로 합치면, 2개의 구문이 달라지는 부분에서 *lookahead* 를 이용한 구문 선택이 가능

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

새로운 Nonterminal A' 추가

Left Factoring : 예 16

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \quad \Rightarrow \quad \begin{aligned} A &\to \alpha A' \\ A' &\to \beta_1 \mid \beta_2 \end{aligned}$$

$$\boxed{\begin{aligned} S &\to iEtS \mid iEtSeS \mid a \\ E &\to b \end{aligned}} \quad \Rightarrow \quad \boxed{\begin{aligned} S &\to iEtSS' \mid a \\ S' &\to eS \mid \varepsilon \\ E &\to b \end{aligned}}$$

여기서

$$\begin{aligned} A &= S \\ \alpha &= iEtS \\ \beta_1 &= \varepsilon \\ \beta_2 &= eS \end{aligned}$$

Left Factoring : 예 17

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

stmt → s

stmt-sequence → *stmt stmt-seq'*

stmt-seq' → ; *stmt-sequence* | ε

exp → *term* + *exp* | *term*

exp → *term exp'*

exp' → + *exp* | ε

Left Factoring : 예 18

statement → *assign-stmt* | *call-stmt* | *other*
assign-stmt → *identifier* := *exp*
call-stmt → *identifier* (*exp-list*)

statement → identifier statement' | other
statement' → := exp | (exp-list)

statement →
identifier := *exp* | *identifier* (*exp-list*) | *other*

left recursion(좌 순환)

expr → ***expr*** + ***term*** / ***term***

일반형 $A \rightarrow A\,\alpha \mid \beta$

예: A = *expr*, α = + *term*, β = *term*

- ***lookahead*** **symbol** 은 생성 규칙의 **RHS**의 ***terminal*** 과 **match** 될 때만 바뀐다.
  - 좌 순환을 갖는 생성규칙은 Nonterminal이 맨 처음 나타나므로
  - matching 은 일어나지 않고 불필요한 유도만 자꾸 발생한다.

- **[예]** 아래 문법을 사용해 **id + id * id**를 좌단 유도로 생성해보자.

  E → E + T | T

  T → T * F | F

  F → (E) | id

  ➔ E ⇒ **E** + T ⇒ **E** + T + T ⇒ **E** + T + T ⇒ …

Eliminate left recursion

Left recursion

$$A \longrightarrow A\,a$$
$$A \longrightarrow b$$

Right recursion

$$A \longrightarrow bA'$$
$$A' \longrightarrow aA'$$
$$A' \longrightarrow \epsilon$$

Nonterminal A’ 추가

$$\mathcal{L}(A) = \{b, ba, baa, baaa, baaaa, \ldots\}$$

좌 순환 *versus* 우 순환

left-recursion

$A \rightarrow A\,\alpha \mid \beta$

$expr \rightarrow expr + term \mid term$

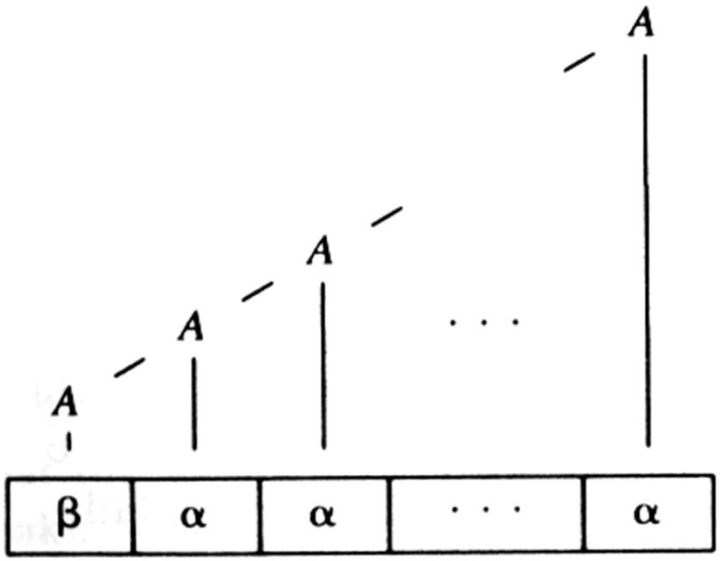
right-recursion

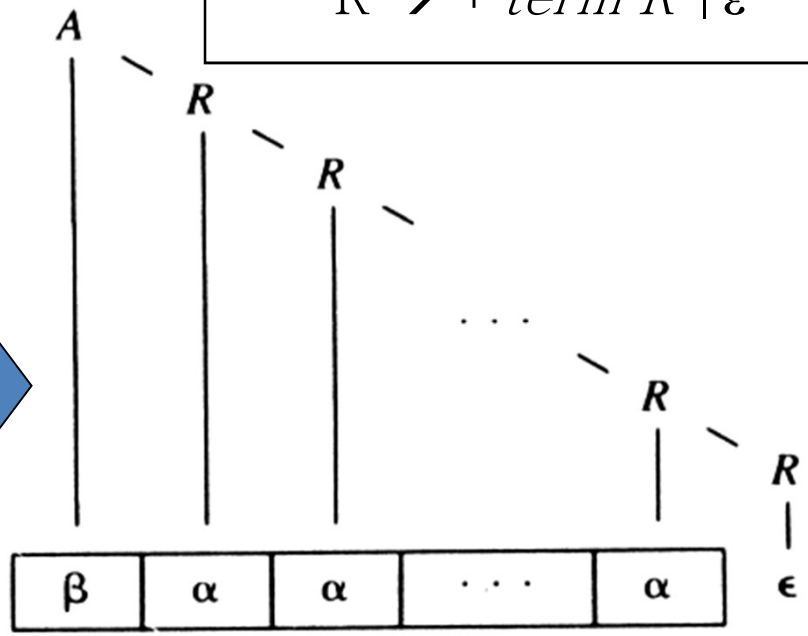
$A \rightarrow \beta\ R$

$R \rightarrow \alpha\, R \mid \varepsilon$

$expr \rightarrow term\ R$

$R \rightarrow +\ term\ R \mid \varepsilon$





Eliminate left recursion : 예 19

$$E \longrightarrow E + E$$
$$E \longrightarrow \text{id}$$

$$\boxed{\begin{array}{lcl} A & \longrightarrow & A\,a \\ A & \longrightarrow & b \end{array}}$$

$A = E$

$a = + E$

$b = id$

$$\Downarrow$$

$$E \longrightarrow \text{id}\ E'$$
$$E' \longrightarrow + E\ E'$$
$$E' \longrightarrow \epsilon$$

$$\boxed{\begin{array}{lcl} A & \longrightarrow & bA' \\ A' & \longrightarrow & aA' \\ A' & \longrightarrow & \epsilon \end{array}}$$

Eliminate left recursion : 예 20

■ simple immediate left recursion

exp → *exp addop term* | *term*

exp → term exp′

exp′ → addop term exp′ | ε

■ General immediate left recursion

exp → *exp* + *term* | *exp* - *term* | *term*

exp → term exp′

exp′ → + term exp′ | - term exp′ | ε

Eliminate left recursion : 예 21

S → Aa | b

A → Ac | Sd | e

- S에 관한 생성규칙에서 A 대신 A의 생성규칙을 대입하면

  S → Aa | b ➔ **S → Sda** | A**ca** | **ea | b**

- S도 좌순환 생성규칙을 갖는다.
  - S를 우순환 규칙으로 변환하더라도 여전히 A는 좌순환 규칙을 갖고 있다.

Eliminate left recursion : 예 22

S → Aa | b

A → Ac | Sd | e

- A에 관한 생성규칙에서 S 대신 S의 생성 규칙을 대입

  A → Ac | Sd | e ➔ A → **Ac** | **Aad** | **bd** | **e**

- Nonterminal **A** 의 좌순환 규칙을 우순환 규칙으로 변환

  S → Aa | b

  A → bd**A'** | e**A'**

  A' → c**A'** | ad**A'** | ε

Eliminate left recursion : 예 23

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

```
exp → term exp'
exp' → addop term exp' | ε
addop → + | -
term → factor term'
term' → multop factor term' | ε
multop → *
factor → ( exp ) | number
```

Pushdown Automata (1/2)

- **$M = (Q, \Sigma, T, \delta, q_0, z_0, F)$**

  Q : 상태 집합

  Σ : 입력 기호 집합

  T : stack 기호 집합

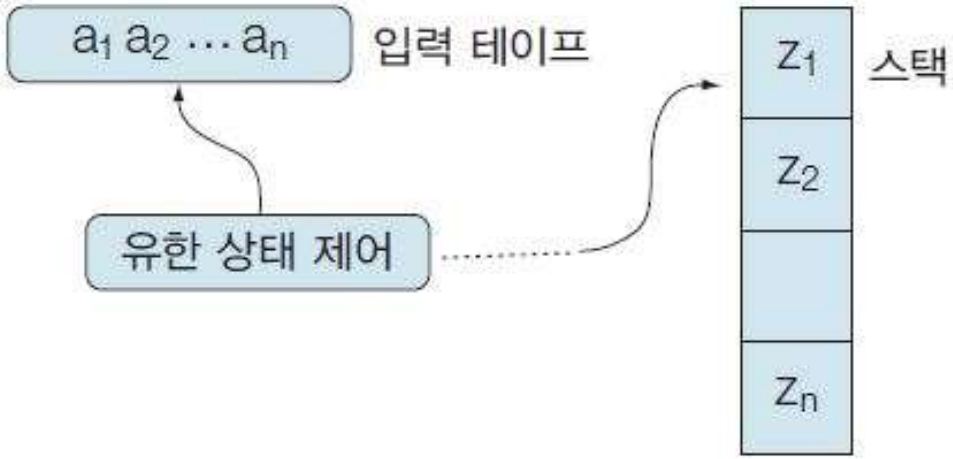
  $q_0 \in Q$: 시작 상태

  $z_0 \in T$: stack의 시작기호

  $F \subseteq Q$: 종결 상태 집합

  δ : mapping function

  $$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times T \rightarrow Q \times T^*$$



Pushdown Automata (2/2)

- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times T \rightarrow Q \times T^*$
  - δ (**q**, a, z) = {(**p**, r)}
    - 상태 **q** 에서 입력 기호가 **a** 이고 stack의 top에 저장된 기호가 ***z*** 라면
      - 상태 **p**로 전이
      - stack의 top에 있는 기호 **z**를 삭제(pop)하고, 기호 **r**을 삽입(push)
      - 만약 r=ε 이면, stack의 top에 있는 기호 **z**만 삭제(pop)
    - 연산 직후 입력 테이프 포인터는 오른쪽으로 한 칸 이동
      - 다음 입력 문자를 읽어 온다.