COMPILER 강의 노트

강 병 도

대구대학교 컴퓨터정보공학부

Programming Languages

A programming language serves as a means of communication between the PERSON WITH A PROBLEM and the COMPUTER USED TO HELP SOLVE IT.

A program solution to a given problem will be easier and more natural to obtain if the programming language used is close to the problem. The language should contain constructs which reflect the terminology and elements used in describing the problem and are independent of the computer used. "High-level language"

Digital Computer : "Low-level Language" sequences of 0's & 1's

A hierarchy of PL based on increasing machine independence

1. Machine - Level languages
2. Assembly languages
3. High - level (user-oriented) languages
4. Problem - oriented languages

Machine-level language : lowest form of computer language
: Each instruction in a program is represented by a numeric code, and numerical addresses are used.

Assembly language : a symbolic version of a machine-level language
: Each operation code is given a symbolic code (ADD, SUB)
Memory location are given symbolic names (SUM, ACCOUNT)

High-level language :
the features of an assembly language
\+ structured control constructs,
nested statements,
blocks,
procedures

Problem-Oriented language :
Provides for the expression of problems in a specific application or problem area.
- SEQUEL : for database retrieval applications
- CoGo : for civil engineering applications

Advantages of high-level language over Low-level language

1. High-level languages are easier to learn.
   : no computer HW background, machine-independent.

2. The programmer does not have to be concerned with clerical tasks involving numerical or symbolic references to instructions, memory locations, constants, etc.

3. A programmer is not required to know how to convert data from external forms to various internal forms within the memory of a computer.
   eg) numeric data ——convert——> floating-point numbers
   packed-decimal numbers

4. Most high-level languages offer a programmer a variety of control structures which are not available in low-level languages.
   - Conditional statements
   - Looping statements
   - Nested Statements
   - Block Structures

5. Program written in a high-level language are usually more easily debugged than their machine - or assembly - language equivalents.

6. Since most high-level languages offer more powerful control and data-structuring capabilities than low-level languages, the former class of languages facilitates the expression of a solution to a particular problem.

7. Because of the availability of certain language features such as procedures, high-level languages permit a modular and hierarchical description of programming tasks.
   : division of labor, a team of programers.

8. High-level languages are relatively machine-independent.

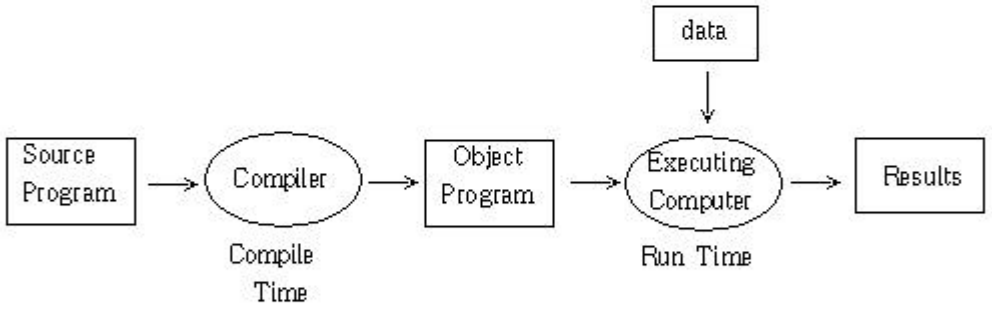
TRANSLATION - compiler
- Interpreter

A translator inputs and then converts a source program into an object or target program. The source program is written in a source language and the object program belongs to an object language.

Source program	→	translator	→	target program	/	object program
Assembly language	→	Assember	→	Machine language		
High-level language	→	Compiler	→	Assembly language	/	Machine language

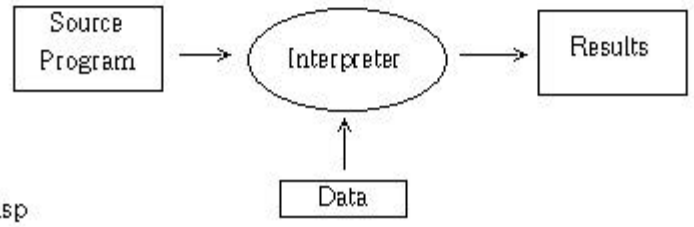
The time at which the conversion of a source program to an object program occurs is called "compile time".
The object program is executed at "run time".

Compilation process



cf) Interpreter (another kind of translator)

The interpreter processes an internal form of the source program and data at the same time. Interpretation of the internal source form occurs at run time and no object program is generated.



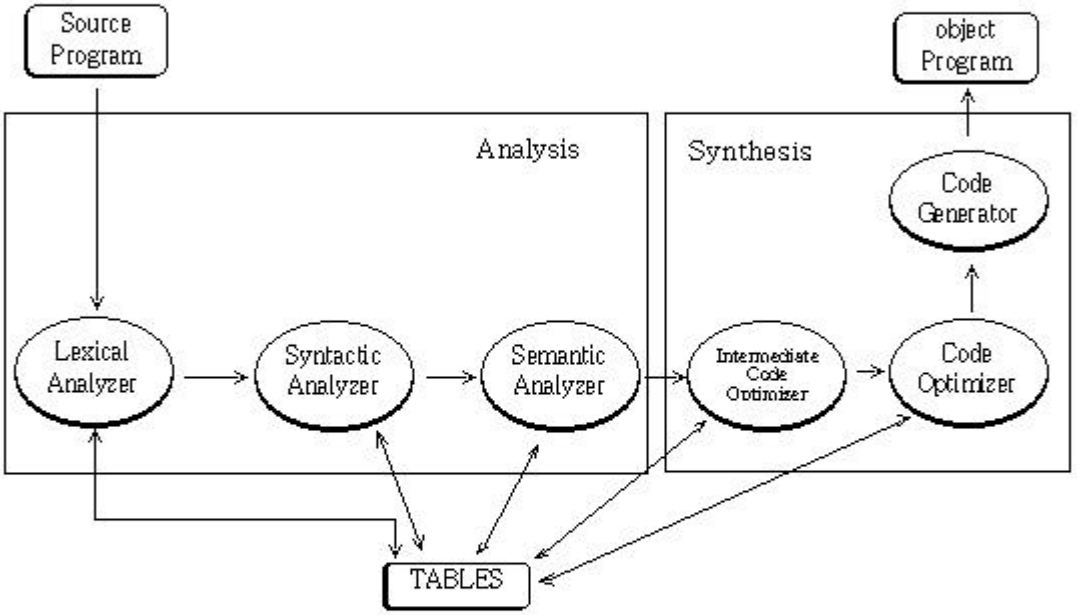
- Basic, Lisp
- APL, Smalltalk-80

MODEL OF A COMPILER

The task of Constructing a compiler for a particular source language is complex. The complexity and nature of the compilation process depend on the Source Language.

Compiler complexity can often be reduced if a programing-language designer takes various design factors into consideration.

COMPONENTS OF A COMPILER



A compiler must perform two major tasks:

① the analysis of a source program

and ② the synthesis of its corresponding object program.

The analysis task deals with the decomposition of the source program into it's basic parts. Using there parts, the synthesis task builds their equivalent object program modules.

The performance of these tasks is realized more easily by building and maintaining several tables.

The source program is input to a lexical analyzer(scanner) whose purpose is to separate the incoming text into tokens such as constants, variable names, keywords, and operators.

In essence, the lexical analyzer performs low-level syntax analysis. For efficiency reasons, each class of tokens is given a unique internal representation number.

eg) TEST : IF A > B THEN X = Y;

lexical analyzer →

TEST	3
:	26
IF	20
A	1
>	15
B	1
THEN	21
X	1
=	10
Y	7
;	27

★ Blanks and comments are ignored.

Some scanners place constants, labels, and variable names in appropriate tables(Symbol table).

var name	TYPE	object-program address	value	line #
	REAL INT BOOL			

The lexical analyzer supplies tokens to the syntax analyzer. These tokens may take the form of a pair of items.

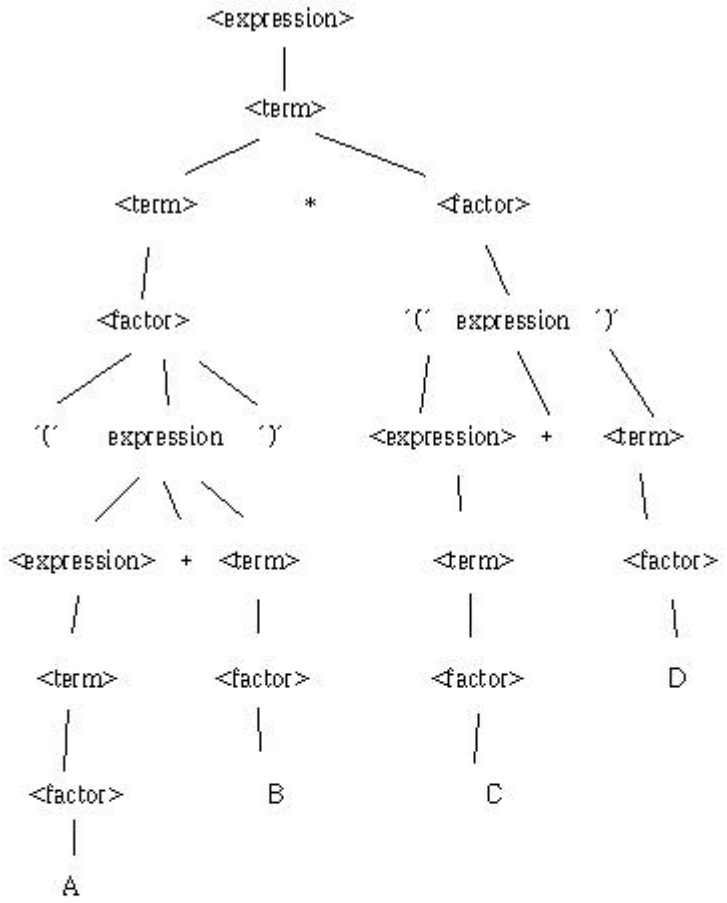
addr(Location) of the token in symble table	representation # of the token
(address)	(integer)

fixed-length information

The syntax analyzer is much more complex than the lexical analyzer. Its function is to take the source program in the form of tokens from the lexical analyzer and determine the manner in which it is to be decomposed into its constituent parts. That is the syntax analyzer determines the overall structure of the source program.

In syntax analysis we are concerned with grouping tokens into larger syntactic classes such as expression, statement, and procedure. The syntax analyzer(parser) outputs a syntax tree in which its leaves are the tokens and every nonleaf node represents a syntactic class type. A set of rules known as a grammar is used to define precisely the source language. A grammar can be used by the syntax analyzer to determine the structure of the source program. This recognition process is called parsing, and consequently we often refer to syntax analyzer as parsers.

eg) (A + B) * (C + D)



The syntax tree produced by the syntax analyzer is used by the semantic analyzer. The function of the semantic analyzer is to determine the meaning(semantics) of the source program.

eg) (A + B) * (C + D)

When the parser recognizes an operator such as '+' or '*', it invokes a semantic routine which specifies the action to be performed.

(operand, type, value)

The semantic analyzer often interacts with the various tables of the compiler in performing its task.

The semantic-analyzer actions may involve the generation of an intermediate form of source code.

eg) (A + B) * (C + D) : the infix expression

intermediate source code → (+, A, B, T1)
(+, C, D, T2)
(*, T1 ,T2, T3)

suffix expression → AB + CD + *

The output of the semantic analyzer is passed on to the Code generator. At this point the intermediate form of the source-language program is usually translated to either assembly language or machine languages.

eg)

LOAD	A	Contents(A) → Accumulator
ADD	B	C(B) + C(Acc) → Accu
STO	T1	C(Acc) —store→ T1
LOAD	C	C(Acc) → Accumulator
ADD	D	C(D) + C(Acc) → Acc
STO	T2	C(Acc) → T2
LOAD	T1	C(T1) → Acc
MUL	T2	C(T2) * C(Acc) → Acc
STO	T3	C(Acc) → T3

The output of the code generator is passed on to a code optimizer. Its purpose is to produce a more efficient object program.

eg)

LOAD	A
ADD	B
STO	T1
LOAD	C
ADD	D
MUL	T1
STO	T2

"Optimization Technique"

⇒ (C + D) * (A + B)

Lexical Analysis

※ Lexical Analysis : Source program 의 각 word를 읽어서 "token" 단위로 구성

cf) Lexical Analyzer (Scanner)

token :
- Constant
- Identifier / Variable name
- Keywords (reserved words)
- Operators
- Label

예) PROGRAM SAMPLE ;
```
CONST
   M = 5 ;
   N = 9 ;
VAR
   SUM, MUL : INTEGER ;
BEGIN
   SUM := M + N ;
   MUL := M * N ;
END .
```

→ identifier : PROGRAM, SAMPLE, CONST, M, N, VAR, SUM, MUL, BEGIN, END

constant : 5,9

operator : = : ; , := * + .

variable : SAMPLE, M, N, SUM, MUL

reserved word : PROGRAM, CONST, VAR, INTEGER, BEGIN, END

- Token Table

Token	Code	Value
BEGIN	1	-
CONST	2	-
END	3	-
INTEGER	4	-
PROGRAM	5	-
VAR	6	-
identifier	7	pointer to symbol table
constant	8	pointer to symbol table
=	9	1
;	9	2
:	9	3
:=	9	4
+	9	5
*	9	6
,	9	7
.	9	8

reserved word는 토큰의 값이 Token table에 정해져 있음
variable name은 symbol table을 만들어 Token Code 생성

→ Symbol Table : identifier⑦와 Constant⑧ in token table을 위한 symbol Table

Base Location	identifier	class(종류)	value
15	SAMPLE	id	
16	M	id	
17	MUL	id	
18	N	id	
19	SUM	id	
21	5	cons	
22	9	cons	

→ Source Program ⇒ Token

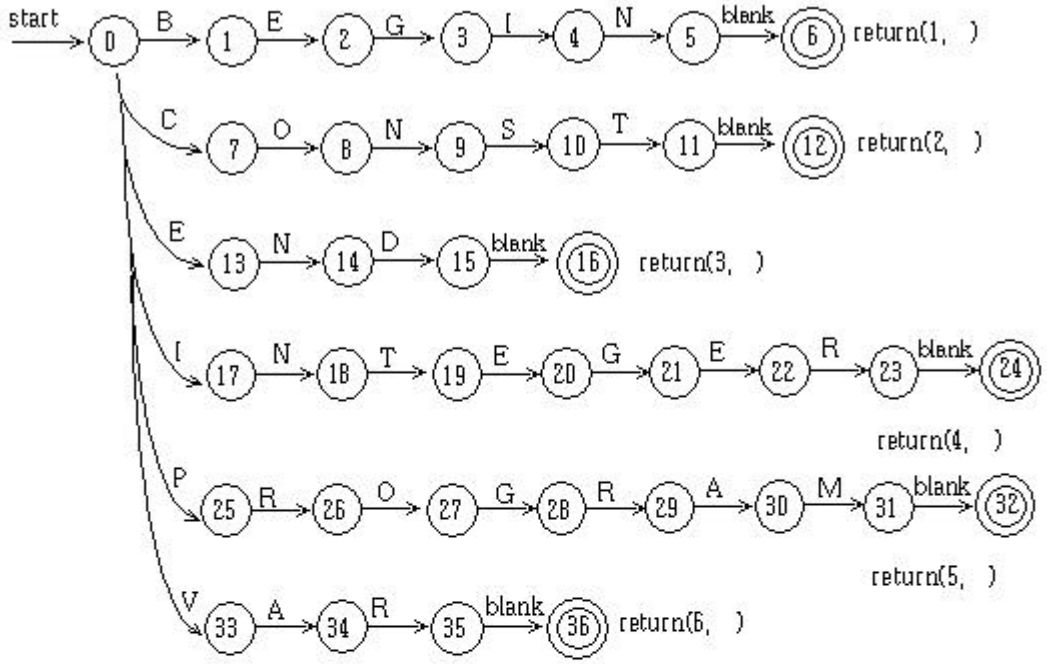
```
PROGRAM          SAMPLE    ;
  [5, ]          [7,15]    [9,2]
CONST     M    =     5       ;
  [2, ]   [7,16] [9,1] [8,21]  [9,2]
          N    =     9       ;
          [7,18] [9,1] [8,22]  [9,2]

VAR
  [6, ]
        SUM   ,   MUL   :   INTEGER ;
        [7,19] [9,7] [7,17] [9,3]  [4, ]  [9,2]
BEGIN
  [1, ]
        SUM   :=    M     +     N     ;
        [7,19] [9,4] [7,16] [9,5] [7,18] [9,2]
        MUL   :=    M     *     N     ;
        [7,17] [9,4] [7,16] [9,6] [7,18] [9,2]
END .
[3, ] [9,8]
```

※ Token을 만들기 위해서 Token Table을 찾는 방법

→ "Finite Automata"를 많이 이용

· reserved word :

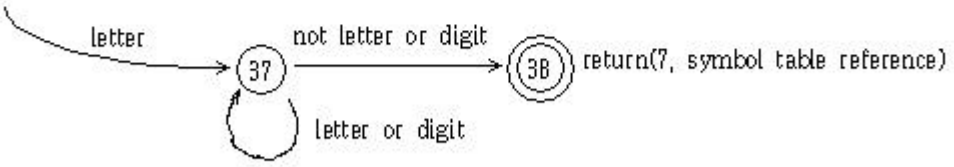


<identifier> ::= <letter> | <identifier><letter> | <identifier><digit>

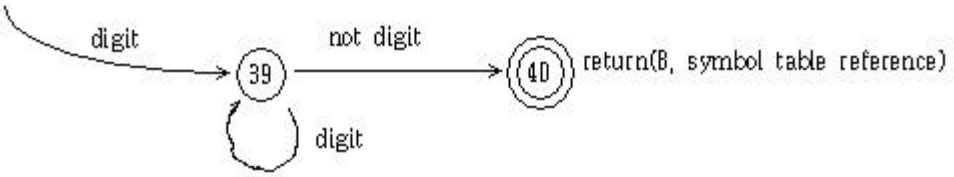
<letter> ::= A | B | C | … | Z

<digit> ::= 0 | 1 | 2 | … | 9

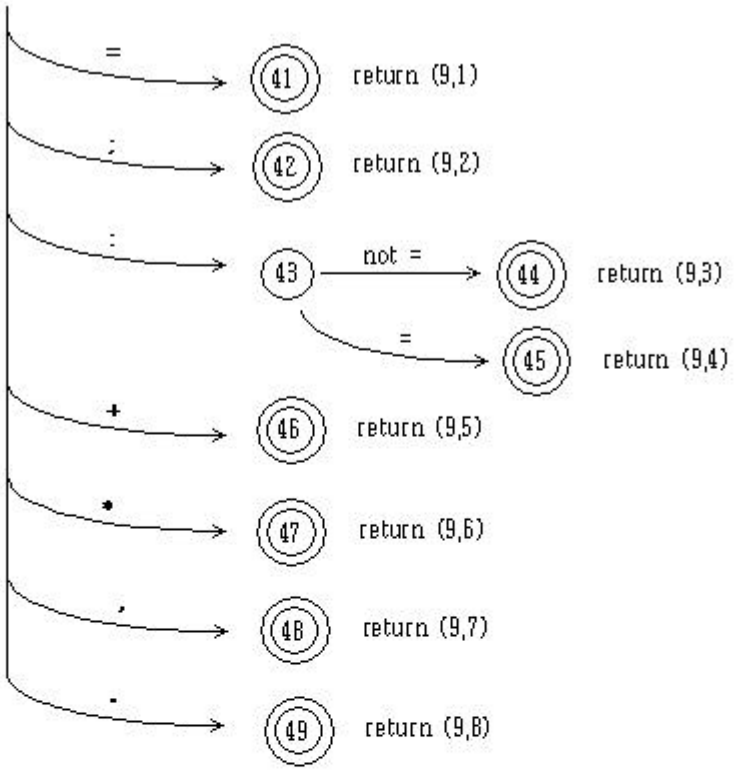
· identifier :



· Constant :



· Operator :



Finite Automata 의 state : 49개
return(m, n) : Finite Automata 에 의해 정해짐

1) Symbol Table

Source Program에 나타나는 identifier에 대한 정보를 Symbol Table 이라는 Data Structure 에 저장

명칭	정보		
identifier	attribute	offset	pointer
character string	data type	프로그램의 처음을 기준으로 거리를 계산	point to attribute structure

In lexical analysis,

Symbol Table 생성 → identifier 저장

In semantic Analysis,

attribute, attribute structure, pointer to attribute data structure 저장

※ **Symbol Table 생성시 고려할 사항**

① 어떤 id(명칭)이 ST에 있는지 없는지를 Lexical Analysis 단계에서 알아야 한다(없으면 첨가)

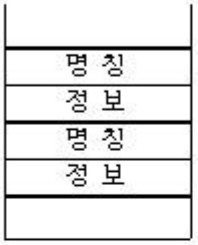
② id(명칭)를 ST에 첨가시 정보도 동시에 저장 가능해야 한다.

③ ST에 있는 id의 정보를 검색할 수 있어야 한다

④ ST에 있는 정보들은 컴파일 도중에 필요 없는 경우 지울 수 있어야 한다.

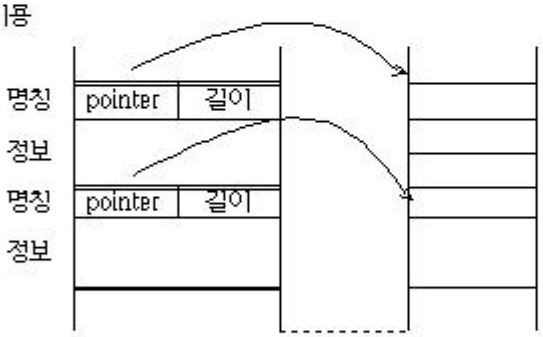
※ Symbol Table의 명칭과 정보를 갖는 Record를 나열하는 방법.

① 일정한 크기



항목 크기가 일정해야
: 긴 것은 모두 기록이 곤란함
짧으면 메모리 낭비

② pointer 이용



※ Symbol Table 구성 방법 for Non-block-structured Language

① Linked List 이용 : 항목수가 n이면 n/2 비교, n이 작으면 유리.

② Binary Tree 이용

③ Hash Table

※ ST 구성방법 for Block-Structured Language

① Stack Symbol Table

② Stack-implemented Tree-Structured ST

③ Stack-implemented Hash-Structured ST

Lex & Yacc

· Lex : tool for building scanners(lexer, lexical analyzer)
· Scanner : ┌ input : arbitrary input stream
└ output : token

1. Word Recognition
"Lex Specification"
Regular Expression : a pattern description using a "meta-language"

.	single character except "\n"
*	zero or more copies of the preceding expression eg) a *
[ ]	character class (any character within the brackets) eg) [abc] ≒ a, b, c [abc]* [0-9] ≒ [0123456789] [a-zA-Z] [^] any character except the ones within the brackets A "-" or "]" as the first character after the "[" is interpreted literally.
^	at the beginning of a pattern beginning of an input line as the 1st character of a regular expression
$	at the end of a pattern end of an input line(not "\n") as the last character of a regular expression
{ }	How many times the previous pattern is allowed to match eg) A{1,3} 1~3 occurrences of the letter A
\	a single following special character eg) \\ "\" \b backspace \t tab character \n line feed * "*" \" "
+	One or more occurrence of the preceding regular expression eg) [0-9]+ (not empty string) [0-9]* (also empty string)

?	zero or one occurrence of the preceding regular expression eg) -?[0-9]+ a signed number -1, 1, …
|	Either the preceding regular expression or the following regular expression eg) A|B|C ≒ A or B or C, any one of three
"……"	terminal symbol, operators eg) "*" "+" "=" ":="
/	Preceding regular expression but only if followed by the following regular expression ex) 0/1 "0" in the string "01", not "0" or "02"
( )	A series of regular expressions ex) (01) character sequence 01

(Examples of Regular Expressions)

1. operators

   "=" "*"

   ":=" "/"

   "+" "-"

2. white space

   [\b\t\n]

3. digit

   [0-9]

4. constants

   [0-9] [0-9]*

5. identifier

   [A-Za-z_][A-Za-z0-9]*

6. unary minus

   -?[0-9]+

7. integer

   [0-9]+

8. single-line comments

   "/*" . *"*/"

9. C-Style comments

   "/*" "/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/"

10. Pascal-Style comment

    "{"[^}]"}"

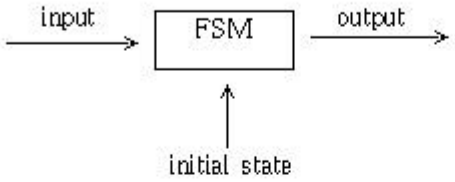
```
/*
* word count, WC - the program, simple standalone PCLEX application,
*/
%{
#include <stdlib.h>
static unsigned nchar=0; /* # of characters in file */
static unsigned nword=0; /* # of words in file */
static unsigned nline=1; /* # of lines in file */
%}
%%
\n nchar +=2, ++nline; /* line boundary in MS-DOS is CR LF *
[^\t\n]+ ++nword, nchar +=yyleng;
.       ++nchar;
%%
main()
{
        yylex();
        printf("%d\t%d\t%d\n", nchar, nword, nline);
        exit(0);
}
```

This read.me file is for those who have never used LEX.
type,
        "pclex wc.l", generates wc.c
type,
        "cl wc.c", generate wc.exe, assuming Microsoft C
To Run,
        "wc < wc.c", count the number of lines in wc.c.

If you must analyze files with non-ascii data then, invoke pclex-"pclex -8 wc.l"

Finite State Machine(FSM)

- FSM의 행동은 discrete time(이산순간)들에 일어나는 일련의 반응들로 구성
- 시간 t에 입력 s(t)가 FSM에 accept되면 출력 r(t)가 발생
- 출력 r(t)는 입력 s(t)와 s(t) 이전의 입력들에 의해서 결정



$FSM \quad M = (Q, S, R, f, g, q_1)$

Q : finite set of internal states

S : input alphabet

R : output alphabet

f : state transition function

f : Q×S -------> Q

g : output function

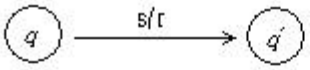
g : Q×S -------> R

q_1 : initial state $\subset$ Q

q(t+1) = f(q(t), s(t+1)) t≥0

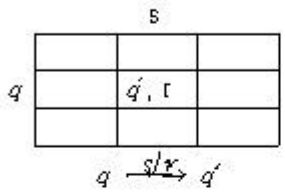
r(t+1) = g(q(t), s(t+1)) t≥0

Finite State Diagram



$$q' = f(q, s)$$
$$r = g(q, s)$$

Finite State Table



eg) parity를 검사하는 FSM

입력 : 0과 1의 sequence

출력 : 1이 짝수개 --------> 0출력 (even parity)

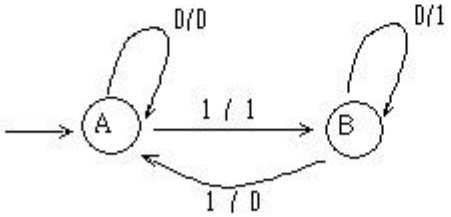
1이 홀수개 --------> 1출력 (odd parity)

⇒ S = R = { 0, 1 }

Q = {A, B}

A : Even 상태(짝수)

B : Odd 상태(홀수)



input / state	0	1
A	A , 0	B , 1
B	B , 1	A , 0

Finite Automata 와 Lexical Analysis

Finite Automata

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q : Finite set of states

Σ : Finite input alphabets

q_0 : Q 내의 initial state

F : Q 내의 한 집합의 final state

δ : state transition function

어떤 String x 가 M에 의해서 accept 된다고 하면 초기단계 q_0에서 입력 x 에 의해서 최종단계로 도달한다.

$$\delta (q_0, x) = p \quad p \subset F$$

이때 x는 M에 의해서 accept된 language이다.

$$L(M) = \{ x \mid \delta (q_0, x) \subset F \}$$

Deterministic FA : 어느 단계에서 어떤 입력에 대해 전이되는 상태가 하나인 것.
Nondeterministic FA : 어느 단계에서 어떤 입력에 대해 전이되는 상태가 정해지지 않은 것.

eg) Finite Automata

$$M = (Q, \Sigma, \delta, q_0, F)$$

$$Q = \{ q_0, q_1, q_2, q_3 \}$$

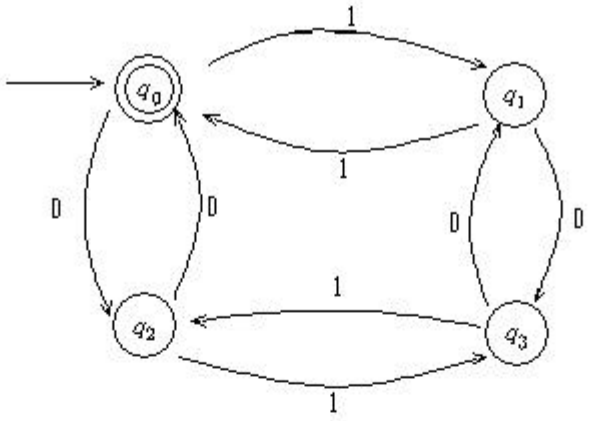
$$\Sigma = \{ 0, 1 \}$$

$$F = \{ q_0 \}$$

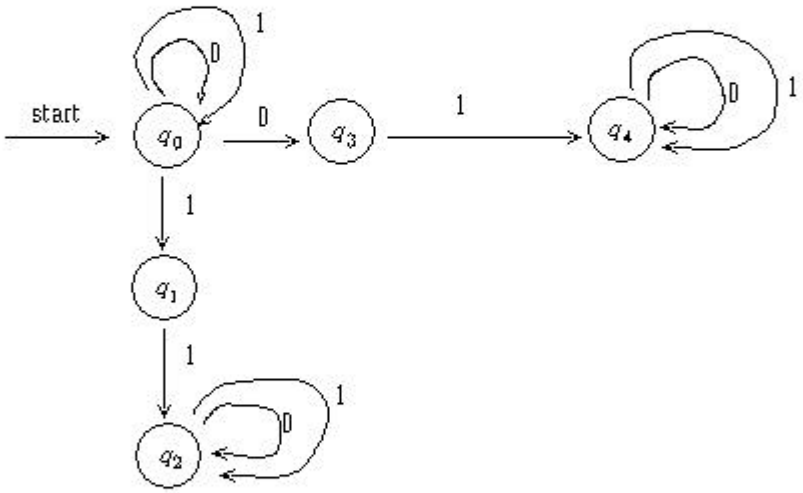
$$\delta = Q \times \Sigma^* \rightarrow Q$$

δ

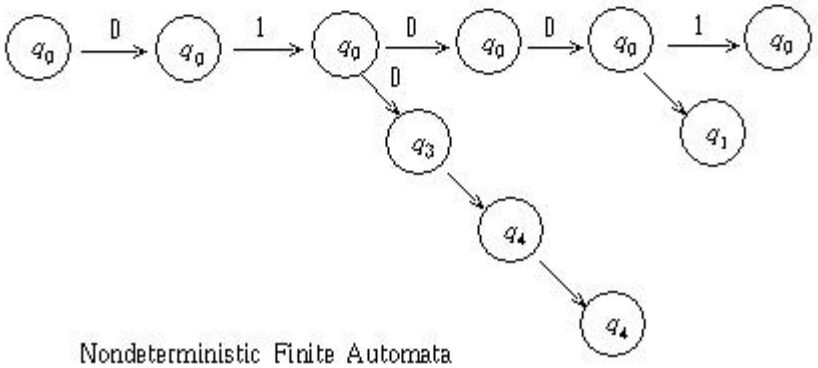
	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2



Deterministic Finite Automata



입력 : 01001



Nondeterministic Finite Automata

Language와 Grammar를 위한 기호법 / 개념

1 기초개념

1) alphabet V : finite nonempty set of symbols

eg) V = { a,b,c, ... ,z }

V = { 0,1,2, ... ,9 }

2) Concatenation (연결) :

eg) a, b ⟶ ab

ab, ab ⟶ abab

3) String : alphabet V 내에 있는 문자이거나 그 문자들을 0번 이상 concatenation한 문자들

eg) V = { a, b, c }

---> a, b, c, aa, aba, aabb, ...

V = { 0, 1 }

---> 0, 1, 001, 01, ...

x = 'dog' |x| = 3

y = 'house' |y| = 5

xy = 'doghouse' |xy| = 8

4) empty string : 공백만으로 구성 'λ' or 'ε'으로 표현

(null) |ε|=0

$A = \{ a, b \}$ $B = \{ \alpha, \beta \}$

$\rightarrow AB = \{ wx \mid w \subset A, x \subset B \}$

$\rightarrow AB = \{ a\alpha, a\beta, b\alpha, b\beta \}$

$A^0 = \{\varepsilon\}$

5)

$$A^1 = A$$

$$A^2 = AA$$

$$A^3 = AAA = A^2A$$

$$A^4 = AAAA = A^3 \cdot A$$

$$\vdots$$

$$A^n = \begin{cases} \{\varepsilon\}, & n=0 \\ A^{n-1}A, & n\neq 0 \end{cases}$$

6) closure

$$A^* = A^0 \cup A^1 \cup A^2 \cup \ldots = \bigcup_{0\le i<\infty} A^i$$

$$A^+ = A^1 \cup A^2 \cup \ldots = \bigcup_{1<i<\infty} A^i$$

2. Grammar

def) Grammar $G = (V_n, V_t, S, P)$

V_n: Set of nonterminal symbols

V_t: Set of terminal symbols

S : Starting symbol

P : Production rules

$\alpha \rightarrow \beta$

eg) $G = (V_n, V_t, S, P)$

$V_n = \{ A, B, C \}$

$V_t = \{ a, b, c \}$

P : $S \rightarrow A \mid A \rightarrow aABC \mid A \rightarrow abC$

$\mid CB \rightarrow BC \mid bB \rightarrow bb \mid bC \rightarrow b$

$\rightarrow$ $\{ a^k b^k \mid k \geq 1 \}$를 생성가능

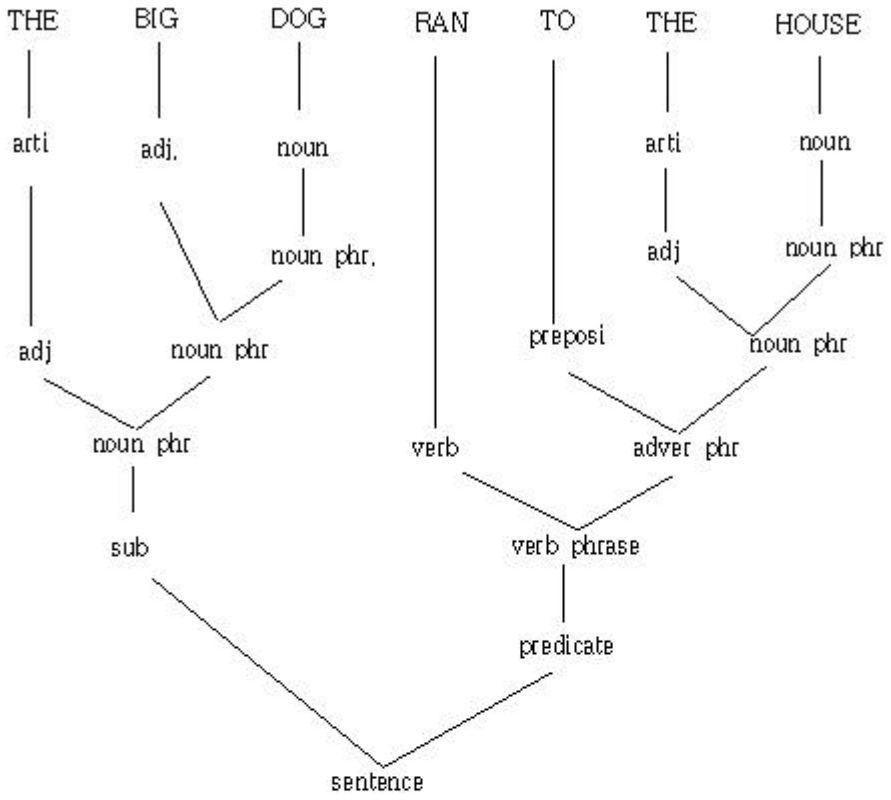
$S \xRightarrow[S \rightarrow A]{} A \xRightarrow[A \rightarrow aABC]{} aABC \xRightarrow[A \rightarrow aABC]{} aaABCBC \xRightarrow[A \rightarrow abC]{} aaabCBCBC$

$\xRightarrow[bC \rightarrow b]{}$ aaabBCBC $\xRightarrow[bB \rightarrow bb]{}$ $aaabbCBC \xRightarrow[bC \rightarrow b]{} aaabbBC \xRightarrow[bB \rightarrow bb]{}$

$\Longrightarrow$ aaabbbC $\xRightarrow[bC \rightarrow b]{}$ $aaabbb (= a^3 b^3)$

"sentential form" Language

symbols are terminal



<sentence> → <subject><predicate>
<subject> → <noun phrase>
<noun phrase> → <noun> | <adjective> <noun phrase>
<predicate> → <verb phrase>
<verb phrase> → <verb><adverbial phrase>
<adverbial phrase> → <preposition><noun phrase>
<adjective> → <article> | BIG
<preposition> → TO
<noun> → DOG | HOUSE
<article> → THE
<verb> → RAN

def) Language $L(G) = \left\{ \sigma \mid S \xrightarrow{*} \sigma \text{ and } \sigma \in V_T^* \right\}$

: Set of all sentential form whose symbols are terminal.

3 Grammar의 분류 "Chomsky"

1) Type 0 : unrestricted grammar
rules are unrestricted $L(G_0)$

2) Type 1 : Context-sensitive grammar
contains only production of the form $\alpha \rightarrow \beta$, where $|\alpha| \leq |\beta|$

$\xrightarrow[\text{생성}]{}$ Context − sensitive language

$| \phi_1 A \phi_2 \rightarrow \phi_1 \phi \phi_2 \quad \phi \neq \varepsilon$

eg) $G_1 = (V_n, V_t, S, P)$
V_n= {S, B, C}
V_t = {a, b, c}
P::= $S \rightarrow aSBC|S \rightarrow abC|bB \rightarrow bb|bC \rightarrow bc|CB \rightarrow BC|cC \rightarrow cc$

$S \Rightarrow aSBC$
$\Rightarrow aabCBC$
$\Rightarrow aabBCC$
$\Rightarrow aabbCC$
$\Rightarrow aabbcC$
$\Rightarrow aabbcc$

$L(G_1) = \{ a^n b^n c^n \mid n \geq 1 \}$

3) Type 2 : Context-free Grammar
contains only productions $\alpha \rightarrow \beta$, where $|\alpha| \leq |\beta|$ and $\alpha \in V_n$

eg) $G_2 = (V_n, V_t, S, P)$
V_n= {S, C}
V_t = {a, b}
P ::= $S \rightarrow aCa|C \rightarrow aCa|C \rightarrow b$
"Left-hand side consists of a single class symbol."

$S \Rightarrow aCa$
$\Rightarrow aaCaa$
$\Rightarrow aaaCaaa$
$\Rightarrow aaabaaa$

$L(G_2) = \{ a^n b a^n \mid n \geq 1 \}$
"Context-free Language"

4) Type 3 : Regular Grammar
Contains only productions of the form $\alpha \rightarrow \beta$, where $|\alpha| \leq |\beta|$, $\alpha \in V_n$ and β has the form aB or a, where $a \in V_t$ and $B \in V_n$

eg) $G_3 = (V_n, V_t, S, P)$

$V_n = \{S, A, B, C\}$

$V_t = \{a, b\}$

$P ::= S \to aS|S \to aB|B \to bC|C \to aC|C \to a$

$S \Rightarrow aS$

$\Rightarrow aaS$

$\Rightarrow aaaB$

$\Rightarrow aaabC$

$\Rightarrow aaabaC$

$\Rightarrow aaabaa$

$L(G_3) = \{ a^n b a^m \mid n, m \geq 1 \}$

$L(G_3) \subset L(G_2) \subset L(G_1) \subset L(G_0)$

Regular Grammar & Deterministic FSA

$$\text{FSA}\quad M = (\underbrace{\{S, A, B, C\}}_{Q},\ \underbrace{\{0, 1\}}_{\Sigma},\ \underbrace{H}_{\delta},\ \underbrace{S}_{q_0},\ \underbrace{\{S\}}_{F})$$

H

	Input	
state	0	1
S	B	A
A	C	S
B	S	C
C	A	B

Regular Grammar

$$G = (\underbrace{\{S, A, B, C\}}_{V_n},\ \underbrace{\{0, 1\}}_{V_T},\ \underbrace{S}_{S},\ \underbrace{P}_{P})$$

P: $S \to 0B$

$S \to 1A$

② $S \to \varepsilon$

$A \to 0C$

① $A \to 1S$

$A \to 1$

① $B \to 0S$

$B \to 0$

$B \to 1C$

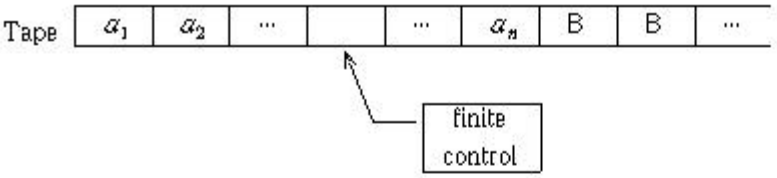
$C \to 0A$

$C \to 1B$

① Because S is a finial state, wherever a production of the form $\alpha \to \beta S$ occurs, the production $\alpha \to \beta$ is also included in δ.

② The production $S \to \varepsilon$ is also in δ because $S \in F$ in the DFA.

Turing Machine : **Alan H. Turing 1936**



구성 : 1개의 finite control
여러 개의 cell로 구성된 입력 Tape

Turing Machine : finite control의 current state와 incoming input 에 따라서 state를 전이하거나, cell에 쓰여있는 기호를 다른 기호로 대체하거나 finite control의 머리를 왼쪽 or 오른쪽으로 한 cell 씩 옮김.

$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Q : finite set of states

Γ : Tape 에 허용되는 기호의 유한집합

B : 공백 $\subset \Gamma$

Σ : input $\subset \Gamma$

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$

q_0 : initial state

F : final state $\subset$ Q

eg)

$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

$Q = \{q_0, q_1, q_2, q_3, q_4\}$

$\Sigma = \{0, 1\}$

$\Gamma = \{0, 1, X, Y, B\}$

$F = \{q_4\}$

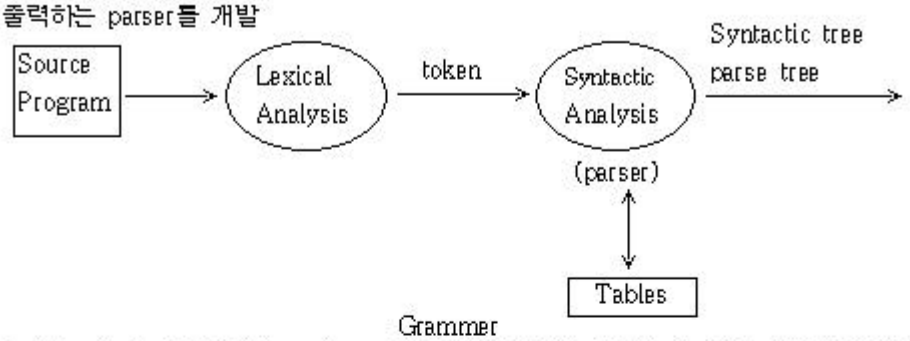
δ	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	
q_1	(q_1, 0, R)	(q_2, Y, L)	-	(q_1, Y, R)	
q_2	(q_2, 0, L)	-	(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	(q_4, B, R)
q_4					

input 0011

$q_0$0011 → X $q_1$011 → X0 $q_1$11 → X $q_2$0Y1 → q_2X0Y1 → X $q_0$0Y1 → XX q_1Y1 → XXY $q_1$1 → XX q_2YY → X q_2XYY → XX q_0YY → XXY q_3Y → XXYY q_3 → XXYYB q_4

Syntax Analysis (Parsing)

- 문법에 맞는 Program을 하나의 Syntactic tree로 표현될 수 있는데, 이 Syntactic tree를 만들어 출력하는 parser를 개발



- Lexical Analysis 결과인 token stream이 문법에 의해서 생성될 수 있는 지를 검사한다.
- 문법오류를 발견하고, 일반적인 오류가 발생 시에는 복구를 해서 나머지 입력을 계속해서 처리할 수 있어야 한다.

parser

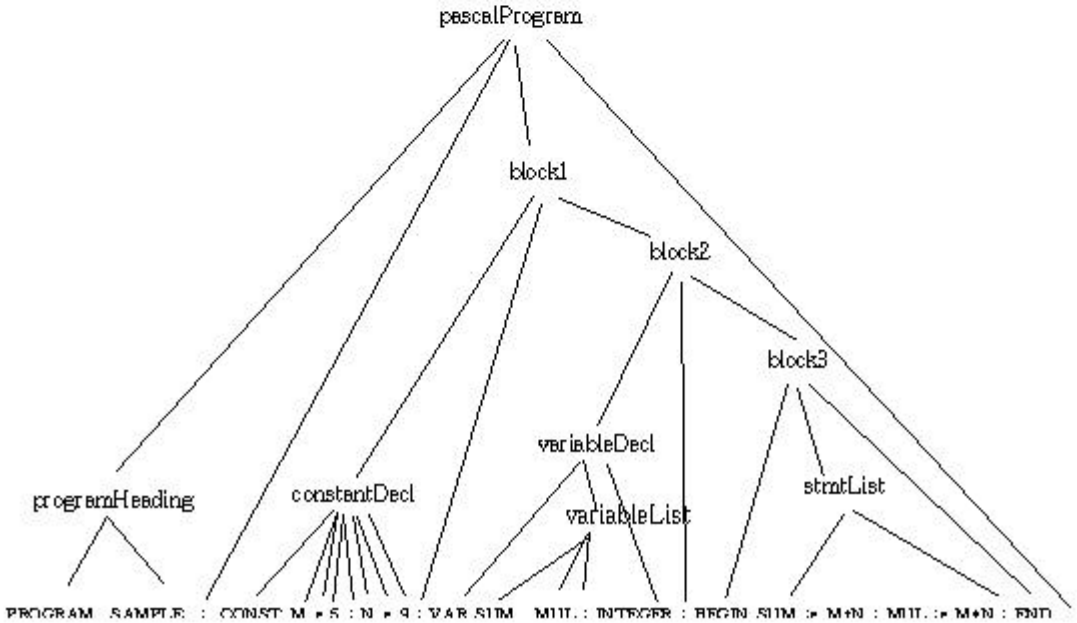
: string w가 문법 G에 의한 Language가 될 수 있다면 parser는 이 string w를 입력으로 받아들여서 출력으로는 tree구조를 만듦. 이 때 만일 string w가 문법 G에 의한 문장이 아니면 string w는 이 문법에 의하여 accept 되지 않으며, error를 알려줌.

예)

```
PROGRAM  SAMPLE ;
CONST
        M = 5;
        N = 9;
VAR
        SUM, MUL: INTEGER ;
BEGIN
        SUM := M + N ;
        MUL := M * N ;
END.
```

Syntax (BNF : Backus - Naur Form)

```
pascalProgram    ::= programHeading ; block1 .
programHeading   ::= PROGRAM IDENTIFIER
block1           ::= ConstantDecl ; block2
ConstantDecl     ::= CONST IDENTIFIER = INTNUMBER
                   | ConstantDecl ; INTENTIFIER = INTNUMBER
block2           ::= variableDecl ; block3
variableDecl     ::= VAR variableIdList : INTEGER
variableIdList   ::= IDENTIFIER
                   | variableIdList , IDENTIFIER
block3           ::= BEGIN stmtList END
stmtList         ::= stmt ;
                   | stmtList stmt ;
stmt             ::= variable := simpleExpression
variable         ::= IDENTIFIER
simpleExpression ::= term
                   | simpleExpression operator term
term             ::= IDENTIFIER
                   | INTNUMBER
operator         ::= +
                   | *
```



"Parsing의 개념" (Parser)

Parser는 어떤 스트링 w가 문법 G에 의한 언어가 될 수 있다면 parser는 이 스트링 w를 입력으로 받아들여 출력으로는 어떤 tree를 만듦. 이때 만일 스트링 w가 문법 G에 의한 문장이 아니면 입력 스트링 w는 이 문법에 의하여 "accept"되지 않으며, parser는 error를 알려줌
(parse tree에 의해서 생성되는 문자열과 입력 token stream이 같도록 만들면 된다)

1) Left most derivation

Parse tree를 구성하려면 production에 의하여 production의 오른쪽 항을 변환한다. 즉 생성 $\alpha \Rightarrow \beta$의 경우 $\alpha = wAr$, $\beta = w\delta r$이며 w가 Terminal로 구성되었다면, nonterminal A는 production에 의하여 δ로 대치되는데, 이러한 생성을 left most derivation이라 한다.

$$wAr \underset{lm}{\Rightarrow} w\delta r$$

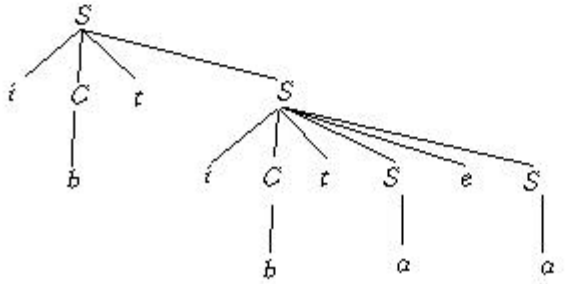
left most derivation를 이용하여 입력 w 에 대한 parse tree는

$S \underset{lm}{\Rightarrow} \alpha_1 \underset{lm}{\Rightarrow} \alpha_2 \cdots \underset{lm}{\Rightarrow} \alpha_{n-1} \underset{lm}{\Rightarrow} w$ 와 같이 구성됨.

이를 $S \overset{*}{\underset{lm}{\Rightarrow}} w$ 로 표기함. $S \underset{lm}{\Rightarrow} \alpha$일 때, α는 left sentential form 이다.

eg) $S \Rightarrow iCtS$
$S \Rightarrow iCtSeS$
$S \Rightarrow a$
$C \Rightarrow b$

$S \underset{lm}{\Rightarrow} iCtS$
$\underset{lm}{\Rightarrow} ibtS$
$\underset{lm}{\Rightarrow} ibtiCtSeS$
$\underset{lm}{\Rightarrow} ibtibtSeS$
$\underset{lm}{\Rightarrow} ibtibtaeS$
$\underset{lm}{\Rightarrow} ibtibtaea$



"왼쪽 Nonterminal 부터 생성에 의해 대치됨"

2) right most derivation

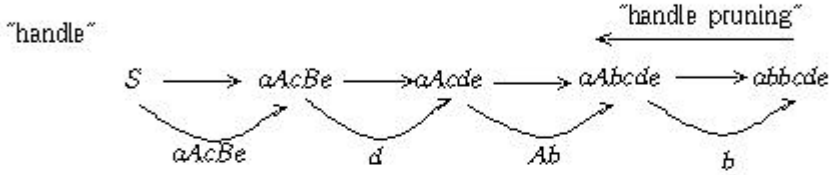
$\alpha \underset{rm}{\Longrightarrow} \beta$, $S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha$ 일 때, α를 right sentential form

주어진 입력 스트링으로부터 production에 의하여 tree를 leaf에서 root를 향하여 구성.

- $A \rightarrow \beta$는 A가 β를 reduction 해 나간다고 하며, β는 A로 대치됨.
- reduction을 위하여 "handle(production규칙의 오른쪽과 일치하는 부분 문자열)"이라고 하는 것을 발견하고, 이를 반복하여 역유도

$S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A w \underset{rm}{\Longrightarrow} \alpha \beta w$, β가 "handle"

eg) $S \Rightarrow aAcBe$
$A \Rightarrow Ab \mid b$
$B \Rightarrow d$



어떤 생성에서 한 개 이상의 "handle" 이 생기면 "ambiguous" 하다.

eg) $E \Rightarrow E + E$
$E \Rightarrow E * E$
$E \Rightarrow$ id

입력 string id1 + id2 * id3 에 대해서

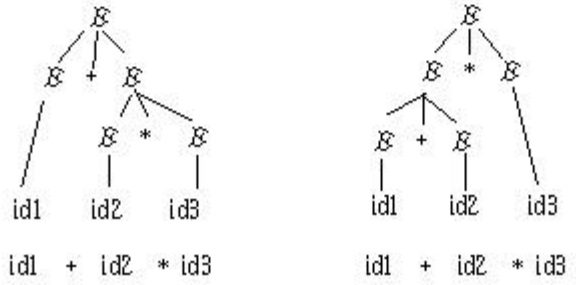
$E \underset{rm}{\Longrightarrow} \underline{E + E} \underset{rm}{\Longrightarrow} E + \underline{E * E} \underset{rm}{\Longrightarrow} E + E * \underline{id3}$

$\underset{rm}{\Longrightarrow} E + \underline{id2} * id3 \underset{rm}{\Longrightarrow}$ id1 + id2 * id3

ambiguous

$E \underset{rm}{\Longrightarrow} \underline{E * E} \underset{rm}{\Longrightarrow} E * \underline{id3} \underset{rm}{\Longrightarrow} \underline{E + E} * id3$

$\underset{rm}{\Longrightarrow} E + \underline{id2} * id3 \underset{rm}{\Longrightarrow} \underline{id1} + id2 * id3$



w가 어떤 문법의 문장이며, r_n 이 어떤 right most derivation의 n번째 right sentential form으로 w를 나타내면 $w = r_n$ 이다. 각 right most derivation에서 그때의 "handle"을 이용하여 바로 앞의 right sentential form을 구한다면 아래와 같이 역유도의 순서를 구할 수 있다.

$$S = r_0 \underset{rm}{\Longrightarrow} r_1 \underset{rm}{\Longrightarrow} r_2 \underset{rm}{\Longrightarrow} r_3 \cdots \underset{rm}{\Longrightarrow} r_{n-1} \underset{rm}{\Longrightarrow} r_n = w$$

eg) G: $\begin{cases} E \rightarrow E + E \\ E \rightarrow E * E \\ E \rightarrow id \end{cases}$

입력 : id1 + id2 * id3

right sentential form	handle	production
id1 + id2 * id3	id1	E → id
E + id2 * id3	id2	E → id
E + E * id3	id3	E → id
E + E * E	E * E	E → E * E
E + E	E + E	E → E + E
E		

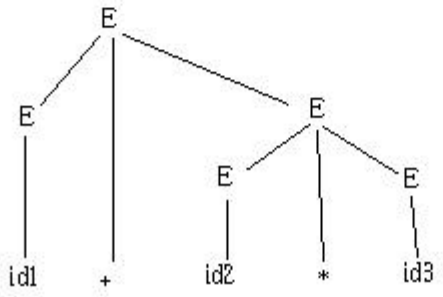
이와 같은 parsing 기법을 "Shift - reduce" parsing

Shift - Reduce Parsing 과정

- stack과 input buffer를 이용

stack	input	action
$	id1 + id2 * id3 $	shift
$ id1	+ id2 * id3 $	reduce E → id
$ E	+ id2 * id3 $	shift
$ E +	id2 * id3 $	shift
$ E + id2	* id3 $	reduce E → id
$ E + E	* id3 $	shift
$ E + E *	id3 $	shift
$ E + E * id3	$	reduce E → id
$ E + E * E	$	reduce E →E * E
$ E + E	$	reduce E → E + E
$ E	$	accept

┌ parsing이 성공하면 accept
└ 오류가 발생하면 error message



Parsing 기법(1) ⟨ ① Top-down parsing 전략 / ② Bottom-up parsing 전략

: 문법에 맞는 Program을 하나의 Syntactic tree로 표현

① Top-down parsing 기법

- Beginning with the goal symbol of the grammar, attempted to produce a string of terminal symbols that was identical to a given source string.
- Top-down parse moves from the goal symbol to a string of terminal symbols. In the terminology of tree, this is moving from the root of the tree to a set of the leaves in the syntax tree for a program.
- "left most derivation 기법"

② Bottom - up parsing 기법

- Builds a parse tree from the leaves toward its root.
- "right most derivation"

Parsing 기법 (2)

- 문법에 맞는 program을 하나의 syntactic tree로 표현

- Top-down parsing
  - Brute - Force Approach, Recursive-Descent, LL(1)
  - move from the goal symbol to a string of terminal symbols
- Bottom-up parsing
  - Operator Precedence Grammars
  - Simple Precedence Grammars
  - LR Grammar
    - LR(0) pasers
    - SLR(1) parsers
    - Canonical LR(1) parsers
    - LALR(1) parsers

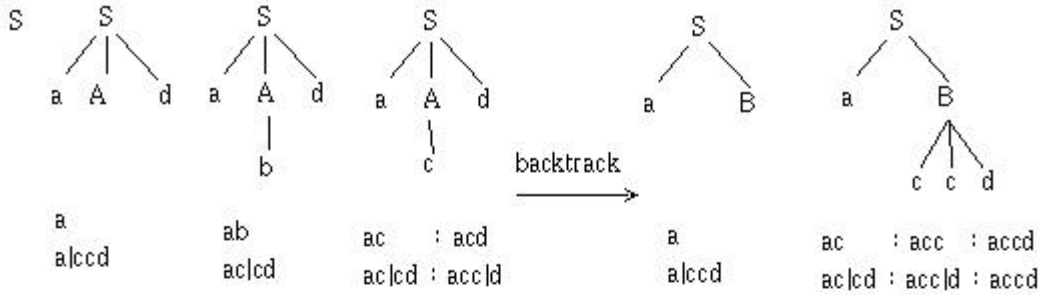
1 Top-down parsing

(=Backtrack)

1. 1 Top-down parsing with full Backup ("Brute-Force Approach")

eg) $S \rightarrow aAd|aB$ $A \rightarrow b|c$ $B \rightarrow cd|dc$

- S : goal (start symbol)
- → 'accd' : input



i) 주어진 Nonterminal에 대해서, 첫 번째 production rule를 적용하여 확장 string 생성
ii) 새로이 생성된 string의 left-most Nonterminal에 대해서 첫 번째 확장 production rule를 적용
iii) 더 이상 Nonterminal이 발견되지 않으면

→ * parsing 이 성공이거나
* source string(input)과 match하지 않으면 the process is backed up b undoing the most recently applied production.

→ "very time - consuming"

- parsing중에 input과 tree의 leaf과 일치가 되지 않으면 어디에서부터 다시 반복을 해서 parsing을 해야하는지 backtracking의 범위를 알 수가 없으며, 특히 left-recursion인 경우는 반복의 순환이 생겨서 복잡한 문제가 발생함.

→ Backtracking를 피하기 위해 Recursive - Descent Parsing 기법 이용

1. 2 Recursive-Descent Parsers (predictive parsing)

: top-down parsing which does not allow backup. But, certain grammar require backup in order for successful parsing to occur.

- a sequence of production application = a sequence of function calls
- functions are written for each nonterminal
- Each function returns a value of True of False depending on whether or not it recognizes a substring which is an expansion of that nonterminal.

- right-recursive rule grammar
- longest rule first

예제)

<factor> ::= (<expr>) | i

<term> ::= <factor> * <term> | <factor>

<expr> ::= <term> + <expr> | <term>

```
1.[Initialize]
Read(INPUT)
2.[Loop through all input strings]
    Repeat while there still remains an input string
      Repeat for i=1, 2. ...., LENGTH(INPUT)
        STRING[i] ← SUB(INPUT, i, 1)
      CURSOR ← 1
      NEXT ← GET_CHAR
      If EXPR
      then If NEXT = '#'
           then Write(INPUT, '□VALID')
           else Write(INPUT, '□INVALID')
      else Write(INPUT, '□INVALID')
      Read(INPUT)
    Exit

Function EXPR

1. [<expr> ::= <term> + <expr> | <term>]
     If not TERM
     then Return(false)
     If NEXT = '+'
     then NEXT ← GET_CHAR
       If NEXT = '#'
       then Return(false)
       If not EXPR
       then Return(false)
       else Return(true)
    else Return(true)

Function TERM

1. [<term> ::= <factor>*<term>|<factor>]
       If not FACTOR
       then Return(false)
       If NEXT = '*'
```

```
        then NEXT ← GET_CHAR
                If NEXT = '#'
                then Return(false)
                If not TERM
                then Return(false)
                else Return(true)
        else Return(true)
```

Function FACTOR

```
1. [<factor> ::= (<expr>) | i]
        If NEXT = '#'
        then Return(false)
        If NEXT = '('
        then NEXT ← GET_CHAR
                If NEXT = '#'
                then Return(false)
                If not EXPR
                then Return(false)
                If NEXT ≠ ')'
                then Return(false)
                else NEXT ← GET_CHAR
                        Return(true)
        If NEXT ≠ 'i'
        then Return(false)
        else NEXT ← GET_CHAR
                Return(true)
```

Function GET_CHAR

```
1. [Returns the next character from the input string]
        CHAR ← STRING[CURSOR]
        CURSOR ← CURSOR + 1
        Return(CHAR)
```

A trace of the parse for the input

(*i* + *i*) * #

```
Input: (i+i) * i#
Perform MAIN
        Call EXPR
          Call TERM
                Call FACTOR
                   check for #, No.
                   check for (, Yes, h ← 2.
                   check for #, No.
                   Call EXPR
                         Call TERM
                            Call FACTOR
                                 check for #, No.
                                check for (, No.
                                 check for i, Yes, h ← 3.
                                   Return true from FACTOR.
                            check for *, NO.
                            Return true from TERM
                         check for +, Yes, h ← 4.
                         check for #, No.
                         Call EXPR
                            Call TERM
                                  Call FACTOR
                                     check for #, No.
                                      check for (, No.
                                      check for i, Yes, h ← 5.
                                      Return true from FACTOR.
                                  check for *, No.
                                  Return true from TERM.
                              Return true from EXPR.
                         Return true from EXPR.
                   check for ). Yes. h ← 6
                   Return true from FACTOR.
                check for *, Yes, h ← 7.
                check for #, No.
                Call TERM
                   Call FACTOR
                         check for #, NO.
                         check for (, No.
                         check for i, Yes, h ← 8
                         Return true from FACTOR.
                   check for *, No
                   Return true from TERM.
              Return true from TERM.
          check for +, No.
          Return true from EXPR.
        check for #, Yes, h ← 9.
        Return "VALID" from MAIN.
```

< Trace of Algorithm RECDSNT for the string (i + i)*i# >

Left Recursion

- Recursive - Descent parser는 무한 loop로 빠질 수가 있다.

eg) expr --> expr + term

여기서 production rule 오른쪽의 맨 왼쪽기호는 왼쪽의 nonterminal과 동일하다. 만약 expr을 위한 procedure(function)가 (eg)와 같은 production rule을 사용할 경우에, 생성규칙의 오른쪽이 expr로 시작되기 때문에 expr을 위한 function을 recursive하게 자신을 실행시킬 것이다. 그리고 이러한 과정은 무한 loop에 빠지게 된다.

<left recursion 제거>

$A \to A\alpha \mid \beta$

제거 ⟹ $A \to \beta A'$: right − recursion

$A' \to \alpha A' \mid \varepsilon$

eg) G: $E \to E + T \mid T$, $T \to T * F \mid F$, $F \to (E) \mid id$

⟹

$A = E$, $\alpha = +T$, $\beta = T$: $E \to TE'$, $E' \to +TE' \mid \varepsilon$

$A = T$, $\alpha = *F$, $\beta = F$: $T \to FT'$, $T' \to *FT' \mid \varepsilon$

$F \to (E) \mid id$

일반화

A- 생성식들이 아무리 많다고 해도 아래와 같은 기법을 사용하면 그러한 직접- 왼쪽 순환을 제거할 수 있다. 우선 아래와 같이 A- 생성식들을 묶을 수 있다.

$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

여기에서 어떠한 β_i도 A로 시작하지 않는다. 이제 A- 생성식들은 아래와 같이 고쳐쓸 수 있다.

$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$
$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \varepsilon$

Left Factoring

Left Factoring이란 문법 변환의 하나로서 predictive parsing에 알맞는 문법을 만드는데 매우 효과적이다. 기본적인 생각은 nonterminal A에 대한 두 개의 생성식들 중, 어떤 것을 사용해야 할지가 불확실할 때, 옳은 선택을 할 수 있도록 입력을 충분히 읽을 때까지 그 선택을 연기할 수 있도록 A-생성식들을 변경시키는 것이다.

예를 들어 아래와 같은 두 개의 생성식을 보자.

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

입력 토큰 **if**를 읽어들인 상태에서는 *stmt*를 확장하는데 어느 생성식을 사용해야할지를 즉시 결정할 수 없다. 일반적으로 만약 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$와 같이 두 개의 A- 생성식이 있으면, 입력이 α로부터 유도된 공백이 아닌 문자열로 시작하기 때문에 A를 $\alpha\beta_1$이나 $\alpha\beta_2$ 중 어느 것으로 확장해야 할지 알 수가 없다. 그러나 A를 $\alpha A'$로 확장을 하면 이러한 결정을 연기시킬 수 있다. 그리고 나서 α에서 유도된 입력을 보고나서, A'를 β_1이나 β_2로 확장시킨다. 이것이 바로 left factored 된 것이다. 원래의 생성식들은 아래와 같이 변화된다.

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \xrightarrow{\text{left factoring}} \begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \end{array}$$

예제) 아래의 문법은 매달린 else 문제를 일반화한 것이다.

$S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

여기서 i, t 와 e는 각각 **if**, **then** 그리고 **else**를 뜻하고 E와 S는 "표현식"과 "명령문"을 표시한다. 이 문법을 Left-factoring하면 아래와 같이 된다.

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \varepsilon$
$E \rightarrow b$

Predictive Parsing :

- Recursive-Descent Parsing의 특별한 방법으로서 "lookahead"(예측기호)를 이용하기 때문에 각 Nonterminal을 처리하는 function을 선택할 때 모호함이 없다. 입력을 처리하는 function들의 실행 순서가 곧바로 그 입력에 대한 parse tree를 구성한다.
  cf) "lookahead" = 입력문자열에서 현재 처리되고 있는 token.

- Backtracking을 하지 않고 parsing할 수 있음

  eg) type ⟶ simple
  | ↑ id
  | array [simple] of type

  simple ⟶ integer
  | char
  | num dotdot num

  array [num dotdot num] of integer

predictive parser는 nonterminal *type*과 *simple* 그리고 단순한 프로시저인 *match*로 구성되어 있다.

```
procedure match (t : token) ;
begin
        if lookahead = t then
                lookahead := nexttoken
        else error
end ;

procedure type ;
begin
        if lookahead 는 { integer, char, num } 중에 있다 then
                simple
        else if lookahead = '↑' then begin
                match ('↑') ; match(id)
        end
        else if lookahead = array then begin
                match(array) ; match('['); simple ; match (']'); match(of); type
        end
        else error
end ;

procedure simple ;
begin
        if lookahead = integer then
                match(integer)
        else if lookahead = char then
                match(char)
        else if lookahead = num then begin
                match(num) ; match(dotdot) ; match(num)
        end
        else error
end ;
```

< 예측 파서의 의사 코드(Pseudo-code) >

파싱은 문법의 비단말 *type*을 위한 프로시저를 실행시킴으로써 시작된다. 변수 *lookahead*는 처음에는 첫 입력 토큰인 **array**를 가질 것이다. 프로시저 *type*은

match(**array**); *match*('['); *simple*; *match*(']'); *match*(**of**); *type*

비단말 *type*에 대한 아래의 생성 규칙과 비교하면서 위의 코드를 실행할 것이다.

type → **array** [*simple*] **of** *type*

predictive parser는 생성 규칙의 오른쪽에 의해서 생성되는 첫 번째 기호가 무엇인가라는 정보를 파싱의 기반으로 하고 있다.

FIRST(*simple*) = { **integer**, **char**, **num** }
FIRST(↑**id**) = {↑}
FIRST(**array** [*simple*] of *type*) = { **array** }

FIRST 집합은 $A \rightarrow \alpha$와 $A \rightarrow \beta$와 같이 한 개의 nonterminal에 대해서 두 개 이상의 생성 규칙이 있다면 반드시 고려되어야 한다. 즉, backtracking이 없는 recursive descent parsing 하기 위해서는 FIRST(α)와 FIRST(β)가 고려되어야 한다.
그러면 예측 기호가 어느 생성 규칙을 사용할 것이냐를 결정하는데 사용될 수 있다. 예를 들어 예측 기호가 FIRST(α) 집합에 속한다면, α가 사용될 것이고, 그렇지 않고 예측 기호가 FIRST(β) 집합에 속한다면, β가 사용될 것이다.

언제 ε생성 규칙을 사용할 것인가?

오른쪽에 ε가 있는 생성 규칙은 특별한 처리가 필요하다. recursive-descent parser는 사용할 생성 규칙이 하나도 없을 때 기본적으로 ε 생성 규칙을 사용한다. 예를 들어 아래를 보자.

stmt → **begin** *opt_stmts* **end**
opt_stmts → *stmt_list* | ε

여기서 *opt_stmts*를 파싱할 때 예측 기호가 FIRST(*stmt_list*) 집합에 속하지 않는다면 ε 생성 규칙이 사용될 것이다. 만약 예측 기호가 **end**라면 이러한 선택은 올바른 것이다. *stmt*를 파싱할 때 **end** 이외에 다른 예측 기호가 들어왔다면 분명히 오류(error)로 판명될 것이다.

predictive parser의 설계

predictive parser 프로그램은 여러 개의 nonterminal을 위한 프로시저로 구성되어 있다.
각 프로시저들은 아래의 두 가지 일을 한다.
1. lookahead symbol을 읽고서 어떤 생성 규칙을 사용할 것인지를 결정한다. 이 때 lookahead가 FIRST(α) 집합에 속한다면, α를 오른쪽에 가진 생성 규칙을 사용할 것이다. 만약 어떤 lookahead에 대해서 두 개의 생성 규칙이 존재해서 어떤 것을 사용할 것인지를 결정할 수 없다면 이 문법은 predictive parsing 방법을 사용할 수 없다. 만약 예측 기호가 어떠한 생성 규칙의 오른쪽에 대한 FIRST 집합에도 속하지 않는다면 오른쪽에 ε가 있는 생성 규칙이 사용된다.
2. 각 프로시저는 생성 규칙의 오른쪽을 흉내내듯이 차례대로 입력을 처리한다. 즉, 생성 규칙의 오른쪽에 있는 nonterminal에 대해서는 그것에 대한 프로시저를 수행시키고, lookahead와 일치하는 토큰들은 다음 입력을 읽어들이게 한다. 만약, 처리 중에 생성 규칙에 있는 token과 lookahead가 일치하지 않는다면 이것은 오류라고 판정한다.

① Parsing Table 생성

1. predictive parsing table

predictive 파싱 방법은 recursive-descent 방법을 효과적으로 처리하기 위하여 스택을 이용하는데 이 파서는 입력, 스택 그리고 파싱표를 갖게된다.

입력과 스택의 제일끝지에 있는 단계가 파싱표에 의하여 자동으로 다음단계를 정하면서 파싱을 계속하게 된다. 이 파싱표를 만들기 위하여는 FIRST와 FOLLOW를 구하게 되는데 이들은 다음과 같이 정의되고 있다.

FIRST(α) : 만일 α가 문법기호의 어떤 스트링이라면 FIRST(α)는 α로부터 유도되는 스트링을 시작하는 terminal들의 집합이다. 또한 $\alpha \overset{*}{\Rightarrow} \varepsilon$ 이라면 ε은 또한 FIRST(α)에 속하게 된다.

따라서 FIRST(α)를 계산하는 방법은 다음과 같이 정할 수가 있다.

(1) 만일 α가 하나의 terminal이면 {α}는 FIRST(α)이다.

(2) 만일 α가 하나의 nonterminal이며 $\alpha \rightarrow a\beta$가 생성(生成)이면 terminal a는 FIRST(α)에 속하게 되고 또한 $\alpha \rightarrow \varepsilon$의 생성(生成)이 존재하면 ε도 FIRST(α)에 속하게 된다.

(3) 만일 $\alpha \rightarrow Y_1 Y_2 \ldots Y_K$의 생성이 있고 $Y_1 \ldots Y_{j-1}$가 nonterminal이며 $Y_1 Y_2 \ldots Y_{j-1} \overset{*}{\Rightarrow} \varepsilon$ 라면 FIRST(Y_j)는 FIRST(α)에 속하게 되나 ε은 제외한다. 또한 ε이 FIRST(Y_j), j = 1, 2, ..., K이면 ε은 FIRST(α)에 속하게 된다.

예를 들어 다음 문법에서

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \mathrm{id} \end{array} \quad \text{left recursion제거} \Longrightarrow \quad \begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \varepsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \varepsilon \\ F \rightarrow (E) \mid \mathrm{id} \end{array}$$

규칙 (1)에서 모든 터미널은 그 자체가 FIRST가 된다는 뜻이며[FIRST(F) = id], 규칙 (2)는 생성 $\alpha \rightarrow a\beta$에서 터미널 a가 FIRST($\alpha$)가 되는데, $\alpha \rightarrow a\beta$의 모양을 한 생성은

① $E' \rightarrow +TE'$, $T' \rightarrow *FT'$, $F \rightarrow (E)$, $F \rightarrow \mathrm{id}$

등인데, +는 FIRST(E'), *은 FIRST(T'), (는 FIRST(F), id 는 FIRST(F)에 속함를 알 수 있다. 규칙 (3)에 의하면 $\alpha \rightarrow Y_1 Y_2 \ldots Y_K$ 에서 FIRST(Y_1)은 FIRST(α)에 속하는데, 센텐셜 폼 $\alpha \rightarrow Y_1 Y_2 \ldots Y_K$ 의 모양을 한 생성은

② $E \rightarrow TE'$, $T \rightarrow FT'$

이며, 따라서 FIRST(T) ⊂ FIRST(E), FIRST(F) ⊂ FIRST(T) 가 되므로 FIRST(F) ⊂ FIRST (T) ⊂ FIRST(E)가 된다. ③ $\alpha \rightarrow \varepsilon$를 FIRST로 갖는 생성은

$E' \rightarrow \varepsilon$, $T' \rightarrow \varepsilon$

이므로 ε은 FIRST(E'), FIRST(T')에 각기 속하게 된다. 따라서

$$\text{FIRST}(E') = \{+, \varepsilon\}$$
$$\text{FIRST}(T') = \{*, \varepsilon\}$$
$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, id\}$$

가 됨을 알 수가 있다.

FOLLOW(A) : 어떤 nonterminal A에 대하여 sentencial form에서 A의 오른쪽에 나타나는 terminal의 집합을 말하는데, 만일 A가 sentencial form에서 가장 오른쪽 기호이면 $(입력의 끝표시)이 포함된다. FOLLOW($A$)는 다음과 같이 계산한다.

(1) 만일 A가 start symbol이면 $는 FOLLOW(A)에 포함된다.
(2) 생성 $A \rightarrow \alpha B \beta$, $\beta \neq \varepsilon$에 대하여는 ε를 제외한 FIRST(β)는 FOLLOW(B)에 속한다.
(3) 생성 $A \rightarrow \alpha B$ 혹은 $A \rightarrow \alpha B \beta$에서 만일 $\beta \rightarrow \varepsilon$이면 FOLLOW($A$)는 모두 FOLLOW($B$)에 속한다.

위의 문법에서 FOLLOW를 계산하여 보자.
규칙 (1)에 의하여 $는 FOLLOW($E$)에 속하며,
규칙 (2)에 의하여 $A \rightarrow \alpha B \beta$, $\beta \neq \varepsilon$의 모양을 한 생성은

$$F \rightarrow (E),\ T' \rightarrow *FT',\ E' \rightarrow +TE'$$

인데, FIRST(β)가 FOLLOW(B)에 속하게 되므로

$$\text{FOLLOW}(E) \supset \text{FIRST}()) = \{)\}$$
$$\text{FOLLOW}(F) \supset \text{FIRST}(T') = \{*\}\ (\varepsilon\text{은 제외})$$
$$\text{FOLLOW}(T) \supset \text{FIRST}(E') = \{+\}\ (\varepsilon\text{은 제외})$$

를 얻게 된다.
규칙 (3)에 의하여 $A \rightarrow \alpha B$의 모양을 한 생성은

$$E \rightarrow TE',\quad T \rightarrow FT'$$

인데, FOLLOW(A)는 모두 FOLLOW(B)에 속하므로

$$\text{FOLLOW}(E) \subset \text{FOLLOW}(E')$$
$$\text{FOLLOW}(T) \subset \text{FOLLOW}(T')$$

또한 규칙 (3)에서 $A \rightarrow \alpha B \beta$, $\beta \overset{*}{\Longrightarrow} \varepsilon$ 의 모양의 생성은

$$E \rightarrow TE',\ E' \rightarrow \varepsilon \text{과 } T \rightarrow FT',\ T' \rightarrow \varepsilon$$

인데, FOLLOW(A)는 FOLLOW(B)에 속하므로

$$\text{FOLLOW}(E) \subset \text{FOLLOW}(T)$$
$$\text{FOLLOW}(T) \subset \text{FOLLOW}(F)$$
$$\text{FOLLOW}(E) \subset \text{FOLLOW}(T) \subset \text{FOLLOW}(F)$$

이 성립되므로 다음 계산이 이루어진다.

FOLLOW(E) = { $,) }
FOLLOW(T) = { $,), + }
FOLLOW(F) = { $,), +, * }

또한 FOLLOW(E')과 FOLLOW(T')를 별도로 계산할 수 있는 기회가 없기 때문에

FOLLOW(E') ⊃ FOLLOW(E), FOLLOW(T') ⊃ FOLLOW(T)

의 식에서

FOLLOW(E') = FOLLOW(E) = { $,) }
FOLLOW(T') = FOLLOW(T) = { $,), +}

를 얻을 수가 있다.

이들 FIRST와 FOLLOW를 이용하여 다음과 같이 predictive parsing table을 구성한다.

(1) $A \rightarrow \alpha$의 생성에 대하여 (2), (3)을 수행한다.

(2) FIRST(α)에 있는 터미널 a에 대하여 M[A, a]에 $A \rightarrow \alpha$를 넣으며

(3) FIRST(A)에 속하는 터미널 b에 대하여 만일 ε이 FIRST(α)에 속하면 $A \rightarrow \alpha$를 M[A,b]에 넣는다

만일 ε이 FIRST(α)이며 $이 FOLLOW($A$)에 속하면 $A \rightarrow \alpha$를 M[A, $]에 넣는다

(4) 나머지 빈 곳은 error로 정한다.

규칙 (2)에 의하여 $A \rightarrow \alpha$에서 FIRST(α)를 구해 보자. 센텐셜 폼 α는

{TE', $+TE'$, ε, FT', $*FT'$, ε, (E), id}

들인데, 먼저 TE'를 이용하여 FIRST(TE')를 계산하면 규칙 (2)에 의하여

FIRST(TE') = FIRST(T) = { (, id }

를 얻을 수 있어 생성 $E \rightarrow TE'$는

M[E, ( ], M[E, id]

에 들어감을 알 수 있으며, 다음 $+TE'$에 대하여

FIRST($+TE'$) = { + }

이므로 생성 $E' \rightarrow +TE'$는

M[E', +]

에 들어간다.

다음 FT'에 대하여

FIRST(FT') = FIRST(F) = {(, id}

를 얻어 생성 T→FT'는

M[T, (], M[T, id]

에 들어가게 된다.

다음 $*FT'$에 대하여

FIRST($*FT'$) = {*}

이므로 생성 $T' \rightarrow *FT'$는

M[T', *]

에 들어가게 된다.

또한 ($)와 id에 대하여

FIRST((E)) = {(}

FIRST(id) = {id}

이므로 생성 F→(E)와 F→id는 각기

M [F, (]

M [F,id]

에 들어가게 된다.

규칙 (3)에 의하여 생성 $A \rightarrow \alpha$ 에 대하여 ε 이 FIRST(α)에 속하는 A는

E´ →ε, T´→ε

의 경우므로, 이들의 FOLLOW(A)를 보면

FOLLOW(E´)={$,) }

이므로 생성 E´→ ε은

M[E´,)]과 M[E´, $]

에 들어가게 되며,

FOLLOW(T´)={$,) ,+ }

이므로 생성 T´→ε은

M[T´, +], M[T´,)], M[T´, $]

에 들어가서 parsing table를 만들게 된다.

이들을 모아서 표를 만들면 아래와 같다.

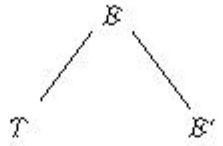
< parsing table >

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→ +TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

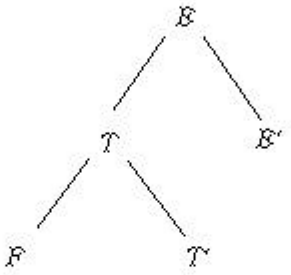
② Parse Tree 생성(p56 그림6참조)

파싱표에 의하여 스택과 입력으로 파싱이 되는 과정을 살펴보자.

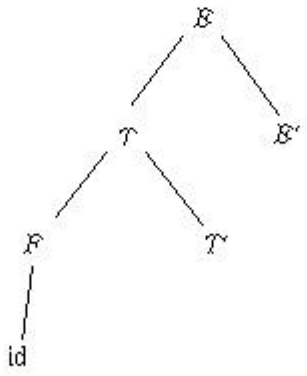
(1) 에서는 스택에 $E로 시작을 하게 되며 id + id*id $의 입력을 맞아 스택의 꼭지에는 $X=E와 입력은 a = id가 되므로 파싱표에서 $A[E, id] = $E \rightarrow TE'$를 얻게 되므로 스택은 (2)에서와 같이 되며 $E \rightarrow TE'$의 tree를 만들어 주게 된다.



(2) 에서는 $X = T, a = id이므로 $A[T, id] = $T \rightarrow FT'$를 얻게 되고, 스택은 (3)과 같이 되며 다음과 같은 나무구조를 얻게 된다.

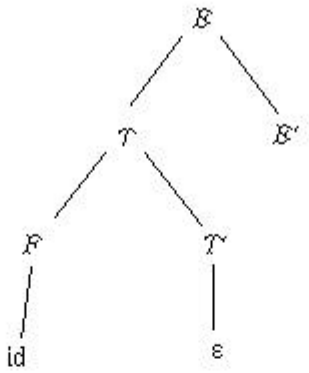


(3) 에서는 $X = F, a = id이므로 $A[F, id] = $F \rightarrow$id가 되어 나무구조는 다음과 같이 변하게 된다.

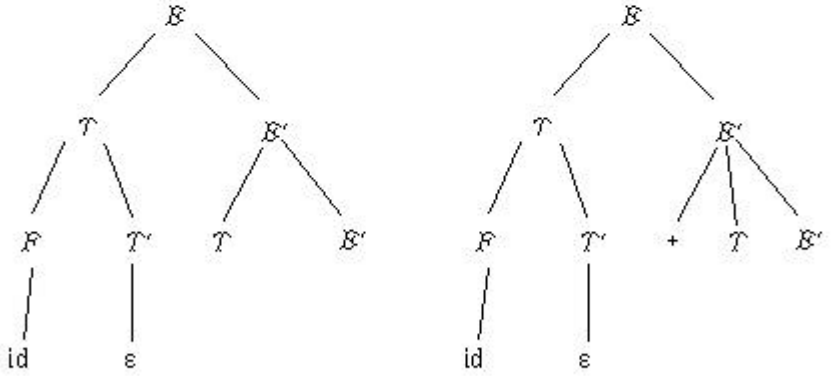


(4) 에서는 $X = id이므로 규칙 (2)에 의하여 X=id는 스택에 들어내고 입력 포인터는 하나 우진(右進)하여 (5)와 같은 모양이 된다.

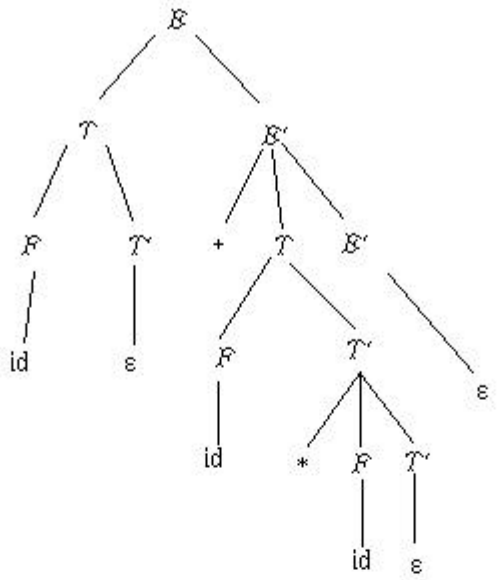
(5) 에서는 $X \rightarrow T', a = +이므로 $A[T', +] = $T' \rightarrow \varepsilon$이므로 나무구조는 다음과 같이 된다.



(6) 에서는 $X = E'$, $a = +$이므로 $A[E', +] = E' \rightarrow + TE'$이다. 스택은 (7)과 같이 되며 그 나무 구조는 다음과 같이 된다.



(7) 에서는 $X = +$이므로 규칙 (2)에 의하여 스택의 +는 지워지고, 입력 포인터는 하나 우진(右進)하게 되어 (8)과 같이 된다. 이와 같은 과정을 거쳐 파싱표와 입력 id + id*id를 이용한 top-down파싱은 다음과 같다.



그 잎을 모아서 id ε+ id * id ε ε, 즉 id + id *id 를 얻게 된다. 파싱하는 과정에서 실제로 필요한 코우드를 생성하는 시맨틱 해석은 생략하였으며 뒤에서 구체적인 설명이 있게 되겠다.

p.222

⑤ 예측 파서(Predictive Parser)

문법을 주의해서 만들고 왼쪽 순환성을 제거한 다음 left factoring 처리를 하면, 전혀 백트랙킹이 필요 없는 순환적 내림차순 파서, 예측 파서가 분석할 수 있는 문법을 얻을 수 있다. 이러한 예측 파서를 만들기 위해서는 현재입력 기호 a 와 확장되어야 할 비단말 A, 그리고 이것에 대한 선택적인 생성식들 $A \rightarrow \alpha_1 \mid \alpha_2 \cdots \mid \alpha_n$이 주어졌을 때, a 로 시작하는 문자열을 유도시키는 유일한 생성식을 알아야 한다. 예를 들어, 아래와 같은 생성식을 가졌다고 하면,

stmt → **if** *expr* **then** *stmt* **else** *stmt*
| **while** *expr* **do** *stmt*
| **begin** *stmt_list* **end**

여기서, 키워드 if, while 그리고 begin만을 읽으면 어느 생성식을 선택할 것인가를 알 수 있다.

예측 파서에 대한 전이 다이어그램

예측 파서의 설계 도구로서 전이 다이어그램을 이용. 각 비단말에 대해서 하나의 다이어그램이 있고, edge들의 이름은 토큰이거나 비단말들이다. 토큰(단말)에 대한 전이는 그 토큰이 다음 입력 기호일 때, 그러한 이동이 일어난다는 뜻이다. 비단말 A에 대한 전이는 A에 대한 프로시저를 수행시키는 것이다. 어떤 문법의 예측 파서에 대한 전이 다이어그램을 구성하기 위해서는 우선 그 문법에서 왼쪽 순환성을 제거하고 문법을 left factoring해야 한다.

각 비단말 A에 대해서 아래를 수행 상태 s에서 상태 t로 가는 이름이 단말 a인 연결선이 있을 때, 다음 입력 기호가 a이면 파서는 입력 포인터를 하나 오른쪽으로 움직이고 상태는 t가 된다. 한

편, 연결선의 이름이 비단말 A이면 파서는 입력 포인터를 움직이지 않고 대신에 A에 대한 시작 상태로 간다. 만약 계속 진행해서 A에 대해 끝나는 상태에 도착하면 즉시 상태 t로 가는데, 결과적으로 상태 s에서 상태 t로 가는 동안 입력에서 A를 읽어들인 효과가 발생한다. 마지막으로 만약 ε에 의해서 s에서 t로 가는 연결선이 있다면 파서는 입력에서 진행 없이 상태 s에서 상태 t로 즉시 움직인다.

비단말 기호에 대한 연결선을 이동해야 할 때는, 순환적으로 그것을 처리하는 프로시저를 부르게 된다. 한편, 비순환적인 구현도 가능한데 이는 상태 s밖으로 비단말에 대한 전이가 있을 때, 그 s를 스택에 저장하고 그 비단말에 대한 끝나는 상태에 도달했을 때는 스택에서 상태를 다시 복구시키는 방법을 이용하면 된다.

위의 방법은 어떤 상태에서 같은 입력에 대해 한 가지 이상의 전이를 가지지 않을 때에 한해서다. 만약 그 비결정이 제거되지 못한다면 예측 파서를 만들 수 없다. 하지만 그렇지 않다면 모든 가능성을 체계적으로 백트랙킹을 하는 순환적 내림차순 파서를 만들 수 있을 것이다. 물론 그것은 최고의 파싱 전략을 가지고 있어야만 한다.

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

문법 1

(ᴇg) 아래의 그림 1은 문법 1에 대한 전이 다이어그램들이다. 단 한가지의 모호성은 ε-연결선을 선택할 것인지의 여부이다. 만약 E'의 시작 상태에서 연결선을 선택할 때, 다음 입력이 +일 때에만 그쪽을 선택하고, 그렇지 않을 때는 ε 연결선을 선택하기로 하고 T'에 대해서도 비슷한 가정을 하면 모호성은 제거된다.

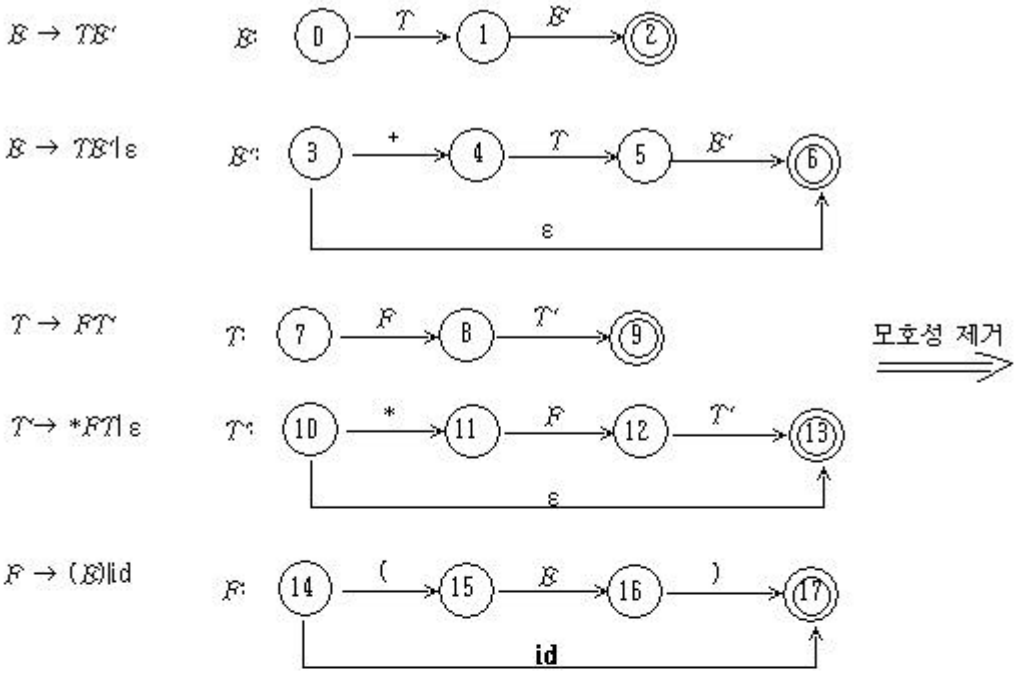


그림 0. 문법1에 대한 전이 다이어그램

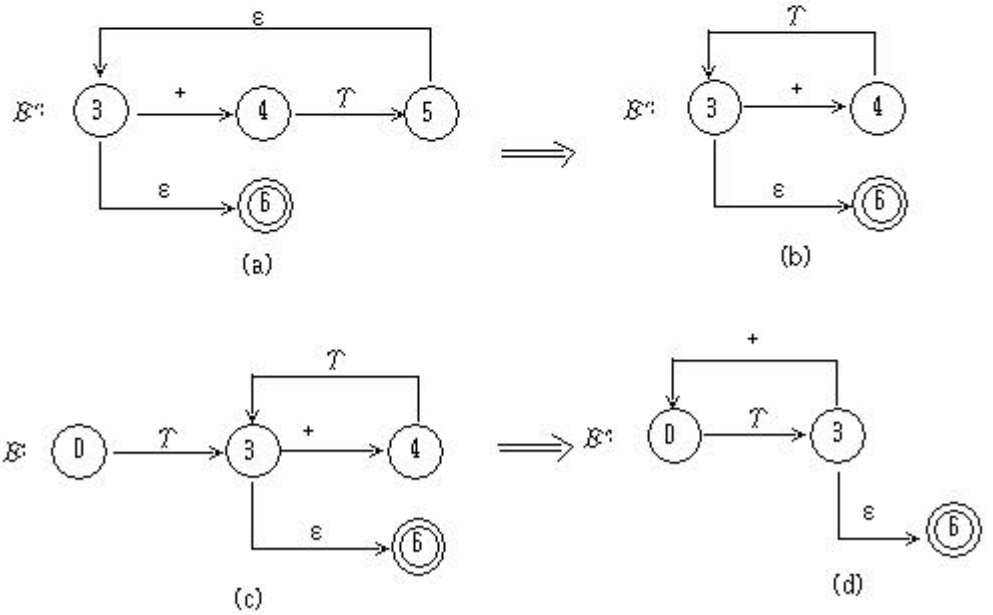


그림 1. 단순화된 전이 다이어그램

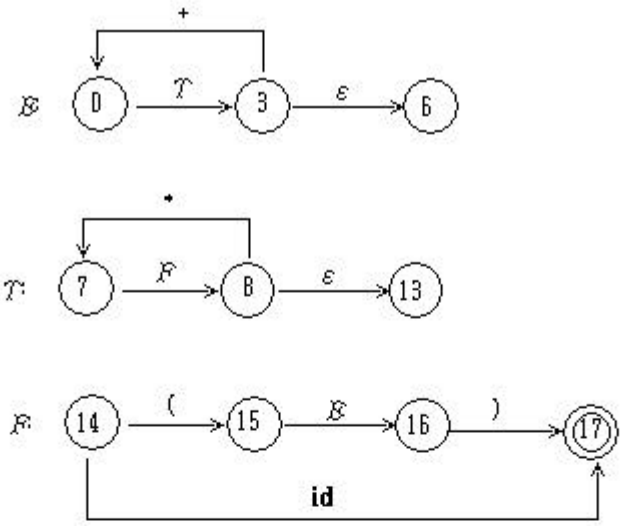


그림2.수학 연산식을 위한 단순화된 전이 다이어그램

비순환적 예측파서

물론실한 순환 함수(recursive call)를 사용하는 것보다 확실하게 스택을 이용해서 비순환적인 예측 파서를 만드는 것이 가능하다. 예측 파싱할 때 제일 중요한 문제는 하나의 비단말에 대해서 어떤 생성식을 사용할 것인지를 결정하는 것이다. 비순화적 파서는 생성식을 파싱테이블에서 찾는다. 주어진 문법에서 어떻게 테이블을 직접 만들어내는가를 배우게 된다.

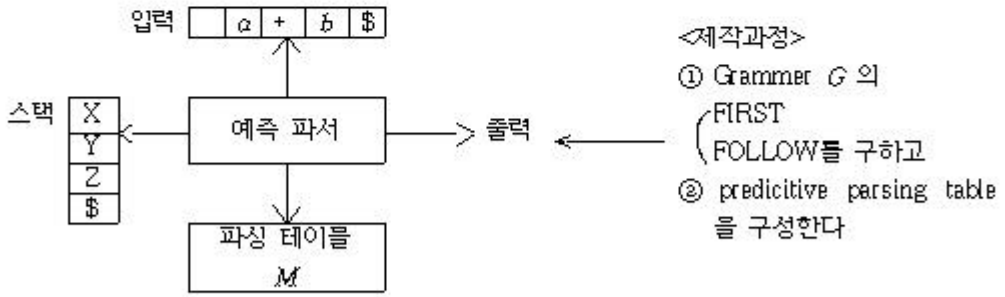


그림 3. 비순환적 예측 파서의 모델

① 입력 버퍼는 분석될 문자열을 가지고 있는데 맨 끝에 $기호가 있어서 이것은 오른쪽 끝표시로 사용되는 기호로서 입력 문자열의 끝을 나타낸다.

② 스택은 연속적인 문법 기호들과 맨 아래의 $ 기호를 가지는데 이것은 스택의 맨 아래를 나타낸다. 맨 처음에 스택에는 $위에 문법의 시작 기호가 있다.

③ 파싱 테이블은 2차원 배열 M[A, a]인데, A는 비단말이고, a는 단말이거나 기호 $이다.

파서 프로그램은 스택의 맨 위에 있는 기호 X와 현재의 입력 기호인 a를 처리한다. 이러한 두 기호는 파서의 행동을 결정하는데 아래와 같은 세 가지의 가능성이 있다.

1. 만약 X = a = $이면, 파서는 정지하고 분석이 완벽하게 끝났음을 알린다.
2. 만약 X = a ≠ $이면, 파서는 스택에서 X를 꺼내고 입력 포인터를 다음 입력 기호로 전진시킨다.
3. 만약 X가 비단말이면 프로그램은 파싱 테이블 M의 M[X,a]의 항목값을 참조한다. 이 항목 값은 문법에서의 X-생성식이거나 오류일 것이다. 예를 들어 만약 M[X,a] = {X → UVW}이라면, 파서는 스택의 맨 위에 있는 X를 WVU(맨 위는U이다)로 대치시킨다. 이때 파서가 사용한 생성식을 출력한다고 가정한다. 즉, 어떠한 다른 내용이 여기서 실행될 수도 있다. 만약 M[X,a] = error이면 파서는 오류 복구 함수를 부르게 된다.

파서의 행동은 스택과 남아있는 입력들로 이루어진 구성상황으로 표현될 수 있다.

① Parsing Table생성

비단말 기 호	입 력 기 호					
	id	+	*	(	)	$
E	E →TE´			E→TE´		
E´		E´→+TE´			E´→ε	E´→ε
T	T→FT´			T→FT´		
T´		T´→ε	T´→*FT´		T´→ε	T´→ε
F	F→id			F→(E)		

그림 5. 문법 1에 대한 파싱 테이블 M

② Parse tree 생성

입력 id + id * id에 대해서, 예측 파서는 아래 표와 같은 순서로 수행될 것이다. 입력에 대한 맨 왼쪽 유도를 추적하는 것임을 알 수 있다. 즉, 생성식이 출력한 것은 맨 왼쪽 유도의 그것이다.

스 택	입 력	출 력
$$E$	**id** + **id** * **id** $	
$$E'T$	**id** + **id** * **id** $	$E \rightarrow TE'$
$$E'T'F$	**id** + **id** * **id** $	$T \rightarrow FT'$
$$E'T'$**id**	**id** + **id** * **id** $	$F \rightarrow$ **id**
$$E'T'$	+ **id** * **id** $	
$$E'$	+ **id** * **id** $	$T' \rightarrow \varepsilon$
$$E'T$+	+ **id** * **id** $	$E' \rightarrow +TE'$
$$E'T$	**id** * **id** $	
$$E'T'F$	**id** * **id** $	$T \rightarrow FT'$
$$E'T'$**id**	**id** * **id** $	$F \rightarrow$ **id**
$$E'T'$	* **id** $	
$$E'T'F$*	* **id** $	$T' \rightarrow *FT'$
$$E'T'F$	**id** $	
$$E'T'$**id**	**id** $	$F \rightarrow$ **id**
$$E'T'$	$	
$$E'$	$	$T' \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$

그림 6. 입력 **id** + **id** * **id** 에 대해서 예측 파서가 수행되는 과정

LL(1) grammar 는 Top-Down parser 가 만들어질 수 있는 문법으로서 LL(1) grammar 를 정의하기 전에 S-grammar를 살펴보자.

S-grammar의 정의

1. 각 production의 오른쪽은 terminal로 시작한다.
2. 어떤 nonterminal이 production의 왼쪽에 한 번 이상 나타나면 이들 production의 오른쪽은 다른 terminal로 시작되어야 한다.

다음 문법은

$$S \rightarrow pX$$
$$S \rightarrow qY$$
$$X \rightarrow aXb$$
$$X \rightarrow x$$
$$Y \rightarrow aYd$$

$X \rightarrow y$

S-grammar이며, S-grammar를 분간한다는 것은 매우 쉬운 일이라 볼 수 있다.

각 생성의 오른쪽은 모두 터미널을 갖고 있기 때문에 어떤 입력언어에 대하여 파싱을 할 때 어느 생성를 택하느냐 정하는 데 도움이 되기 때문에 S-생성은 정(定) Top-Down파싱이 가능하게 된다.

LL(1) 문법은 S-문법을 보다 일반화한 것인데, 두 개의 L은 입력스트링을 왼쪽에서 오른쪽으로 scanning 한다는 뜻과 left-most derivation를 이용한다는 뜻이다. 그리고 1은 하나의 lookahead를 이용함을 나타내고 있다. 이러한 용어는 1971년에 Knuth가 제안한 것으로 알려져 있다.

시작부호의 집합에 대하여 살펴보면 시작부호의 정의는

$$a \in S(\alpha) \Leftrightarrow \alpha \Rightarrow a\beta$$

위에서 α, β 는 스트링이며 a 는 터미널이다.

다음 문법에서 시작부호의 조합를 살펴보기로 한다.

$P \rightarrow Ac$

$P \rightarrow Bd$

$A \rightarrow a$

$A \rightarrow aA$

$B \rightarrow b$

$B \rightarrow bB$

이 때 $\{a,b\}$는 P 의 시작부호가 됨을 알아야 한다. 즉, lookahead 부호가 a이면

$P \rightarrow Ac$

의 를 이용할 수 있으며,

lookahead 부호가 b 이면

$P \rightarrow Bd$

의 생성를 이용할 수 있기 때문에 $\{a,b\}$ 는 P의 시작부호가 될 수 있어

$\{a,b\} \subset S(P)$

라고 할 수 있다.

(정의) LL(1) 문법이 되기 위해서는 어떤 nonterminal의 production의 오른쪽의 시작부호들의 조합이 상이(disjoint) 하여야 한다.

- parsing table의 각 항목이 하나의 값만 가짐.
- Left recursion 제거
  모호성 제거(left factoring)

예를 들면

$P \rightarrow AB$

$P \rightarrow BG$

$A \rightarrow aA$

$A \rightarrow \varepsilon$

$B \rightarrow c$

$B \rightarrow bB$

$G \rightarrow ?$
$S(AB) = \{a, b, d\}$
$S(BG) = \{b, d\}$

따라서 이 문법은 LL(1) 문법이 아니다.

LL(1) 문법

위의 알고리즘은 파싱 테이블 M을 만드는데 어떠한 문법 G에 대해서도 적용시킬 수 있다. 그러나 어떤 문법에 대해서는 M의 일부 항목의 값이 한 가지 이상으로 정의되기도 한다. 예를 들어 G가 왼쪽 순환적이거나 모호성이 있으면 M에는 분명히 여러 개의 값을 갖는, 최소한 한 개의 항목 값이 있다.

eg) 아래 문법을 생각해보자.

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \varepsilon$ (문법 2)
$E \rightarrow b$

이 문법에 대한 파싱 테이블은 아래 그림에 있다.

비단말 기호	입 력 기 호					
	a	b	c	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \varepsilon$ $S' \rightarrow eS$			$S' \rightarrow \varepsilon$
E		$E \rightarrow b$				

그림 7. 문법 2에 대한 파싱 테이블 M

$M[S', e]$의 항목 값에는 $S' \rightarrow eS$와 $S' \rightarrow \varepsilon$ 두 개가 있다. 왜냐하면, FOLLOW(S') = {e,\$} 이기 때문이다. 이 문법은 모호성이 있는데, 이 모호성은 입력에서 e(**else**)가 발견됐을 때 어떤 생성식을 사용할 것인지의 선택에서 발생한다. 물론 이러한 모호성은 $S' \rightarrow eS$를 선택하면 없어진다. 이러한 선택은 **else**는 제일 가까운 바로 앞의 **then**과 연결되어야 한다는 규칙에 대응되는 것이다. 만약 그렇지 않고 $S' \rightarrow \varepsilon$를 선택하면 이것은 e가 스택에 넣어지는 것과 입력에서 읽어 처리되는 것을 방해하게 된다. 그러면 이것은 명백한 잘못이다.

문법의 파싱 테이블에 어떤 항목도 오직 하나의 값만을 가지면 그런 것을 LL(1)이라고 부른다. LL(1)에서 첫 번째의 "L"은 입력을 왼쪽에서 오른쪽으로 검사하는 것을 뜻하고, 두 번째의 "L"은 맨 왼쪽 유도를 생성해 내는 것을 뜻하며, "1"은 파싱 행동을 결정하는 각 단계에서 예측 기호로 입력 기호 한 개만을 사용함을 나타낸다. 알고리즘 4, 4가 모든 LL(1) 문법 G에 대해서 G의 모든 문장 그리고 오직 G의 문장만을 파싱하는 파싱 테이블을 생성한다는 것은 증명이 가능하다.

LL(1) 문법에는 몇 가지 독특한 특징이 있다. 모호한 문법이나 왼쪽 순환성을 가진 문법은 어떠한 것도 LL(1)이 될 수 없다. 문법 G가 LL(1)이라는 것은 아래의 조건을 만족하는 생성식 $A \rightarrow \alpha \mid \beta$들이 언제라도 문법 G에 속해야만 옳은 말이다.

1. 어떠한 단말 α에 대해서도 α와 β 모두가 α로 시작하는 문장을 유도시킬 수는 없다.
2. α와 β중 많아야 하나만이 공백 문자열로 유도될 수 있다.
3. 만약 $\beta \overset{*}{\Rightarrow} \varepsilon$ 이라면, α는 FOLLOW(A)에 속하는 단말로 시작되는 어떠한 문자열로 유도되지 않는다.

확실히 수학 연산식을 위한 문법 1은 LL(1)이다. 하지만 if-then-else 명령문은 모델링한 문법 2는 그렇지 않다.

이제 남은 질문은 파싱 테이블이 한 개 이상의 항목 값을 가졌을 때, 어떻게 해야하는가에 대한 것이다. 한 가지 해결책은 문법의 모든 왼쪽 순환성 제거와 해야하는가에 대한 것이다. 한 가지 해결책은 문법의 모든 왼쪽 순환성 제거와 가능한 한 left factoring 처리로 문법을 변형함으로써, 한 개 이상의 값을 가지는 항목를 전혀 가지지 않는 파싱 테이블을 생성하는 문법을 만들어 내는 것이다. 불행하게도 위의 어떠한 변형으로도 LL(1)이 되지 않는 문법이 일부 있다. 문법 2가 대표적인 예이다. 그것의 언어는 전혀 LL(1) 문법을 갖지 않는다.

이미 보았듯이 임의로 $M[S', e] = \{S' \rightarrow eS\}$로 만들어서, 예측 파서가 문법 2를 파싱할 수 있게 했다. 일반적으로 파서가 인식하는 언어에 영향을 주지 않으면서 여러 개의 값이 정의된 항목을 한 가지 값이 정의되도록 만드는 포괄적인 방법은 없다.

예측 파싱을 사용할 때 제일 어려운 점을 예측 파서가 문법에서 자동적으로 만들어질 수 있도록, 그 언어에 대한 문법을 만드는 일이다. 물론 왼쪽 순환성의 제거나 left factoring을 하는 것이 어려운 일은 아니지만, 그런 변형들은 문법들을 읽기 어렵게 만들고 번역에 사용하는데 어렵게 만든다. 이러한 어려움을 조금이라도 줄이기 위한 파서의 일반적인 구성은, 구조를 조정할 수 있도록 예측 파서를 사용하거나 연산식에 대해서 연산자 우선 순위(4. 6 절에서 설명한다)를 사용하는 것이다. 그러나 만약 뒤에서 설명하는 LR 파서 생성기(parser generator)를 사용할 수 있다면, 예측 파싱의 모든 이점을 얻으면서 또한 자동적으로 연산자 우선 순위도 사용할 수 있다.

예측 파싱에서의 오류 복구

비순환적 예측 파서에서 스택은, 구문 분석기가 나머지 입력과 비교하게 될 단말과 비단말들을 명확하게 해 준다. 그러므로 이제는 파서 스택에 있는 기호들을 살펴 볼 것이다. 예측 파싱 도중에 발견되는 오류는 스택에 있는 단말과 다음 입력 기호가 일치하지 않을 때나, 비단말 A가 스택의 맨 위에 있고 a가 다음 입력 기호이면서 파싱 테이블의 $M[A, a]$의 항목이 비어있을 때이다.

패닉 모드 오류 복구의 기본적인 생각은 어떤 정해진 동기 토큰이 나타날 때까지 입력에서 기호들을 건너뛰는 것이다. 그것의 효과는 동기 토큰 집합을 어떻게 선택하느냐에 달려 있다. 이 집합은 실제로 발생하기 쉬운 오류에서 가능한 한 빨리 복구할 수 있도록 선택되어야 한다. 아래에 약간의 경험들이 있다.

1. 시작할 때는 FOLLOW(A)에 속한 모든 기호들을 비단말 A에 대한 동기 집합에 넣을 수 있다. 만약 FOLLOW(A)의 원소가 보일 때까지 그리고 스택에서 A를 꺼낼 때까지 토큰들을 지나면, 파싱을 계속할 수 있게된다.
2. A에 대한 동기 집합으로 FOLLOW(A)만을 사용하는 것은 충분하지 않다. 예를 들어, 만약 C에서처럼 세미콜론이 문장을 끝내는 기호이면, 명령문을 시작하는 키워드들은 연산식을 생성하는 비단말들의 FOLLOW 집합에 나타나지 않는다. 그러므로 대입문 뒤에 빠진 세미콜론은

지나버린 다음 명령문을 시작하는 키워드가 되어버린다. 언어들의 구성을 보면 자주 계층적인 구조가 나오게 된다. 즉, 표현식들이 명령문들 안에 나타나기도 하고 이런 명령문들은 물럭 안에 나타나는 그런 식이다. 이 때 하위 구조의 동기 집합에 더 상위 구조를 시작하는 기호들을 추가할 수 있다. 예를 들어, 표현식을 생성하는 비단말들에 대한 동기 집합에 명령문을 시작하는 키워드들을 추가할 수 있을 것이다.

3. 만약 비단말 A에 대한 동기 집합에 FIRST(A)의 기호들을 추가한다면 FIRST(A)의 기호들이 입력에 나타난다는 조건하에, A에 따라서 파싱을 다시 계속 하는 것이 가능하다.
4. 만약 비단말이 공백 문자열을 생성할 수 있다면, ε를 유도하는 생성식은 기본적으로 사용할 수 있다. 이렇게 하면 몇몇 오류의 검출를 연기시킬 수 있다. 이런 방법은 오류 복구 동안에 고려되어야 하는 비단말들의 수를 줄여준다.
5. 만약 스택의 맨 위에 있는 단말의 일치될 수 없으면 간단한 해결책은 스택에서 그 단말을 꺼내고 삽입되었다는 사실을 알려준 후, 구문 분석을 계속하는 것이다. 사실상 이러한 방법은 다른 모든 토큰들로 구성된 동기 집합이 필요하다.

eg) FOLLOW와 FIRST 기호들을 동기 집합으로 사용하는 것은 연산식이 문법 1에 따라서 분석될 때는 매우 합리적이다. 이 문법에 대한 그림 5에 있는 파싱 테이블이 그림 8에 "synch"를 포함해서 다시 그려져 있다. 여기서 "synch"란 문제의 비단말에 대한 FOLLOW 집합에서 얻은 동기 토큰들을 뜻한다. 비단말에 대한 FOLLOW 집합은 예제 4. 1B에서 얻은 것이다.

그림 4. 1B의 테이블은 아래와 같이 사용된다. 만약 $M[A,a]$의 내용을 보고 그것이 비어있는 것을 발견하면 입력 기호 a는 지나친다. 만약 항목값이 synch이면, 파싱을 다시 시작하기 위해 스택의 맨 위에 있는 단말을 꺼낸다. 만약 스택의 맨 위에 있는 토큰이 입력 기호와 일치하지 않으면, 위에서 설명한 것처럼 스택에서 토큰을 꺼낸다.

비단말 기호	입력 기호					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	*synch*	synch
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ **id**	synch	synch	$F \rightarrow (E)$	synch	synch

그림 8. 그림 5에 동기 토큰을 더한 파싱 테이블

id* + id 같은 오류가 있는 입력에 대해서 그림 8의 파서와 오류 복구구조는 그림 9처럼 수행된다.

스 택	입 력	설 명
$E	)id* + id $	오류 ,)를 제거한다.
$E	id* + id $	id는 FIRST(E)에 포함된다.
$E´T	id* + id $	
$E´T´F	id* + id $	
$E´T´id	id* + id $	
$E´T´	* + id $	
$E´T´F*	* + id $	
$E´T´F	+ id $	M[F, +] = synch이므로 오류, F는 제거되었다
$E´T´	+ id $	
$E´	+ id $	
$E´T+	+ id $	
$E´T	id $	
$E´T´F	id $	
$E´T´id	id $	
$E´T´	$	
$E´	$	
$	$	

그림 9. 예측 파서에 의해서 분석되고 오류 복구되는 과정

위에서 설명한 패닉 모드 오류 복구법은 오류 메시지의 언급을 중요하게 생각하지 않는다. 일반적으로 오류에 대한 정보는 컴파일러 설계자가 제공해야 한다.

문구 수준(Phrase-level) 복구 문구 복구의 구현은 예측 파싱 테이블의 공백항에 오류 복구 루틴을 가리키는 포인터를 두는 것으로 된다. 이 루틴은 공백항에 오류 복구 루틴을 가리키는 포인터를 두는 것으로 된다. 이 루틴은 입력에 있는 기호들을 변경하여 삽입하고 빼내기도 한다. 그리고 적절한 오류 내용을 보고한다. 물론 그것들을 스택에서 꺼내진다. 문제는 스택에 있는 기호들의 교체 또는 새로운 기호를 스택에 넣는 일 등을 허락할 것인가이다.

왜냐하면, 파서에 의해서 생성된 단계들이 그 언어에 속하는 어떠한 단어의 유도와도 대응되지 않을 것이기 때문이다. 어떤 경우에는 무한 반복의 가능성이 있지 않은가를 확인해야만 한다. 복구 행동들이 결국 처리되고 있는 입력 기호들로 되는가를(또는 만약 입력의 끝에 도달했으면, 스택이 짧아졌는가를)검사하는 것은, 이러한 무한 반복에 들지 않게 하는 좋은 방법이다.

LR파싱 (right-most derivation)

앞 절에서 top-down 파싱을 공부하였는데, 이 때 파서(parser)는 입력 스트링의 기호들을 왼쪽에서 오른쪽으로 (left-to-right) 하나씩 검조(scanning)를 하였으며, 파싱은 좌단유도(left most derivation)를 이용하여 parse tree를 구성하였다. 따라서 이러한 파싱을 LL파싱이라고 하며, parsing table의 배열 $A[N, a]$가 두 개 이상의 생성을 갖지 않으면 일종의 정유한단계기계(deterministic finite state machine)가 되는 셈이 된다. 이러한 파싱을 LL(1)파싱이라고 하며 이러한 문법을 LL(1)문법이라고 한다.

이와는 반대로 LR파서란 입력 스트링의 검조를 왼쪽에서 오른쪽으로 (left-to-right)하며,①파싱 때 우단유도(right most derivation)의 역을 이용하는 것을 말한다. ②이 LR파서는 context-free 문법의 모든 프로그래밍 언어에 대하여 구성이 가능하며 파싱 방법도 가장 보편성을 띄고 있으며, top-down 파서에 비하면 실행과정에서 여러 면으로 우월함을 인식하게 될 것인데, 특히 오차의 색출에는 입력의 검조에서 쉽게 이루어질 수 있는 것이 특징이다.

LR파싱 방법은 1965년에 Knuth에 의하여 제안되었으나 당시에는 효과적이지 못했는데, DeRemer가 1971년에 좀 더 실용적인 방법을 고안하였으며 1974년에 Aho, Johnson은 이 LR방법으로 파싱을 보다 효율적으로 개발하였다.

파싱 방법이 프로그래밍 언어의 어느 부분에만 적용되는 것에 비하여 파싱 방법은 일정하게 파싱이 될 수 있는 모든 프로그래밍 언어에 적용될 수 있는 것이 특징이다.

LR파서의 구성은 아래 그림과 같이 a)파싱표와 파싱을 할 b)입력과 생성과정을 기억하는 c)스택(stack)과 이들을 운영하는 프로그램인 d) driver routine으로 구성된다. 파싱표는 구성하는 방법에 따라 그 효율 등에 약간의 차이가 있다. 가장 간단한 파싱표는 SLR(simple LR)인데 이는 다른 방법보다 약한 점이 있으며, CLR(canonical LR)은 매우 효율적인 방법이기는 하지만 실제로 작업을 하는 데 어려운 부분이 있으며, 셋째로는 LALR(lookahead LR)은 SLR과 CLR 중간의 성격를 띄고 있는데, 이는 모든 프로그래밍 언어에 효율적으로 적용되는 것이 특징이다.

파싱을 위한 스택이 기억하고 있는 양식은

$$S_0X_1S_1X_2S_2\ldots X_mS_m$$

과 같은데 각 X_i는 문법의 기호를 나타내며, S_i는 각 단계(state)를 나타내는 부호이다. 이 때 S_m이 스택의 제일 top에 오게 된다.

파싱표는 Action부분과 Go To부분으로 되어 있는데, 스택과 입력의 구성이 처음에는 $(S_0, a_1a_2\ldots a_n\$)$으로 시작하여 보통 아래와 같은 모양을 갖게 된다.

$$(S_0X_1S_1X_2S_2\ldots X_mS_m, a_ia_{i+1}\ldots a_n\$)$$

스택의 제일 top의 부호 S_m과 현재 입력 a_i에 의하여 파싱표의 Action[S_m, a_i]은 다음 네 가지 일을 하게 된다.

(1) 만일 Action[S_m , a_i] = shift S 이면 파서를 shift를 실행하며 스택과 입력은

$$(S_0X_1S_1X_2S_2\ldots X_mS_ma_iS,\ a_{i+1}\ldots a_n\$)$$

와 같이 변하며 a_iS가 스택에 shift되며 a_{i+1}이 새로운 현재 입력이 된다.

(2) 만일 Action[S_m , a_i] = reduce $A \rightarrow \beta$이면 파서는 reduce를 실행하고 스택과 입력은

$$(S_0X_1S_1X_2S_2 \ldots X_{m-r}S_{m-r}AS,\ a_ia_{i+1} \ldots a_n\$)$$

와 같이 변하며, 여기서 S = Go To[S_{m-r} A]이며 r = $|\beta|$ 이다.

(3) 만일 Action[S_m , a_i] = accept 이면 파싱은 완료되고,

(4) 만일 Action[S_m , a_i] = error 이면 파서는 착오를 발견한 것이 되고, 이 때 착오교정 루우틴을 부르게 된다.

다음 문법 G에 대하여

G : (1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow$ id

parsing table은 아래와 같다.

단계	Action						Go To		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

이 때 입력이 id + id * id$라면 스택과 입력은 처음에는 다음과 같이 구성된다.

	스 택	입 력	출 력
(1)	0	id + id * id	

위의 스택과 입력에서 S_m= 0, a_i = id 이며 파싱표의 Action[0, id] = s5이다. 이 때 규칙 (1)에 의한 s5는 id를 shift하고 S = 5가 되어 스택과 입력은

	스 택	입 력	출 력	
(2)	0 id 5	+ id * id$	$F \rightarrow$ id	r = $\|id\|$ = 1

와 같이 이루어진다. (2)에서는 S_m = 5, a_i = +이므로 Action[5, +] = r6 인데 r6는 (6)의 감축을 나타내므로 문법(6)의 감축은 $F \rightarrow$ id이며 S = Go To[0, F] 스택과 입력은

	스 택	입 력	출 력
(3)	0 F 3	+ id * id$	T → F n = |F| = 1

와 같이 이루어지며 (3)에서 S_m= 3, a_i= + 이므로 Action[3, +] = r4 이므로 문법에 의하여 (4)의 생성은 T→F 에 의하여 S = Go To[0, T] = 2 이므로

	스 택	입 력	출 력
(4)	0 T 2	+ id * id$	E → T

로 된다. (4)에서 S_m= 2, a_i= + 이므로 Action[2, +] = r2이므로 문법5-4-1에 의하여 E → T의 감축이 생기며 S = Go to[0, E] = 1 이므로 스택과 입력은

	스 택	입 력
(5)	0 E 1	+ id * id $

로 되며, (5)에서 S_m= 1, a_i = + 이므로 Action[1, +] = s6이며, 규칙(1)에 의하여 스택과 입력은

	스 택	입 력
(6)	0 E 1 + 6	id * id $

과 같이 된다. (6)에서 S_m= 6, a_i = id이므로 Action[6, id] = s5이며

(7)	0 E 1+6 id 5	* id $	F → id

가 되어, Action[5, *] = r6 이므로 F → id의 감축과 S = Go To[6, F] = 3 이므로

(8)	0 E 1+6 F 3	* id $	T → F

가 되며, Action[3, *] = r4 이므로 T → F의 감축이 생기며, S = Go To[6, T] = 9 이므로

	스 택	입 력	출 력
(9)	0 E 1+6 T 9	* id $	

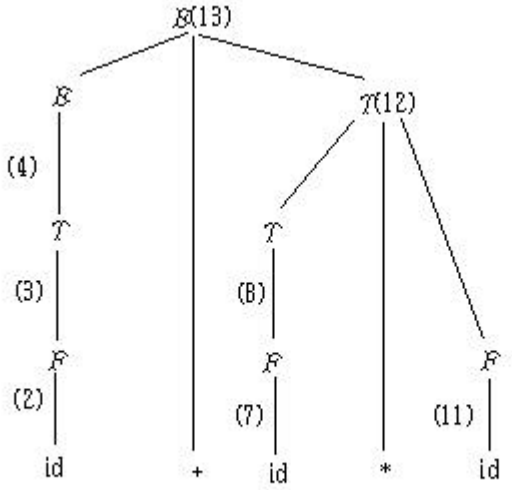
이 되며 Action[9, *] = s7 이므로

(10)	0 E 1+6 T 9 *7	id$	

이하 계속하면

(11)	0 E 1 + 6 T 9 * 7 id 5	$	F → id
(12)	0 E 1 + 6 T 9 * 7 F 10	$	T → T * F n = |T * F| = 3
(13)	0 E 1 + 6 T 9	$	E → E + T n = |E + T| = 3
(14)	0 E 1	$	
(15)	accept		

위의 파싱의 출력 결과를 parse tree로 나타내면 아래와 같다.



id + id * id 의 parse tree

(p.124)

SLR 파싱표

만일 어떤 문법이 파싱표를 가질 수 있다면 이 문법으로 DFA를 구성할 수 있으며 이 DFA를 SLR 파싱표로 변환이 가능하다.

LR(0) 항목이란 어떤 문법 G의 생성에서 도트(dot)가 생성의 오른쪽 어느 곳이나 오는 것을 말하는데, 예를 들어 생성 $X \rightarrow ABC$가 있다면 이 생성은 다음과 같은 네 개의 항목을 생산할 수 있다.

$$\text{LR(0) item} : \quad \begin{array}{l} X \rightarrow \cdot ABC \\ X \rightarrow A \cdot BC \\ X \rightarrow AB \cdot C \\ X \rightarrow ABC \cdot \end{array}$$

위의 첫째 항목인 $X \rightarrow \cdot ABC$가 갖는 뜻은 주어진 입력에 대하여 ABC로 유도될 수 있는 스트링을 기대함을 말하며, 둘째 항목인 $X \rightarrow A \cdot BC$ 는 입력에 의하여 A에서 유도될 수 있는 스트링을 확인하였고, 이어서 BC 로부터 유도될 수 있는 스트링을 기대하고 있음을 말한다. 예를 들어 다음 문법 G에서

$$\begin{array}{l} G : S' \rightarrow Sc \\ \quad S \rightarrow SA \mid A \\ \quad A \rightarrow aSb \mid ab \end{array} \qquad \text{(문법1)}$$

에서 항목 $S' \rightarrow \cdot Sc$ 라는 첫 번째 항목을 얻을 수 있으며, 이 항목은 S로부터 유도될 수 있는 스트링을 기대하며, 입력 S에 의하여 새로운 항목 $S' \rightarrow S \cdot c$를 생산하게 되는데, 이 항목의 뜻은 입력 S를 확인하고 다음 입력에서 c를 기대하고 있다고 볼 수 있다.

항목 $S' \rightarrow \cdot Sc$가 Sc로부터 유도될 수 있는 스트링을 뜻하므로 이 때 유도될 수 있는 범위를 모두 묶어서 CLOSURE 라고 한다.

함수 CLOSURE(I)를 만드는 방법은 다음과 같다.

(1) 만일 $A \rightarrow \alpha \cdot B\beta$의 항목이 CLOSURE(I)에 속하고 $B \rightarrow \gamma$ 의 생성이 존재하면 $B \rightarrow \cdot \gamma$ 는 CLOSURE(I)에 속한다.

(2) 그 이유는 실제로 항목 $A \rightarrow \alpha \cdot B\beta$가 CLOSURE(I)에 속하면서 $B\beta$가 유도할 수 있는 입력 스트링을 기대하고 있기 때문에 $B \rightarrow \gamma$의 생성이 존재한다면 항목 $B \rightarrow \cdot \gamma$는 당연히 같은 입력 스트링을 기대하게 되므로 항목 $B \rightarrow \cdot \gamma$는 CLOSURE(I)에 속해야 할 것이다. 이러한 상태의 전이를 ε-전이라 한다. 예를 들어 문법 G에서

$$\begin{array}{l} G : S' \rightarrow Sc \\ \quad S \rightarrow SA \mid A \\ \quad A \rightarrow aSb \mid ab \end{array}$$

항목 $S' \rightarrow \cdot Sc$ 가 CLOSURE(I)에 속하면 다음 항목들도 CLOSURE(I)에 속하게 된다.

$$S' \rightarrow \cdot Sc$$
$$S \rightarrow \cdot SA$$
$$S \rightarrow \cdot A$$
$$A \rightarrow \cdot \alpha S b$$
$$A \rightarrow \cdot \alpha b$$

항목 $S' \rightarrow \cdot Sc$를 보면 도트(dot)의 바로 오른쪽에 S가 $S \rightarrow \cdot SA$, $S \rightarrow \cdot A$ 이들은 모두 CLOSURE(I)에 속하게 되며, 항목 $S \rightarrow \cdot A$에서 도트 바로 오른쪽에 A가 오므로 $A \rightarrow \cdot \alpha S b$, $A \rightarrow \cdot \alpha b$ 는 모두 CLOSURE(I)에 속하게 된다. 또한 도트 바로 오른쪽에 α가 오더라도 α가 터미널이므로 더 이상 생성이 없어 CLOSURE(I)에 속하는 항목이 생기지 않음을 알 수가 있다.

다음 유용한 함수로는 GO TO가 있는데 GO TO(I, X)라면 I는 CLOSURE(I)의 항목의 집합을 뜻하며 X는 문법기호를 나타내고 있는데, GO TO(I, X)라면 또 다른 항목의 집합을 나타낸다. 만일 $A \rightarrow \alpha X \cdot \beta$가 속하는 항목의 CLOSURE를 말하고 있다.

위의 CLOSURE(I)에서 GO TO(I, S)는 다음 항목들을 나타낸다.

$$S' \rightarrow S \cdot c$$
$$S \rightarrow S \cdot A$$
$$A \rightarrow \cdot \alpha S b$$
$$A \rightarrow \cdot \alpha b$$

즉, S라는 입력스트링에 의하여 $S' \rightarrow \cdot Sc$ 는 $S' \rightarrow S \cdot c$, $S \rightarrow \cdot SA$는 $S \rightarrow S \cdot A$를 얻게 되는데, 도트(dot) 다음에 c가 터미널이므로 c에 생성이 없다. 그러나 항목 $S \rightarrow S \cdot A$는 $A \rightarrow \cdot \alpha S b$, $A \rightarrow \cdot \alpha b$의 ε-전이 항목이 있으며, 도트 다음에 a가 오지만 더 이상 터미널에는 생성이 없으므로 항목도 없게 된다.

이 때 GO TO(I, S)는 그 단계에서 S-전이를 이룬다고 한다.

이들 항목들은 함수 CLOSURE(I)와 GO TO(I, X)를 이용하면 그림 1과 같은 유한자동을 구성할 수가 있다.

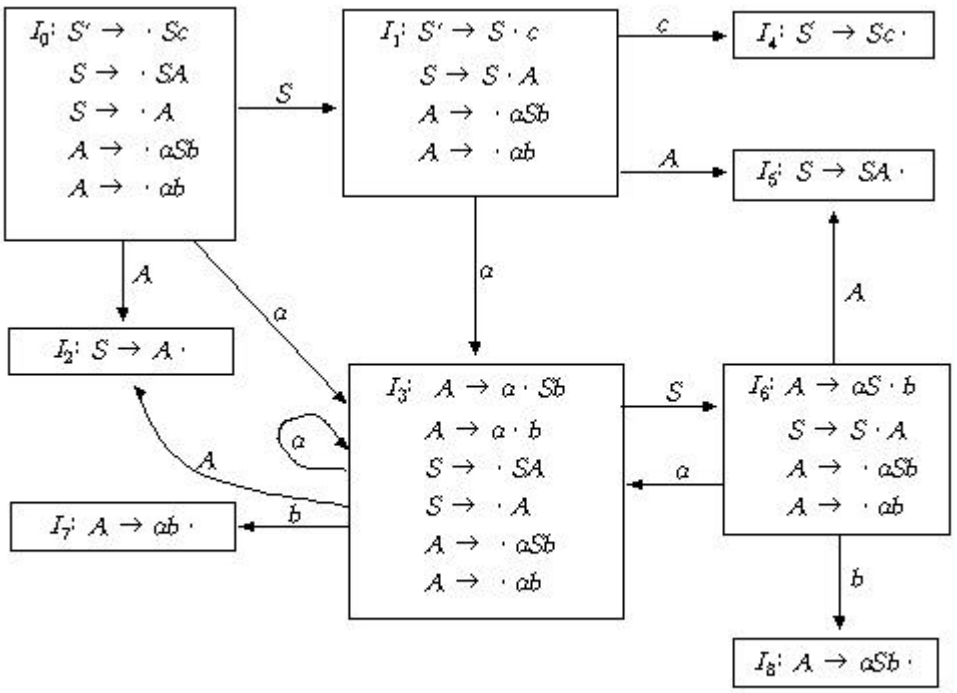


그림 1.항목집합으로 구성된 FA

I_0 : $S' \rightarrow \cdot Sc$	I_1 = GO TO(I_0 S) :
$S \rightarrow \cdot SA$	$S' \rightarrow S \cdot c$
$S \rightarrow \cdot A$	$S \rightarrow S \cdot A$
$A \rightarrow \cdot aSb$	$A \rightarrow \cdot aSb$
$A \rightarrow \cdot ab$	$A \rightarrow \cdot ab$
I_2 = GO TO(I_0 A) :	I_3 = GO TO(I_0 a)
$S \rightarrow A \cdot$	$A \rightarrow a \cdot Sb$
	$A \rightarrow a \cdot b$
I_4 = GO TO(I_1, c)	$S \rightarrow \cdot SA$
	$S \rightarrow \cdot A$
$S' \rightarrow Sc \cdot$	$A \rightarrow \cdot aSb$
	$A \rightarrow \cdot ab$
	I_3 = GO TO(I_3 a)
I_5 = GO TO(I_1, A)	
	I_6 = GO TO(I_3, S)
$S' \rightarrow SA \cdot$	
	$A \rightarrow aS \cdot b$
I_7 = GO TO(I_3 b)	$S \rightarrow S \cdot A$
	$A \rightarrow \cdot aSb$
$A \rightarrow ab \cdot$	$A \rightarrow \cdot ab$
I_8 = GO TO(I_6 b)	
$A \rightarrow aSb \cdot$	

< 항목의 집합 >

다음 문법을 유한자동으로 변환하여 파싱표를 구성해 보자. 문법 G는
ex) ①

G : (0) $E' \rightarrow E$
(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$ (문법 2)
(6) $F \rightarrow$ id

우선 파싱표를 만들기 위하여 FIRST 와 FOLLOW를 구하도록 하자. FIRST를 구하는 방법은 앞 절의 방법을 그대로 활용한다.

규칙 (1)에 의하여 모든 터미널은 그 자체가 FIRST가 된다.

규칙 (2)에 의하여 $\alpha \rightarrow \alpha\beta$의 모양은 F → (E), F → id 이므로 FIRST(F)는 {(, id}를 갖는다.

규칙 (3)에 의하여 $\alpha \rightarrow Y_1 Y_2 \cdots Y_K$에서 i = 1 이면 FIRST(Y_1)은 FIRST(α)에 속하므로 FIRST(F) ⊂ FIRST(T), FIRST(T) ⊂ FIRST(E) FIRST(E) ⊂ FIRST(E')가 되므로

$$FIRST(E') \supseteq FIRST(E) \supseteq FIRST(T) \supseteq FIRST(F) = \{(,\ id\}$$

를 얻게 된다.

FOLLOW를 얻기 위하여 앞 절의 방법을 이용하면

규칙 (1)에 의하여 FOLLOW(E') = {$}

규칙 (2)에 의하여 $A \rightarrow \alpha B\beta$ 에 해당하는 생성 $F \rightarrow (E)$에서 FOLLOW(E) = {)}, 생성 $T \rightarrow T * F$에서 FOLLOW(T) = FIRST(*F) = {*}, 생성 $E \rightarrow E + T$에서 FOLLOW(E) = FIRST(+ T) = {+}를 얻게 된다.

규칙 (3)에 의하여 $A \rightarrow \alpha B$에 해당하는 생성은 $E' \rightarrow E$에서 FOLLOW(E') ⊆ FOLLOW(E), 생성 $E \rightarrow T$에서 FOLLOW(E) ⊆ FOLLOW(T), 생성 $T \rightarrow F$에서 FOLLOW(T) ⊆ FOLLOW (F)가 성립되므로 다음과 같이 정리될 수 있다.

FOLLOW(E') = {$}
FOLLOW(E) = {$, +,)}
FOLLOW(T) = {$, +,), *}
FOLLOW(F) = {$, +,), *}

또한 문법 2로 항목들의 집합을 구성해 보면 다음과 같다.

ex) ②

I_0 : $E' \rightarrow \cdot E$	I_1 = GO TO(I_0 E)
$E \rightarrow \cdot E + T$	$E' \rightarrow E \cdot$ accept
$E \rightarrow \cdot T$	$E \rightarrow E \cdot + T$
$T \rightarrow \cdot T * F$	
$T \rightarrow \cdot F$	I_2 = GO TO(I_0 T)
$F \rightarrow \cdot (E)$	$E \rightarrow T \cdot$
$F \rightarrow \cdot$ id	$T \rightarrow T \cdot * F$
I_3 = GO TO(I_0 F)	I_5 = GO TO(I_0 id)
$T \rightarrow F \cdot$	$F \rightarrow$ id $\cdot$
I_4 = GO TO(I_0 ( )	I_6 = GO TO(I_1, +)
$F \rightarrow (\cdot E)$	$E \rightarrow E + \cdot T$
$E \rightarrow \cdot E + T$	$T \rightarrow \cdot T * F$
$E \rightarrow \cdot T$	$T \rightarrow \cdot F$
$T \rightarrow \cdot T * F$	$F \rightarrow \cdot (E)$
$T \rightarrow \cdot F$	$F \rightarrow \cdot$ id
$F \rightarrow \cdot (E)$	
$F \rightarrow \cdot$ id	I_9 = GO TO(I_6 T)
	$E \rightarrow E + T \cdot$
I_7 = GO TO(I_2 *)	$T \rightarrow T \cdot * F$
$T \rightarrow T * \cdot F$	
$F \rightarrow \cdot (E)$	I_{10} = GO TO(I_7, F)
$F \rightarrow \cdot$ id	$T \rightarrow T * F \cdot$
I_8 = GO TO(I_4 E)	I_{11}= GO TO(I_8))
$F \rightarrow (E \cdot)$	$F \rightarrow (E) \cdot$
$E \rightarrow E \cdot + T$	

< 문법 2의 항목 집합 >

위의 항목집합으로 다음과 같이 정유한 자동을 구성할 수 있다.
이 유한자동으로 파싱표를 만들기 위하여는 다음의 규칙을 이용한다.
ex) ⑤

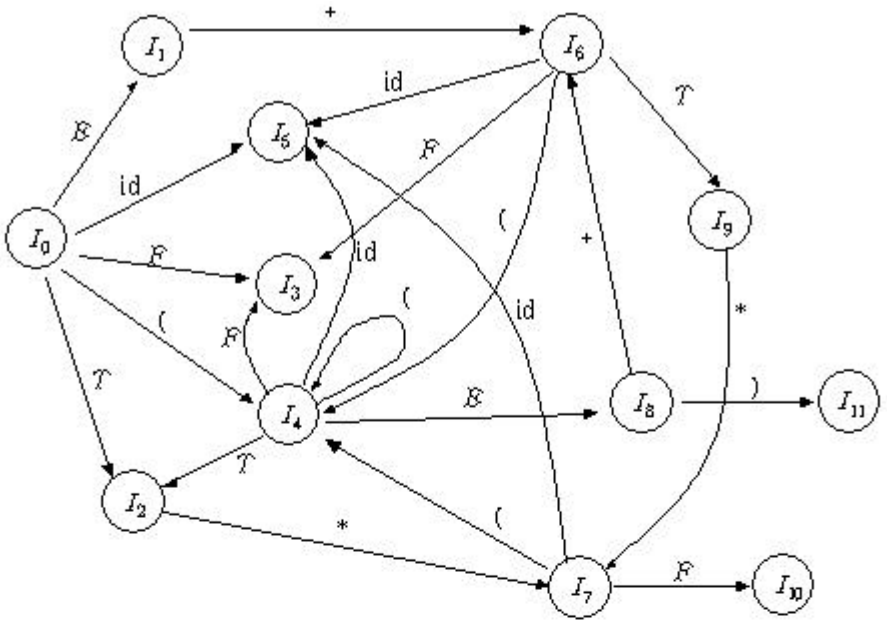


그림 2. 문법 2의 FA 구성

(1) 항목의 집합명칭을 I_i 라 하면 0,1,2, ... ,n 의 I는 I_i 단계의 첨자이며

(2) 만일 $A \rightarrow \alpha \cdot a\beta$ 가 I_i 에 있고 a 가 터미널이며 GO TO(I_3 I_i, a) = I_j 라면 파싱표의 Action[i, a]를 Shift j로 놓는다.

(3) 만일 $A \rightarrow \alpha \cdot$ 가 I_i 에 있으면 FOLLOW(A)에 있는 모든 a 에 대하여 Action[I_3, a]를 reduce $A \rightarrow \alpha$ 로 놓는다.

(4) 만일 $S' \rightarrow S \cdot$ 가 I_i 단계에 있으면 Action[I_3, $]를 accept로 놓는다.

(5) 만일 GO TO(I_i, A) = I_j 이면 파싱표의 GO TO[I_3, A] = j 가 된다. 이렇게 구성한 파싱표를 SLR 파싱표라 하며 SLR 파싱표를 갖는 문법을 SLR(1)이라고 한다. 만일 파싱표의 어느 난에나 두 개 이상의 동작이 들어 있으면 이 파싱표는 SLR(1) 이라고 볼 수 없다. 따라서 파서는 작동이 성공적으로 끝나지 않음을 알아야 한다.

(6) 그 외의 빈칸은 모두 착오(error)에 해당됨을 알 수 있다.

위의 FA를 이용하여 SLR 파싱표를 구성해 보자.

규칙 (2)에 의하여 i = 0이면, I_0 단계에서 a = {id, (}이므로

Action[0, id] = $s5$, Action[0, (] = $s4$

이며 i = 1이면 a = {+} 이므로

Action[1, +] = $s6$

임을 알 수 있다.

i = 2 이면 a = {*} 이므로

Action[2, *] = $s7$

이며, i=3 에서는 α 가 없으며,

i=4에서는 a = {(, id}이므로

Action[4, (] = $s4$, Action[4, id] = $s5$

이며, i=5에서는 a는 없으며,

i=6에서는 a = {id, (} 이므로

Action[6, id] = $s5$, Action[6, (] = $s4$

이며, i=7에서는 a = {(, id} 이므로

Action[7, (] = $s4$, Action[7, id] = $s5$

이며, i=8에서는 a = {+,)} 이므로

Action[8, +] = $s6$, Action[8,)] = $s11$

이며, i=9, a = {*} 이므로

Action[9, *] = $s7$

이 된다. i=10에서 a는 없으며, i=11에서도 a는 없다.

규칙 (3)에 의하여 항목의 집합에서 $A \rightarrow \alpha \cdot$ 의 모양을 찾아보면 I_0에는 없으며, I_1에서는

(0) $E' \rightarrow E \cdot$

가 있는데, FOLLOW(E') = {$} 이므로

Action[1, $] = $r0$, 즉 accept

가 될 것이다.

I_2 에서는 (2) $E \rightarrow T \cdot$ 가 있는데 FOLLOW(E) = {$, +,)} 이므로

Action[2, $] = $r2$, Action[2, +] = $r2$, Acrion[2,)] = $r2$

가 된다.

I_3 에서는 (4) $T \rightarrow F \cdot$ 가 있는데 FOLLOW(T) = {$, +,), *} 이므로

Action[3, $] = $r4$, Action[3, +] = $r4$

Acrion[3,)] = $r4$, Action[3, *] = $r4$

가 된다. I_4에서는 없으며, I_5에서는 (6) $F \rightarrow$ id $\cdot$ 가 있는데, FOLLOW(F) = {$, +,), *} 이므로

Action[5, *] = $r6$, Action[5, +] = $r6$

Acrion[5,)] = $r6$, Action[5, *] = $r6$

가 된다. I_6, I_7, I_8 에서는 없으며, I_9에서는 (1) $E \rightarrow E + T \cdot$ 가 있어 FOLLOW(E) = {$, +,)}이므로

Action[9, $] = $r1$, Action[9, +] = $r1$, Acrion[9,)] = $r1$

이 된다. I_{10} 에서는 (3) T → T * F · 가 있기 때문에 FOLLOW(T) = {$, +,), *} 이므로

Action[10, $] = $r3$, Action[10, +] = $r3$,

Action[10,)] = $r3$, Action[10, *] = $r3$,

가 된다. I_{11}에서는 (5) $F \rightarrow (E) \cdot$ 가 있어 FOLLOW(F) = {$, +,), *} 이므로

Action[11, $] = $r5$, Action[11, +] = $r5$,

Action[11,)] = $r5$, Action[11, *] = $r5$,

가 된다.

규칙 (4)에 의하여 $E' \rightarrow E \cdot$ 가 있어

Action[1, $] = accept

가 된다.

규칙 (5)에 의하여 유한자동의 I_0 에서 $A = \{E, F, T\}$ 이므로

GO TO[0, E] = 1, GO TO[0, T] = 2, GO TO[0, F] = 3

이 된다. I_1, I_2, I_3 에서는 A가 없으며, I_4에서는 $A = \{E, F, T\}$이므로

GO TO[4, E] = 8, GO TO[4, T] = 2, GO TO[4, F] = 3

이 된다. I_5에는 A가 없으며, I_6에서는 $A = \{F, T\}$ 이므로

GO TO[6, T] = 9, GO TO[6, F] = 3

이 된다. I_7에서는 $A = \{F\}$ 이므로

GO TO[7, F] = 10

이 된다.

I_8, I_9, I_{10}, I_{11} 에서는 A 가 없다. 따라서 해당하는 SLR 파싱표는 아래와 같다.

단계	Action						Go To		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

< SLR 파싱표 >

④ CLR 파싱표

SLR의 경우 생성을 실행하는데 더 많은 정보가 없이는 생성의 선택에 착오가 생기는 경우가 발생하므로 CLR(canonical LR)에서는 터미널을 추가하여 각 항목을 정의하는데, 그 일반적인 항목의 양식은

$$[A \rightarrow \alpha \cdot \beta, \{a\}]$$

여기서 $A \rightarrow \alpha\beta$ 는 생성을 나타내며 α나 β는 센텐샬 폼이며, a는 터미널이나 $$를 나타내게 되는데 lookahead라고 하며, 이러한 것들은 LR(1) 항목이라고 부른다. 1은 항목의 두 번째 부분인 lookahead의 길이가 1임을 나타내고 있다.

LR(1)의 항목을 구성하는 방법은 LR(0) 항목를 구성하는 방법과 유사하다. 본 절에서는 전이를 위한 NFA를 구성한 후 DFA를 구성하여 항목들을 만든다. 전이를 위한 NFA를 구성하는 방

법은 다음과 같다.

(1) X-전이에 대하여

$[A \rightarrow \alpha \cdot X\beta, \{a_1, a_2, \ldots, a_n\}]$ 은

$[A \rightarrow \alpha X \cdot \beta, \{a_1, a_2, \ldots, a_n\}]$이 되며, 여기서 α, β는 sentential form이며, A, X는 nonterminal이다.

(2) ε-전이에 대하여

$[A \rightarrow \alpha \cdot B\beta, \{a_1, a_2, \ldots, a_n\}]$ 은

$[B \rightarrow \cdot r, T]$, 여기서 $B \rightarrow r$ 는 생성이며 T는 다음 경우에 b로 구성된다.

① $\beta \neq \varepsilon$ 면 b = FIRST(β)

② $\beta \Rightarrow \varepsilon$ 일 때 $b = [a_i]$ $(1 \leq i \leq n)$

(3) 생성 S → α 에 대하여는 초기단계에서 S → · α, {$} 이 존재한다.

예를 들어 다음 문법 G에서

$$G: S \rightarrow A$$

$$A \rightarrow BA \mid \varepsilon \quad \text{(문법 3-1)}$$

$$B \rightarrow aB \mid b$$

생성 $S \rightarrow A$는 항목 $S \rightarrow \cdot A$를 제공하며, 항목 $S \rightarrow \cdot A$는 규칙 (3)에 의하여 {$}을 lookahead로 얻는다. 따라서

$[S \rightarrow \cdot A, \$]$

의 CLR 항목을 얻게 된다. $[A \rightarrow \cdot, T]$

항목 [$S \rightarrow \cdot A, \$$]은 문법에 의하여 $[A \rightarrow \cdot BA, T]$와 의 ε-전이를 갖게 되는데, 먼저 A → · BA, T를 살펴보면 규칙 (2)에서 $A \rightarrow \alpha \cdot B\beta, \{a_1, a_2, \ldots, a_n\}$과 $B \rightarrow \cdot r, T$와 연관하여 생각해 보면 $A \rightarrow \alpha \cdot B\beta$와 상응하는 $S \rightarrow \cdot A$와 비교하면 $\beta = \varepsilon$ 임을 알 수 있다. 따라서 규칙 (2)-②에 의하여 T는 {$}임을 알 수 있다. 따라서

[$A \rightarrow \cdot BA, \$$] [$A \rightarrow \cdot, \$$]

를 얻을 수 있다.

또한 항목 [$A \rightarrow \cdot BA, \$$]은 $[B \rightarrow \cdot aB, U]$와 $[B \rightarrow \cdot b, U]$의 ε-전이를 생산한다. 여기서 lookahead U를 구하기 위하여 항목 [$A \rightarrow \cdot BA, \$$]와 [$A \rightarrow \alpha \cdot B\beta, \{a_1, a_2, \ldots, a_n\}$]를 다시 한 번 비교해 보면 $\beta = A$가 됨을 알 수 있다. 따라서 규칙 (2)-①에 의하여 β가 시작되는 터미널들 FIRST(A), 즉 $B \rightarrow aB$에서 {a}와 $B \rightarrow \cdot b$에서 {b}를 U에 포함시킬 수 있다.

따라서 {a, b} ⊂ U를 얻을 수 있으며 β=A에서 또한 생성 [A→ε]은 ε을 유도하므로 [$A \rightarrow \cdot BA, \$$]의 $를 U는 받게 된다. 따라서 U = {a, b, $}를 얻게 된다.

이를 이용하여 NFA를 구성하여 보면

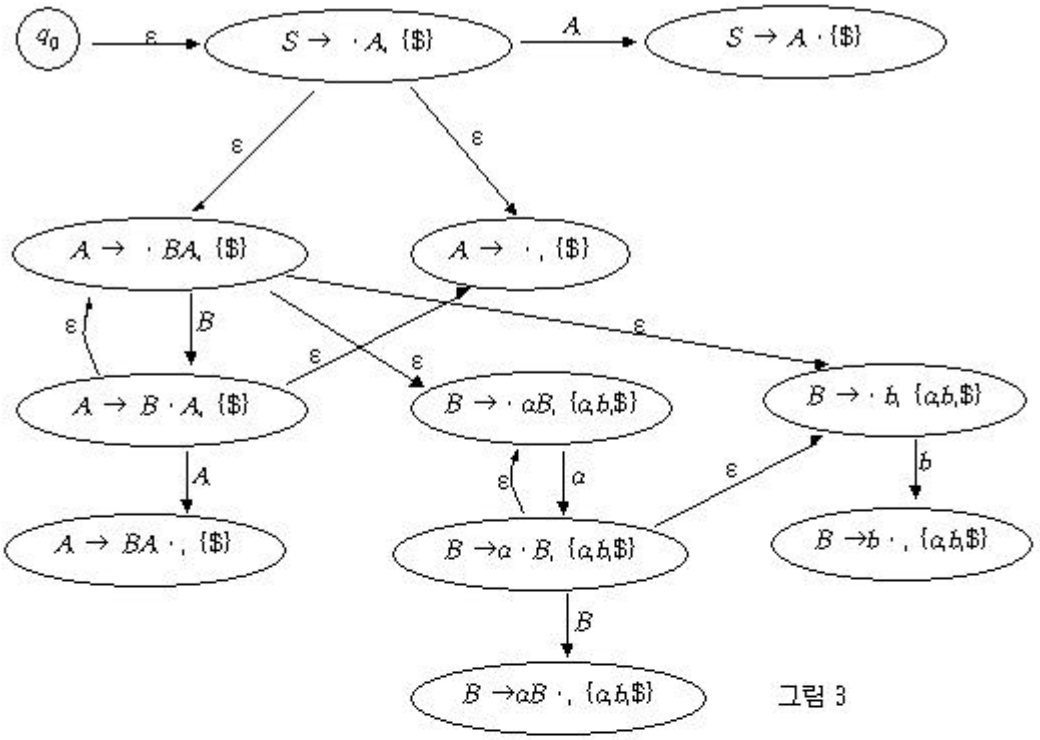


그림 3

이들의 DFA를 그려 보면

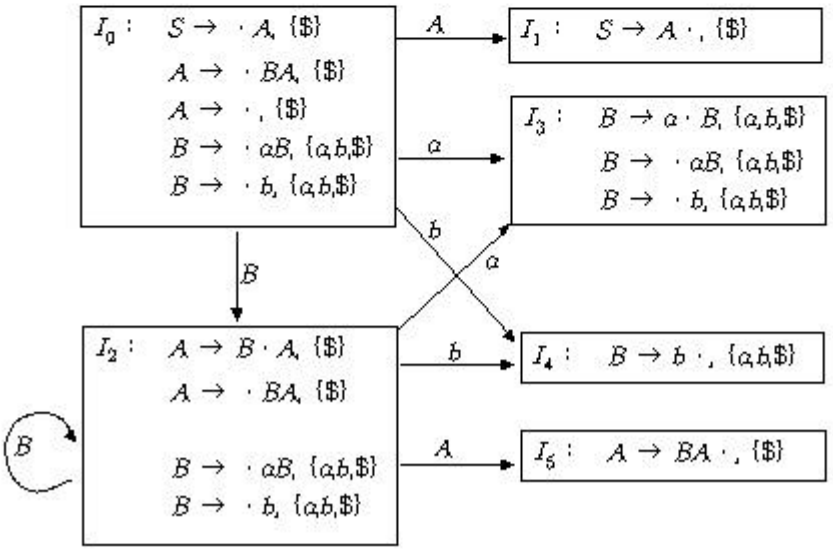


그림 4

다음 문법 G를 이용하여 파싱표를 구성하도록 하자.

G : $S' \rightarrow S$ −⓪
$S \rightarrow CC$ −①
$C \rightarrow aC \mid b$
② ③

(문법 3-2)

위의 문법에서 LR(1)의 항목을 구하여 NFA를 구성해 보면 규칙 (3)에 의하여 생성 $S' \rightarrow S$ 는 LR(1) 항목

$[S' \rightarrow \cdot S\ ,\ \$]$

을 얻게 되며, 항목 $[S' \rightarrow \cdot S\ ,\ \$]$은 ε-전이 $[S \rightarrow \cdot CC\ ,\ T]$를 얻게 되는 데, T는 규칙 (2)에서 $[A \rightarrow \alpha \cdot B\beta,\ \{a_1, a_2, \ldots, a_n\}]$의 양식을 $[S' \rightarrow \cdot S\ ,\ \$]$과 비교하여 얻을 수 있다. 즉, $\beta = \varepsilon$ 이므로 규칙 (2)-②에 의하여 T = {$}임을 알 수 있다. 즉,

$[S \rightarrow \cdot CC\ ,\ \$]$

의 LR(1) 항목를 얻으며, 항목 $[S \rightarrow \cdot CC\ ,\ \$]$은 ε-전이 $[C \rightarrow \cdot aC,\ T]$와 $[C \rightarrow \cdot b,\ T]$를 얻는데, γ를 계산하면 양식 $[A \rightarrow \alpha \cdot B\beta,\ \{a_1, a_2, \ldots, a_n\}]$에서 $\beta = C$ 이므로 규칙 (2)-①에 의하여 FIRST(β) = FIRST(C) = {a, b} 임을 알 수 있다. 즉,

$[C \rightarrow \cdot aC,\ a/b]$, $[C \rightarrow \cdot b,\ a/b]$

를 얻게 된다. LR(1) 항목 $[C \rightarrow \cdot aC,\ a/b]$ 는 a-전이를 하여 규칙(1)에 의하여

$[C \rightarrow a \cdot C,\ a/b]$

를 얻으며, $[C \rightarrow \cdot b,\ a/b]$는 b-전이를 하여 규칙 (1)에 의하여

$[C \rightarrow b \cdot\ ,\ a/b]$

를 얻게 된다. 그러나 LR(1) 항목 $[C \rightarrow a \cdot C,\ a/b]$는 ε-전이를 하여 $[C \rightarrow \cdot aC,\ a/b]$와 $[C \rightarrow \cdot b,\ T]$를 얻게 되는데, γ를 계산하면 $[A \rightarrow \alpha \cdot B\beta,\ \{a_1, a_2, \ldots, a_n\}]$의 양식을 $[C \rightarrow a \cdot C,\ a/b]$와 비교하면 $\beta = \varepsilon$ 임을 알 수 있어 규칙 (2)-②에 의하여 T = {a/b} 임을 알 수 있다. 즉,

$[C \rightarrow \cdot aC,\ a/b]$, $[C \rightarrow \cdot b,\ a/b]$

를 구할 수 있다. 이와 같이 반복하면 아래의 NFA를 얻을 수 있다.

①

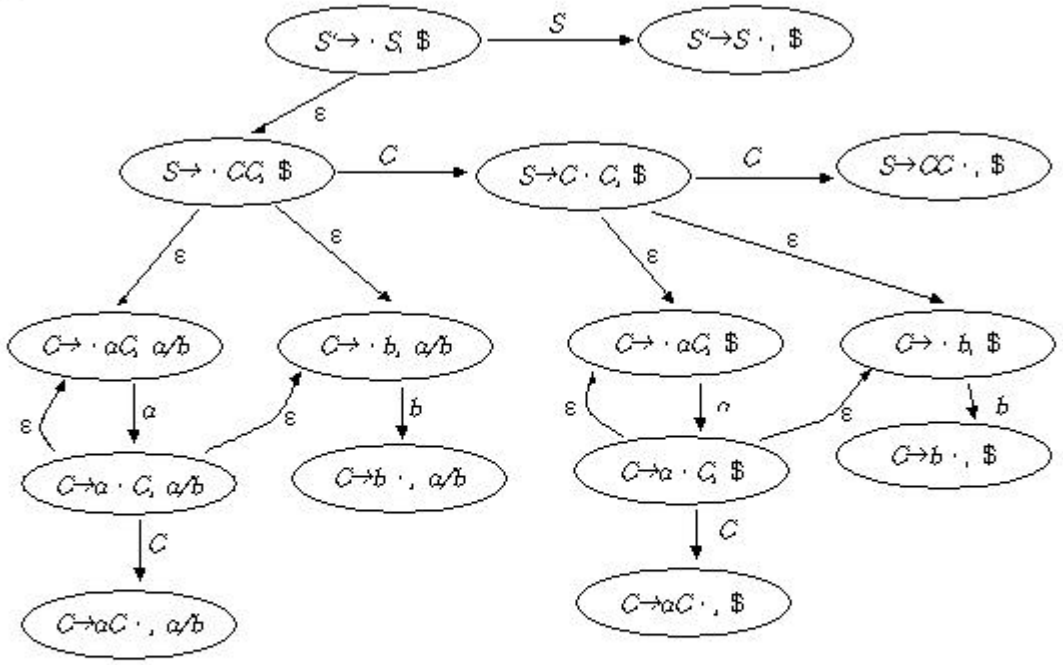
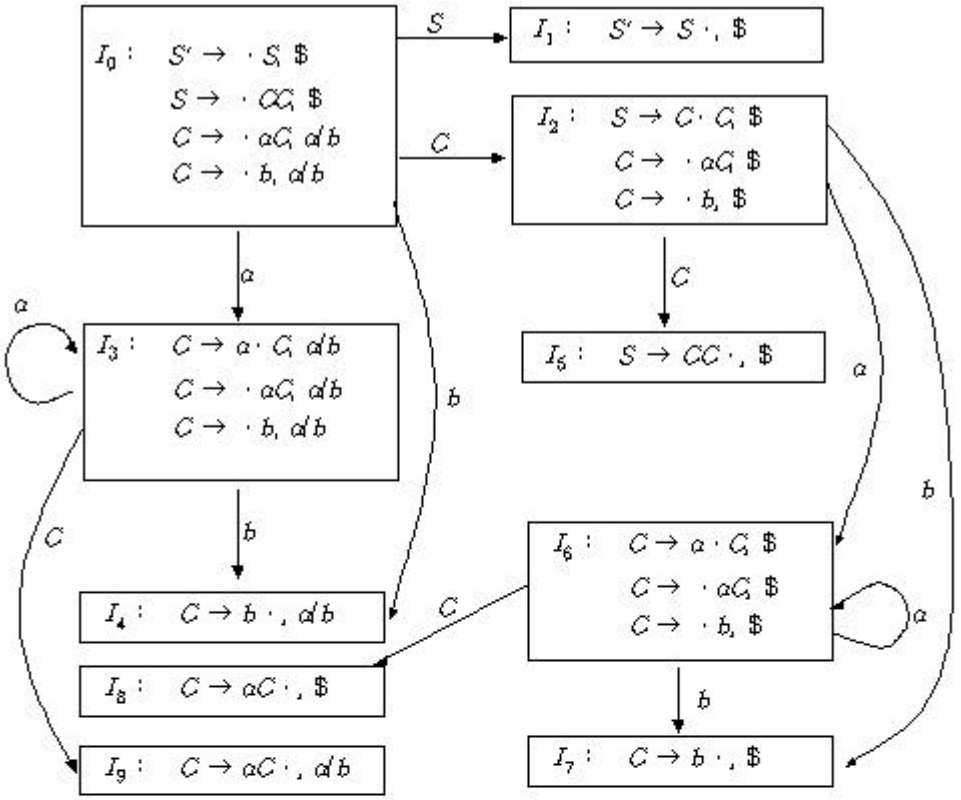


그림 5

따라서 항목들의 집합으로 DFA를 구성해보면 아래와 같이 된다.

ex) ②



(문법 3-2의 DFA)

CLR 파싱표를 구성하는 규칙은 SLR 파싱표를 구성하는 방식과 유사하다.

(1) 항목의 집합명칭을 I_i 라면 0,1, ..., n 의 i 는 I_i 단계의 첨자라 하자.

(2) 만일 항목 $[A \rightarrow \alpha \cdot a\beta\ ,\ b]$가 I_i에 속하며 GO TO(I_i, a) = I_j 라 하면 파싱표의 Action[i, a] 는 shift j 라 놓게 되며,

(3) 만일 $[A \rightarrow \alpha \cdot ,a_i]$ 가 I_i에 속하면 Action[I_i, a_i]는 reduce A $\rightarrow \alpha$ 로 놓게 된다.

(4) 만일 $[S' \rightarrow S \cdot ,\$]$이 I_i 에 속하면 Action[I_i, $] 은 accept 로 한다.

(5) 만일 GO TO(I_i, A) = I_j 라면 파싱표의 Go To [I_i, A]=j로 한다.

(6) 그 외에 공란은 error로 한다.

문법 G의 CLR 파싱표를 구성하기 위하여 먼저 규칙 (2)에 의하여 DFA의 I_0에서 다른 단계로 나갈 때 입력이 터미널인 a_i ∈ {a는 I_3로, b는 I_4로} 이므로

Action[0, a] = $s3$, Action[0, b] = $s4$

를 얻게 되며, I_1에서는 a 가 없으며,

I_2에서는 a_i ∈ {a는 I_6로, b는 I_7로} 이므로

Action[2, a] = s6, Action[2, b] = s7을 얻으며,

I_3에서는 a_f = {a는 I_3로, b는 I_4} 이므로

Action[3, a] = s3, Action[3, b] = s4를 얻게된다.

I_4, I_5 에서는 나가는 단계가 없으며,

I_6에서는 a_f = {a는 I_6로, b는 I_7으로} 이므로

Action[6, a] = s6, Action[6, b] = s7

임을 알 수 있다.

I_7, I_8, I_9 은 나가는 단계가 없다.

규칙 (3)에 의하여 [$A \rightarrow \alpha\cdot$, a_f] 의 모양을 한 단계는 I_4, I_5, I_7, I_8, I_9 으로서 I_4 에서 $C \rightarrow b\cdot$, a/b 는 a_f={a, b} 이며, $C \rightarrow b$는 식 (3) 이므로 r3를 얻게 되어

Action[4, a] = r3, Action[4, b] = r3

임을 알 수 있으며,

I_5 에서 $S \rightarrow CC\cdot$, \$ 는 a_f={\$} 이며, $S \rightarrow CC$는 식 (1) 이므로 r1이 되어

Action[5, \$] = r1

이 된다.

I_7 에서 $C \rightarrow b\cdot$, \$ 는 a_f={\$} 이며, $C \rightarrow b$는 식 (3) 이므로 r3가 되어

Action[7, \$] = r3

가 된다.

I_8 에서 $C \rightarrow aC\cdot$, \$ 는 a_f={\$} 이며, $C \rightarrow aC$는 식 (2) 이므로 r2가 되어

Action[8, \$] = r2

가 된다.

I_9 에서 $C \rightarrow aC\cdot$, a/b 는 a_f={a, b} 이며, $C \rightarrow aC$는 식 (2) 이므로 r2가 되어

Action[9, a] = r2, Action[9, b] = r2

가 된다.

규칙 (4)에 의하여 I_1에 $S' \rightarrow S\cdot$, \$ 이 있기 때문에

Action[1, \$] = accept

가 된다.

규칙 (5)에 의하여 각 단계에서 A를 구하면 I_0에서는 A = {C, S} 이므로, 또한 S는 I_1으로, C는 I_2로 나가므로

GO TO[0, C] = 2, GO TO[0, S] = 1

이 되며, I_1은 없고, I_2에서는 A = {C는 I_5로} 이므로

GO TO[2, C] = 5

이며, I_3는 A = {C는 I_9} 이므로

GO TO[3, C] = 9

이며, I_4, I_5는 A가 없으며, I_6은 A = {C는 I_8로} 이므로

GO TO[6, C] = 8

이며, I_7, I_8, I_9은 A가 없다.

CLR 파싱표를 구성해보면 아래와 같다.

ex) ⑤ CLR 파싱표

state	Action			GO TO	
	a	b	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			9
4	r3	r3			
5			r1		
6	s6	s7			8
7			r3		
8			r2		
9	r2	r2			

⑤ LALR 파싱표

LALR(lookahead LR) 파싱표를 만들기 위하여 문법 G는

$$\begin{aligned} G: \quad & S' \rightarrow S \\ & S \rightarrow CC \qquad \text{(문법 3-2)} \\ & C \rightarrow aC \mid b \end{aligned}$$

의 DFA를 앞 절에서 구성하였다.

이 DFA에서 항목 I_4와 I_7을 살펴보면 각기 첫 부분은 $C \rightarrow b$이며, 두 번째 부분은 lookahead가 I_4에서는 {a, b}이며, I_7에서는 {$}이다. 이 때 I_4와 I_7의 lookahead를 합하여 I_{47}으로 고치면서 항목을 $[C \rightarrow b,\ a/b/\$]$으로 고칠 수가 있다. 이 결과는 DFA 전체에 영향을 미치지 않음을 볼 수 있다. 이어서 항목의 첫 부분이 같은 I_8과 I_9의 lookahead를 합하여 I_{89}를 구성하여 항목 $[C \rightarrow aC,\ a/b/\$]$를 얻게 되는데, 이것도 DFA 전체에 영향을 미치지 않음을 알 수 있다.

I_3와 I_6 도 항목들의 첫 부분이 같으므로 lookahed를 합하여 I_{36}으로 구성할 수가 있어 다음과 같은 DFA와 파싱표를 구성할 수가 있다.

ex) ④

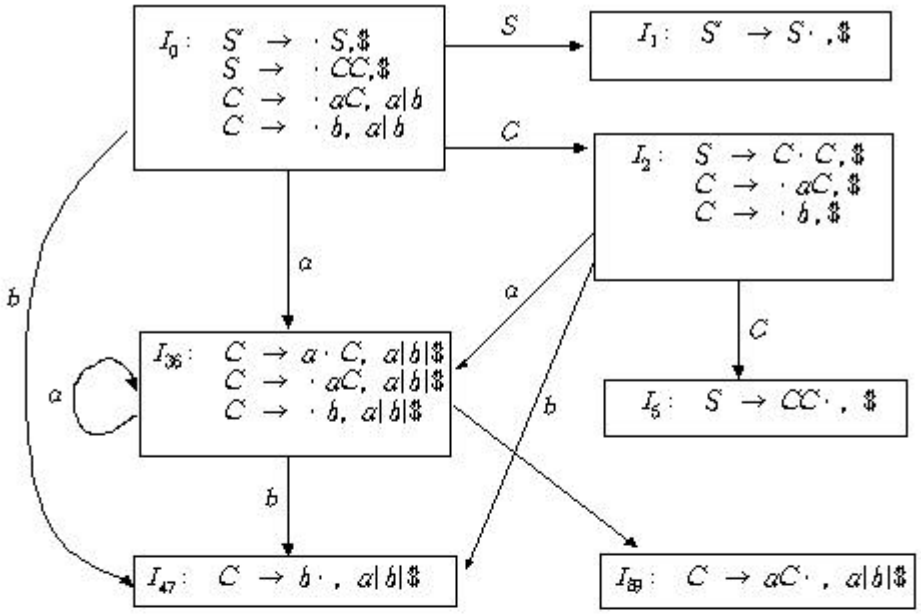


그림 7. 문법 3-2의 LALR의 DFA

ex) ⑤ LALR 파싱표

state	Action			Go To	
	a	b	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

CLR 파싱표에서 LALR 파싱표를 구성하는 방법을 설명하면 다음과 같다.

(1) LR(1) 항목들의 집합을 단계 $I_1, I_2, \ldots, I_n$으로 나타내고,

(2) 위의 항목집합에서 처음 부분이 같은 집합들을 같은 단계로 통합하면 새로운 항목들의 집합으로 단계 $J_1, J_2, \ldots, J_K$ 를 구성할 수 있으며, 이들 단계로 CLR의 경우과 같이 Action 부분을 만들 수 있다.

(3) 만일 $J = I_1 \cup I_2 \cup \ldots \cup I_m$ 이라면 Go To(I_1, X), Go To(I_2, X) … Go To(I_K, X) 들은 모두 같게 되고, 그 값이 K 라면 Go To(J, X) = K 가 된다.

LALR 파싱표과 CLR 파싱표를 비교하여 보면 LALR이 error를 나타내는 공간이 줄어 있음을 알 수 있다.

줄어든 error의 공간에 대하여 살펴보기 위하여 우선 문법

(0) $S' \rightarrow S$
(1) $S \rightarrow CC$
(2) $C \rightarrow aC$
(3) $C \rightarrow b$

의 허용되는 입력을 계산하여 이들 입력이 error의 공간에 어떤 영향를 미치는가 살펴보자.

우선 입력을 살펴보기 위하여

$$C = aC + b$$
$$L(C) = a^* b$$
$$L(S) = L(C)L(C)$$
$$= a^* b\, a^* b$$

즉, 입력은 $aa \cdots ab(aa \cdots a)b$의 형태를 갖게 되어 $aa \cdots abaab$ 의 형태나 $aa \cdots abb$ 의 형태가 허용되고 있다. 예를 들어 입력이 aab인 경우 CLR 파싱표를 이용하여 파싱을 해면 error가 발생되어야 하는데

0	a a b $
0 a 3	a b $
0 a 3 a 3	b $
0 a 3 a 3 b 4	$

에서 CLR의 Action[4, $] = error 임을 나타내어 만족스러우나 LALR에서는

0	a a b $
0 a 36	a b $
0 a 36 a 36	b $
0 a 36 a 36 b 47	$

이 되어 Action[47, $] = $r1$ 이 되며, LALR 스택은 다시 변하여

0 3 c 36 C 89, $

Action[89, $] = $r2$ 가 되며, 스택은 다시

0 C 2, $

이 된다. 이 때 Action[2, $] = error 가 되어 결국은 error가 발견된다.

따라서 CLR 파싱표를 LALR 파싱표로의 변환은 별다른 문제를 제기하지 않음을 볼 수 있으며, 다른 변화된 error란도 같은 방법으로 증명이 될 수 있다.

LALR 파싱표를 구성하는 데는 몇 가지 방법이 있는데, 가장 효과적인 방법의 하나를 소개해 보자. 우선 문법 G에서

0 $S' \rightarrow S$
1 $S \rightarrow CC$
2 $C \rightarrow aC$
3 $C \rightarrow b$

i) LR(1) 항목을 구한 다음 lookahead를 계산한다.

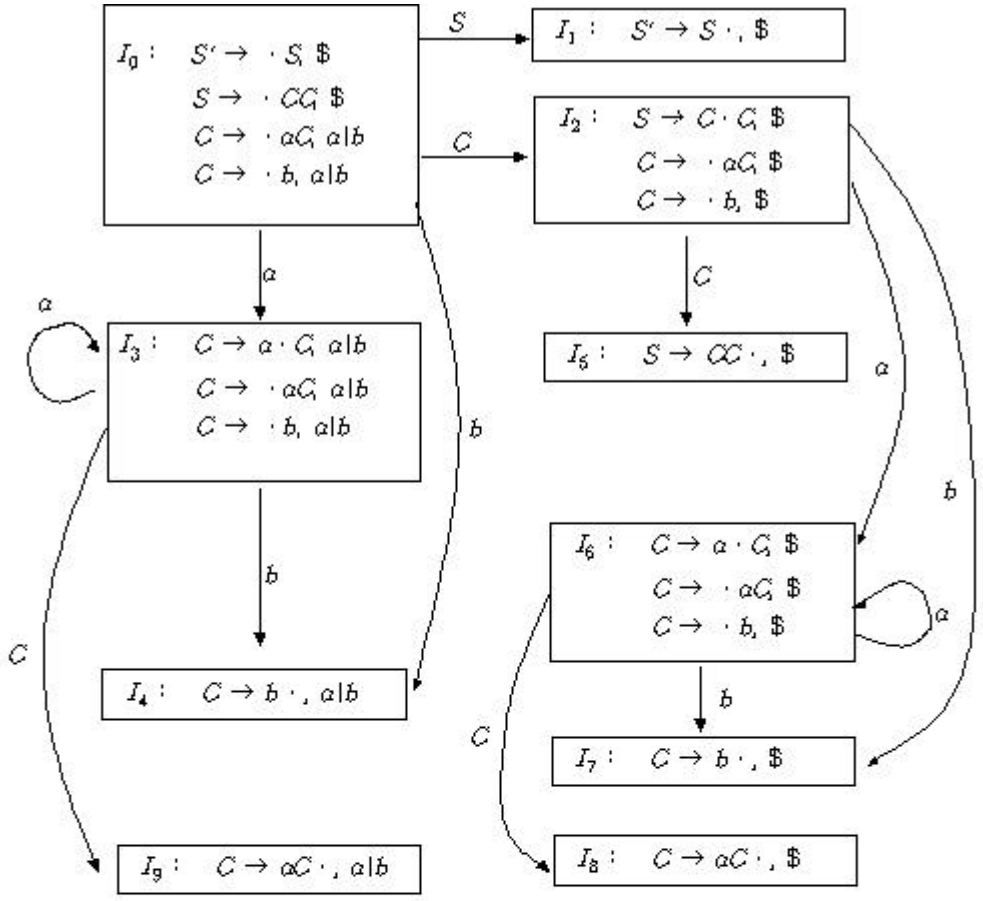


그림 8

ii) 각 단계에서 kernel을 구한다. 각 단계의 kernel은 각 단계에서 ε-전이로 생기지 않은 X-전이로 생긴 항목들(점선 안의 항목)로 다음의 3종들이다.

[$I_0, S'\rightarrow \cdot S,\ \$$], [$I_1, S'\rightarrow S \cdot ,\ \$$], [$I_2, S'\rightarrow C \cdot C,\ \$$]

[$I_3, C\rightarrow a \cdot C,\ a/b$], [$I_4, C\rightarrow b \cdot C,\ a/b$], [$I_5, S\rightarrow CC \cdot ,\ \$$]

[$I_6, C\rightarrow a \cdot C,\ \$$], [$I_7, C\rightarrow b \cdot ,\ \$$][$I_8, C\rightarrow aC \cdot ,\ \$$]

[$I_9, C\rightarrow aC \cdot ,\ a/b$]

등인데

iii) 첫 항목은 같고 lookahead만 다른 3종끼리는 lookahead를 합접시켜 새로운 kernel을 구성하는데 I_3와 I_6 I_4와 I_7, I_8와 I_9를 합치면

[$I_0, S'\rightarrow \cdot S,\ \$$]

[$I_1, S'\rightarrow S \cdot ,\ \$$]

[$I_2, S\rightarrow C \cdot C,\ \$$]

[$I_{36}, C\rightarrow a \cdot C,\ a/b/\ \$$]

[I_{47}, $C \rightarrow b \cdot$, $a/b/\$$]

[I_5, $S \rightarrow CC \cdot$, $\$$]

[I_{89}, $C \rightarrow aC \cdot$, $\$$]

들의 kernel을 구하게 되어

iv) 이들 kernel들을 이용하여 새로운 DFA를 구성하면 아래 그림 1과 같고, 그 LALR 파싱표는 앞에서 나온 LALR 파싱표와 같다.

kernel을 이용하는 방법으로 다음 문법의 LALR(1)파싱표를 구성해 보자.

0 $S' \rightarrow S$
1 $S \rightarrow L$
2 $S \rightarrow R$
3 $R \rightarrow L$
4 $L \rightarrow *R$
5 $L \rightarrow a$

(문법 4-1)

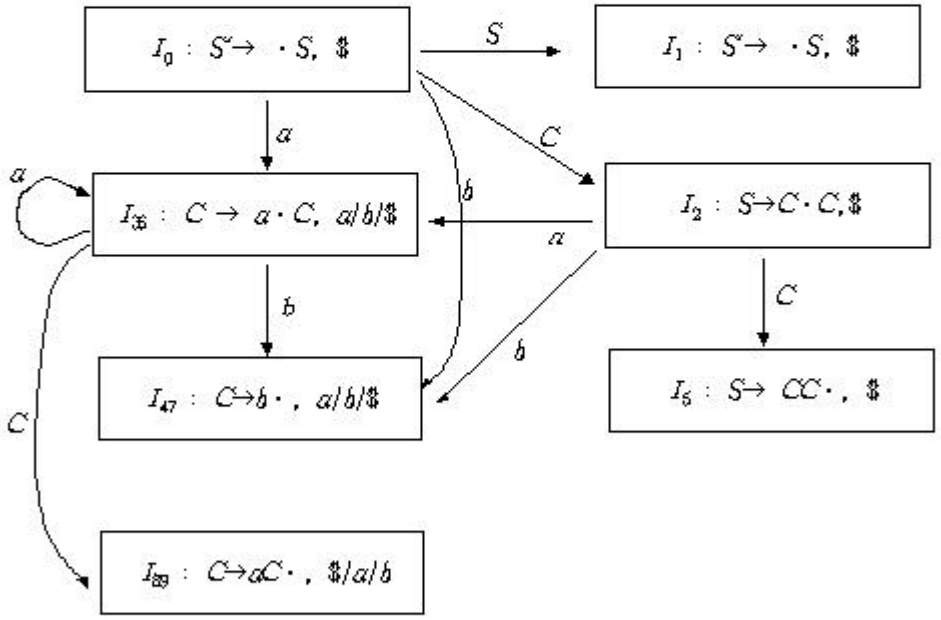


그림 9

위 문법을 이용하여 LR(0)항목을 구한 다음 해당하는 lookahead를 구해 보자.

각 단계에서 LR(0)항목을 구하는 것은 이전에 사용한 방법을 택하는데, I_0 에서는 $S' \rightarrow \cdot S$가 kernel이 되며, 나머지 항목들은 모두 ε-전이로 인하여 생산된 것이다.

I_1 단계는 I_0 단계에서 s-전이로 인하여 생산되었으며, 이 때 항목의 lookahead는 s-전이로 인하여 I_0의 kernel과 같은 lookahead를 가짐을 알 수 있다. 다른 여러 단계의 X-전이들은 모두 전이 전의 항목과 lookahead가 같음을 명심해야 할 것이다.

I_0 단계에서 ε-전이들의 lookahead를 계산해 보자 ε-전이의 경우 규칙에 의하면 $A \rightarrow \alpha \cdot B\beta,\ [a_1,\ldots,a_n]$과 $B \rightarrow \cdot\ r,\ T$의 경우 만일 $\beta=\varepsilon$ 이면 $T=[a_1,\ldots,a_n]$이며 β녹 ε이면 T=First(β)이므로

$[S \rightarrow \cdot\ L = R,\ T]$ $[S \rightarrow \cdot\ R,\ T]$

에서 T는 $[S' \rightarrow \cdot\ S,\ \$]$에서 $\beta=\varepsilon$이므로 T = {$}이 되어

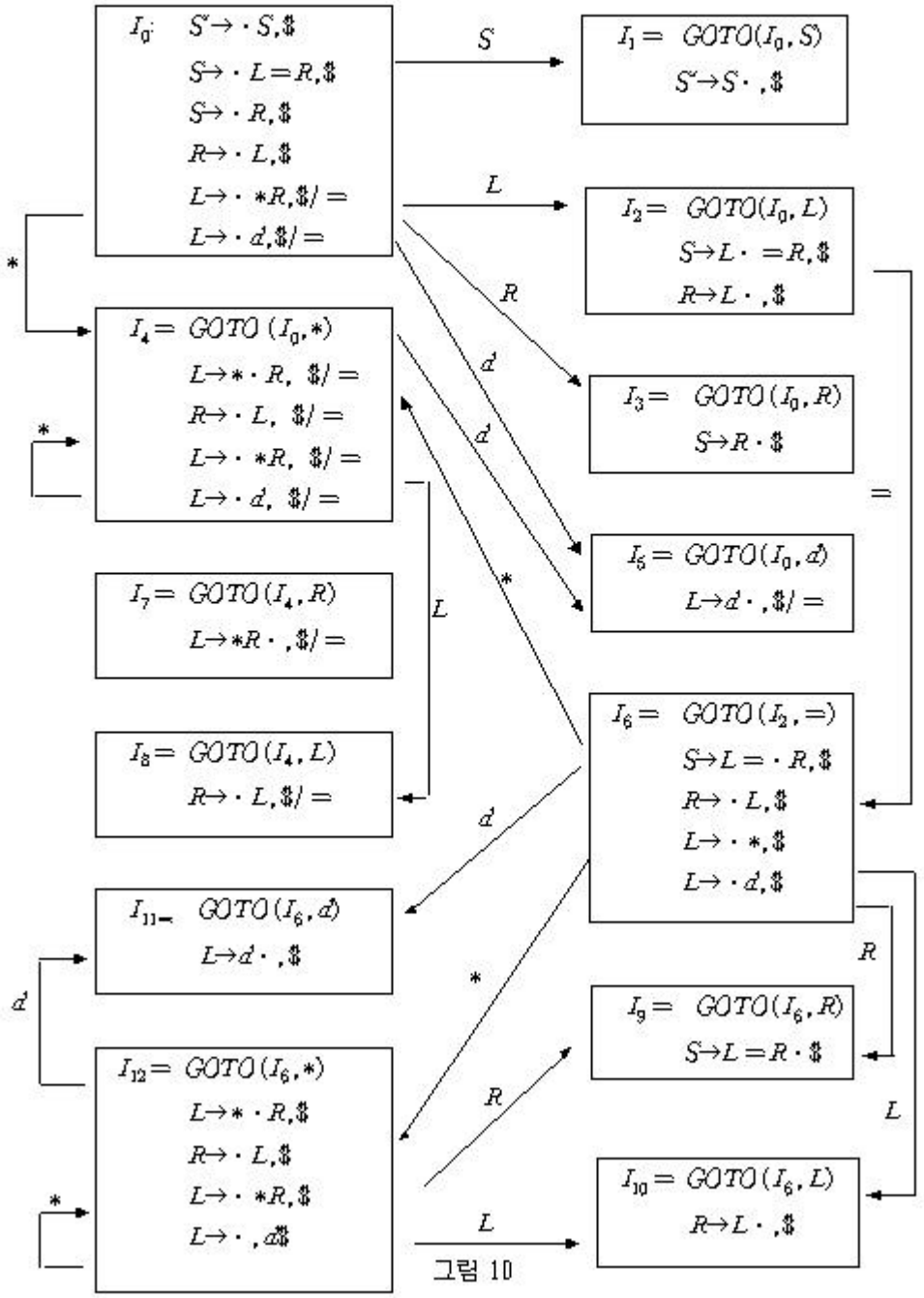


그림 10

[S→ · L = R, $] [S→ · R, $]

의 항목을 얻게 되며

[R→ · L, γ]

의 γ를 계산하려면 [S→ · R, $]에서 β=ε 이므로 γ = {$}를 얻게 될 것이다. 따라서

[R→ · L, $]

을 얻게 되며

[L→ · *R, γ], [L→ · α, γ]

에서 γ를 구하려면 [R→ · L, $]에서 β=ε이므로 γ⊃{$}임을 알 수 있으며, S → · L = R에서 β ==R이므로 First(β) = First(=R) = {=}이므로 γ ={$, =}임을 알 수 있다. 따라서

[L→ · *R, $/=], [L→ · α, $/=]

를 얻게 되며, 이들은 X-전이를 통하여 다른 항목의 kernel이 되게 된다.

각 항목의 점선 안은 kernel를 나타내는데, kernel들 중에서 lookahead만 다른 것을 살펴보면 I_{10} ⊂ I_8, I_{11}⊂ I_5, I_{12}⊂ I_4를 얻을 수가 있어 실제로 사용될 kernel들은 다음과 같다.

I_0 S' → · S, $
I_1 S' → S · , $
I_2 S → L · =R, $
R → L · , $
I_3 S → · R, $
I_4 L → * · R, $/=
I_5 L → α · , $/=
I_6 S → L = · R, $
I_7 L → * R · , $/=
I_8 R → · L, $/=
I_9 S → L = R · , $

이들을 이용하여 파싱표를 만들어 보면 다음과 같다.

	=	*	α	$	S	R	L
0		s4	s5		1	3	2
1				acc			
2	s6			r3			
3				r2			
4		s4	s5			7	8
5, 11	r5			r5			
6		s4	s5			9	8
7	r4			r4			
8, 10	r3			r3			
9				r1			

위 파싱표를 보면 이 문법은 LALR(1)임을 알 수 있다.

위의 파싱표를 검사하기 위하여 문법 (4-4)로 해당언어를 구해보면

S→L=R, S→R 에서

$$L(S) = L(L) = L(R) + L(R)$$

을 얻을 수 있으며,

$L \rightarrow *R$, $L \rightarrow d$ 에서 $L = *R + d$ 를 얻으며,

$$L(L) = *L(R) + d$$

를 얻을 수 있다.

또한 $R \rightarrow L$에서 $R = *R + d$ 를 얻어

$$L(R) = *^{*}d, \quad L(L) = **^{*}d + d$$

를 얻을 수 있으며,

$$L(S) = **^{*}d + d = *^{*} + *^{*}d$$

를 얻게 되어 $**\cdots *d = d\$$를 입력으로 사용해 보면 다음과 같은 파싱을 얻어 작업과정의정당성을 보일 수 있다.

0	$**\cdots *d = d\ \$$
0*4*4	$*\cdots *d = d\ \$$
⋮	
0*4*4 … *4	$d = d\ \$$
0*4*4 … *4 d 5	$= d\ \$$
0*4*4 … *4 L 8	$= d\ \$$
0*4*4 … *4 R 7	$= d\ \$$
0*4*4 … *L 8	$= d\ \$$
0*4*4 … *R 7	$= d\ \$$
0*4 L 8	$= d\ \$$
0*4 R 7	$= d\ \$$
0 L 2	$= d\ \$$
0 L 2 = 6	$d\ \$$
0 L 2 = 6 d 5	$\$$
0 L 2 = 6 L 8	$\$$
0 L 2 = 6 R 9	$\$$
0 S 1	$\$$
acc	