http://pl.skuniv.ac.kr

컴파일러 입문

제 6 장

구문 분석

목 차

6.1 구문 분석 방법

6.2 구문 분석기의 출력

6.3 Top-down 방법

6.4 Bottom-up 방법

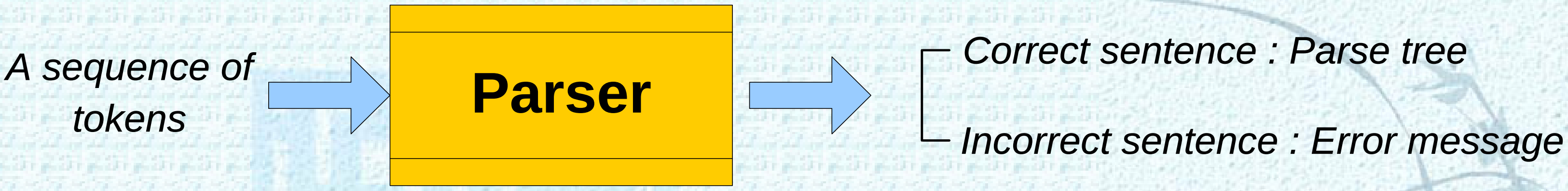
6.1 구문 분석 방법

Text p.233

- How to **check** whether an input string is a sentence of a grammar and how to **construct** a parse tree for the string.

Parsing : $\omega \overset{?}{\in} L(G)$

- A **Parser** for grammar G is a program that takes as input a string ω and produces as output **either** a parse tree(or derivation tree) for ω, if ω is a sentence of G, **or** an error message indicating that ω is not sentence of G.



A sequence of tokens

Parser

Correct sentence : Parse tree

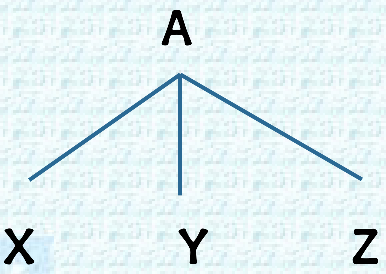
Incorrect sentence : Error message

- **Two basic types of parsers for context-free grammars**

① ***Top down*** - starting with the root and working down to the leaves. recursive descent parser, predictive parser.

② ***Bottom up*** - beginning at the leaves and working up the root. precedence parser, shift-reduce parser.

ex) $\overset{T}{A} \rightarrow \overset{N}{XYZ}$



reduce

expand

bottom-up

top-down

"start symbol로"

"sentence로"

6.2 구문 분석기의 출력

Text p.239

- The output of a parser:

① ***Parse*** - left parse, right parse

② ***Parse tree***

③ ***Abstract syntax tree***

ex) G : 1. E → E + T string : **a + a * a**

2. E → T

3. T → T * F

4. T → F

5. F →(E)

6. F → a

- ***left parse*** : a sequence of production rule numbers applied in leftmost derivation.

$$E \underset{1}{\Rightarrow} E + T \quad \underset{2}{\Rightarrow} T + T \quad \underset{4}{\Rightarrow} F + T$$

$$\underset{6}{\Rightarrow} a + T \quad \underset{3}{\Rightarrow} a + T * F \quad \underset{4}{\Rightarrow} a + F * F$$

$$\underset{6}{\Rightarrow} a + a * F \quad \underset{6}{\Rightarrow} a + a * a$$

$$\therefore \ \mathbf{1\ 2\ 4\ 6\ 3\ 4\ 6\ 6}$$

- ***right parse*** : reverse order of production rule numbers applied in rightmost derivation.

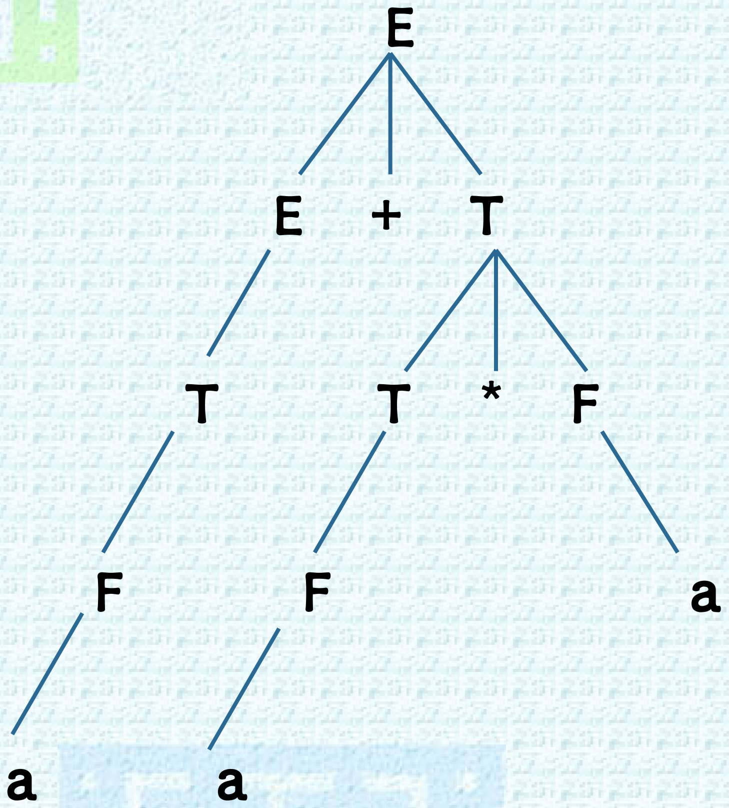
$$E \underset{1}{\Rightarrow} E + T \quad \underset{3}{\Rightarrow} E + T * F \quad \underset{6}{\Rightarrow} E + T * a$$

$$\underset{4}{\Rightarrow} E + F * a \quad \underset{6}{\Rightarrow} E + a * a \quad \underset{2}{\Rightarrow} T + a * a$$

$$\underset{4}{\Rightarrow} F + a * a \quad \underset{6}{\Rightarrow} a + a * a$$

$$\therefore \ \mathbf{6\ 4\ 2\ 6\ 4\ 6\ 3\ 1}$$

- ***parse tree*** : derivation tree



string : **a + a * a**

Text p.243

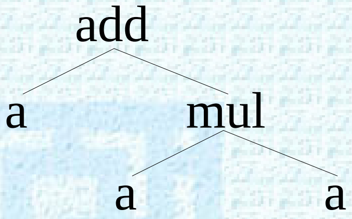
Abstract Syntax Tree(AST)

::= a transformed parse tree that is a more efficient representation of the source program. 의미있는 것만

- leaf node - operand(identifier or constant)
- internal node - operator(meaningful production rule name)

ex) G: 1. $E \rightarrow E + T \Rightarrow$ add
2. $E \rightarrow T$
3. $T \rightarrow T * F \Rightarrow$ mul
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow a$

string : **a + a * a**



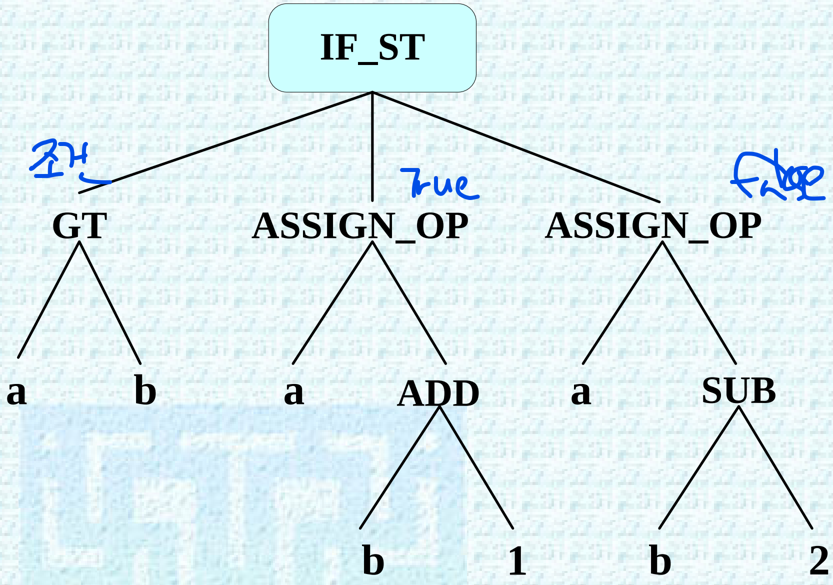
※ 의미 있는 terminal ⇒ terminal node

의미 있는 production rule ⇒ nonterminal node

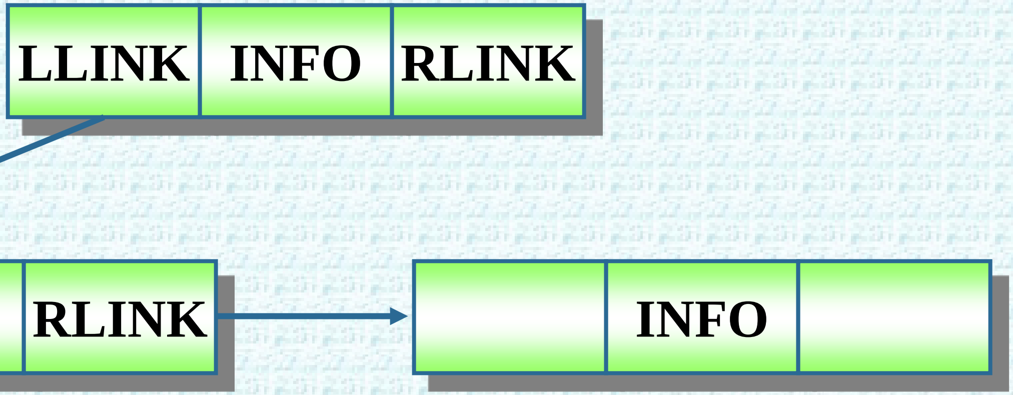
→ naming : compiler designer가 지정.

ex) if (a > b) a = b + 1; else a = b – 2;

Text p.245



- Representation of a parse tree 파트리
  - Implicit representation - the sequence of production rule numbers used in derivation
  - Explicit representation - a linked list structure



* left son/right brother

6.3 Top-Down 방법

Text p.246

7장에서

::= Beginning with the start symbol of the grammar, it attempts to produce a string of terminal symbol that is **identical** to a given source string. This matching process proceeds by successively applying the productions of the grammar to produce substrings from nonterminals.

::= In the terminology of trees, this is moving from the root of the tree to a set of leaves in the parse tree for a program.

Top-Down parsing methods

시행착오

(1) Parsing with backup or backtracking.(brute-force)

(2) Parsing with limited or partial backup.

(3) Parsing with **nobacktracking**.(7장 LL구문)

- backtracking : making repeated scans of the input.

General Top-Down Parsing method

- called a brute-force method
- with backtracking (≡ Top-Down parsing with full backup)

1. Given a particular nonterminal that is to be expanded, the **first** production for this nonterminal is applied.

2. Compare the newly expanded string with the input string. In the matching process, terminal symbol is **compared** with an input symbol is selected for expansion and its first production is applied.

3. If the generated string does not match the input string, an incorrect expansion occurs. In the case of such an incorrect expansion this process is **backed up** by undoing the most recently applied production. And the next production of this nonterminal is used as next expansion.

4. This process continues either until the generated string becomes an input string or until there are no further productions to be tried. In the latter case, the given string cannot be generated from the grammar.

ex) **text p. 247 [예제 6.8]**

Several problems with top-down parsing method

- **left recursion**
  - A nonterminal A is **left recursive** if $A \Rightarrow A\alpha$ for some α.
  - A grammar G is **left recursive** if it has a left-recursive nonterminal.

  ⇒ A left-recursive grammar can cause a top down parser to go into an infinite loop.

  ∴ eliminate the left recursion.

- **Backtracking**
  - the repeated scanning of input string.
  - the speed of parsing is much slower. (very time consuming)

  ⇒ the conditions for nobacktracking : **FIRST, FOLLOW**을 이용하여 formal하게 정의.

Elimination of left recursion

- direct left-recursion : $A \rightarrow A\alpha \in P$
- indirect left-recursion : $A \overset{+}{\Rightarrow} A\alpha$
  - *general form* : $A \rightarrow A\alpha \mid \beta$

    $A = A\alpha + \beta$

    $= \beta\alpha^*$

  - *introducing* new nonterminal A′ which generates α *.

$$\Longrightarrow \quad A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

ex) $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \quad \mid a$

$$E \overset{*}{\Rightarrow} E(+T)^* \Rightarrow T\underline{(+T)}^*$$

$$||$$

$$E' \Leftrightarrow E' \rightarrow +TE' \mid \varepsilon$$

※ $E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

- ***general method*** :

  $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

==> $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

Left-factoring

- if A → $\alpha\beta$ ¦ $\alpha\gamma$ are two A-productions and the input begins with a non-empty string derived from α, we do not know whether to expand A to $\alpha\beta$ or to $\alpha\gamma$.

==> **left-factoring** : the process of factoring out the common prefixes of alternates.

- method :

$$A \rightarrow \alpha\beta \mid \alpha\gamma \implies A \rightarrow \alpha(\beta|\gamma)$$

$$\implies A \rightarrow \alpha A',\ A' \rightarrow \beta \mid \gamma$$

ex) $S \rightarrow iCtS \mid iCtSeS \mid a$

$C \rightarrow b$

$S \rightarrow \mathbf{iCtS} \mid \mathbf{iCtS}eS \mid a$

$\quad \rightarrow \mathbf{iCtS}(\varepsilon \mid eS) \mid a$

$\therefore\ S \rightarrow iCtSS' \mid a$

$\quad S' \rightarrow \varepsilon \mid eS$

$\quad C \rightarrow b$

- **No-backtracking**

  ::= deterministic selection of the production rule to be applied.

6.4 Bottom-up 방법

Text p.255

::= Reducing a given string to the start symbol of the grammar. ☆

::= It attempts to construct a parse tree for an input string beginning at the leaves (the *bottom*) and working up towards the root(the *top*).

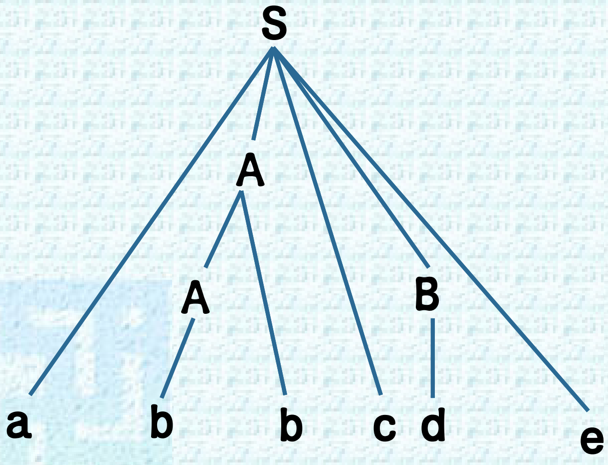
Bottom-up(reduce)

ex) G: $S \rightarrow aAcBe$ ← string : **abbcde**

$A \rightarrow Ab \mid b$ →

$B \rightarrow d$

Top-Down(derivation)



Reduce

[Def 3.1] reduce : the replacement of the right side of a production with the left side.

$$S \overset{*}{\underset{rm}{\Rightarrow}} \alpha\beta\omega, \quad A \rightarrow \beta \in P$$

$$\rightarrow \quad S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A\omega \underset{rm}{\Rightarrow} \alpha\beta\omega$$

[Def 3.2] handle : If $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A\omega \Rightarrow \alpha\beta\omega$, then β is a ***handle*** of $\alpha\beta\omega$.

[Def 3.3] ***handle pruning*** : $\exists S \underset{rm}{\Rightarrow} r_0 \underset{rm}{\Rightarrow} r_1 \underset{rm}{\Rightarrow} \dots \underset{rm}{\Rightarrow} r_{n-1} \underset{rm}{\Rightarrow} r_n = \omega$

$$\omega \Rightarrow r_{n-1} \Rightarrow r_{n-2} \Rightarrow \dots \Rightarrow S$$

“ reduce sequence ”

ex) G : $S \rightarrow bAe$ $\qquad \omega$: b a ; a e) input

$A \rightarrow a;A \mid a$

Right Sentential form	Reducing Production	
ba ; ae	A → a	3
ba ; Ae	A → a;A	2
bAe	S → bAe	1
S		

table

- reduce sequence : 3 2 1

$$S \underset{1}{\Rightarrow} b\ A\ e \underset{2}{\Rightarrow} b\ a\ ;\ A\ e \underset{3}{\Rightarrow} b\ a\ ;\ a\ e$$

- right parse : 3 2 1 (rightmost derivation in reverse)

♣ **note** Bottom-Up parsing의 main aspect는 right sentential form에서 handle을 찾아 적용할 production rule을 deterministic 하게 선택하는 것이다.

Shift-Reduce Parsing

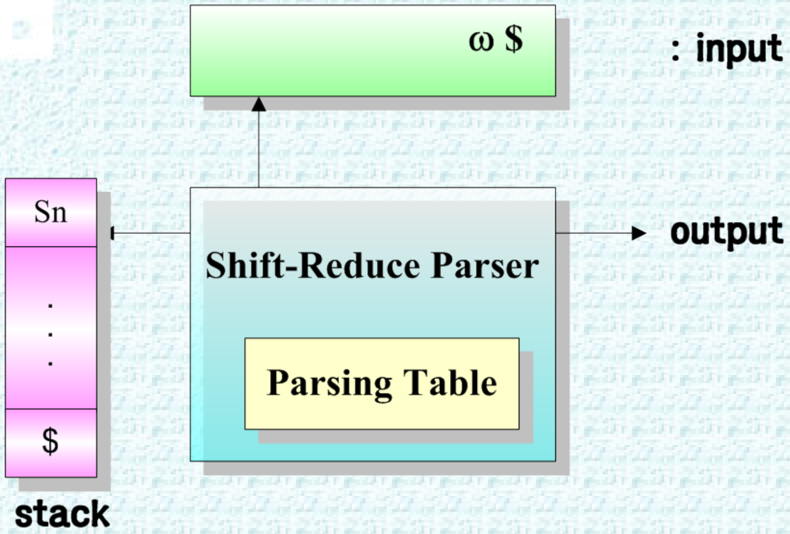
☆☆ 중요

::= a bottom-up style of parsing.

- **Two problems for automatic parsing**
  1. How to find a handle in a right sentential form.
  2. What production to choose in case there is more than one production with the same right hand side.

====> grammar의 종류에 따라 방법이 결정되지만
handle를 유지하기 위하여 stack을 사용한다.

Four *actions* of a shift-reduce parser



"Stack top과 current input symbol에 따라 파싱 테이블을 참조해서 action을 결정."

1. ***shift*** : the next input symbol is shifted to the top of the stack.
2. ***reduce*** : the handle is reduced to the left side of production.
3. ***accept*** : the parser announces successful completion of parsing.
4. ***error*** : the parser discovers that a syntax error has occurred and calls an error recovery routine.

ex) G: E →E + T | T string : **a + a ∗ a**
T →T ∗ F | F
F → (E) | a

STACK	INPUT	ACTION	
(1) $	a + a ∗ a $	shift	a
(2) $a	+ a ∗ a $	reduce	F → a
(3) $F	+ a ∗ a $	reduce	T → F
(4) $T	+ a ∗ a $	reduce	E → T
(5) $E	+ a ∗ a $	shift	+
(6) $E +	a ∗ a $	shift	a
(7) $E + a	∗ a $	reduce	F → a
(8) $E + F	∗ a $	reduce	T → F
(9) $E + T	∗ a $	shift	∗
(10) $E + T ∗	a $	shift	a
(11) $E + T ∗ a	$	reduce	F → a
(12) $E + T ∗ F	$	reduce	T → T * F
(13) $E + T	$	reduce	E → E + T
(14) $E	$	**accept**	

<< Thinking points >>

1. the handle will always eventually appear on top of the stack, never inside.

⇒ ∵ rightmost derivation in reverse.

stack에 있는 contents와 input에 남아 있는 string이 합해져서 right sentential form을 이룬다. 따라서 항상 stack의 top부분이 reduce된다.

2. How to make a parsing table for a given grammar.

→ 문법의 종류에 따라 Parsing table을 만드는 방법이 다르다.

SLR(**S**imple **LR**)
LALR(**L**ook**A**head **LR**)
CLR(**C**anonical **LR**)

Constructing a Parse tree

1. shift : create a **terminal node** labeled the shifted symbol.
2. reduce : A → $X_1X_2 \ldots X_n$.

(1) A new node labeled A is created.

(2) The $X_1X_2 \ldots X_n$ are made direct descendants of the new node.

(3) If A → ε , then the parser merely creates a node labeled A with no descendants.

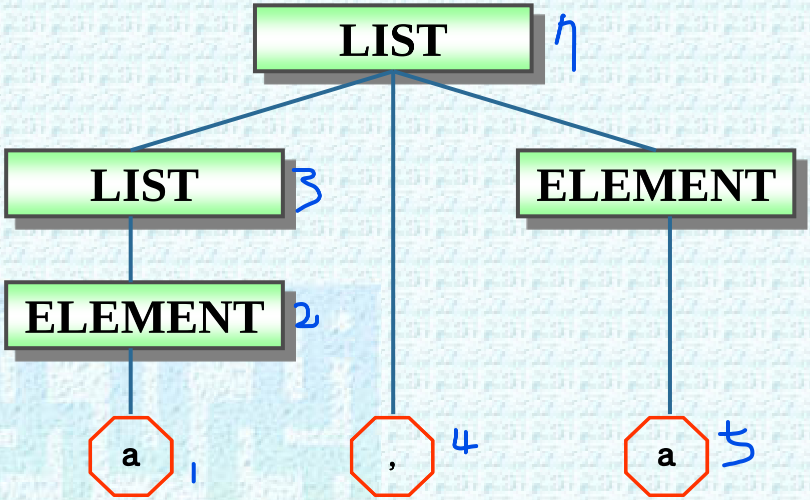
ex) G : 1. LIST → LIST , ELEMENT

2. LIST → ELEMENT

3. ELEMENT → a

string : **a , a**

Step	$ STACK	INPUT	ACTION	PARSETREE
(1)	$	a,a$	shift a	Build Node
(2)	$a	,a$	reduce 3	Build Tree
(3)	$ELEMENT	,a$	reduce 2	Build Tree
(4)	$LIST	,a$	shift ,	Build Node
(5)	$LIST ,	a$	shift a	Build Node
(6)	$LIST , a	$	reduce 3	Build Tree
(7)	$LIST , ELEMENT	$	reduce 1	Build Tree
(8)	$LIST	$	accept	return that tree



Constructing an AST

1. build node : 의미있는 terminal을 shift.

2. build tree : 의미있는 생성 규칙으로 reduce.

ex) G: 1. LIST → LIST, ELEMENT
2. LIST → ELEMENT
3. ELEMENT → a

string: **a,a**

의미있는 terminal : a
의미있는 생성 규칙 : 0번

===> 0. ACCEPT → LIST ⇒ **list**
1. LIST → LIST, ELEMENT
2. LIST → ELEMENT
3. ELEMENT → a

	STACK	INPUT	ACTION	AST
(1)	$	a,a$	shift a	Build Node
(2)	$a	,a$	reduce 3	
(3)	$ELEMENT	,a$	reduce 2	
(4)	$LIST	,a$	shift ,	
(5)	$LIST ,	,a$	shift a	Build Node
(6)	$LIST , a	$	reduce 3	
(7)	$LIST, ELEMENT	$	reduce 1	
(8)	$LIST	$	reduce 0	Build Tree
(9)	$ACCEPT	$	Accept	return that tree