

# Project #2: LL 파서 작성 연습

국민대학교 컴퓨터공학과

20143046 김재희

## 1. Recursive descent 파서.

1) Identifier list를 인식하는 문법

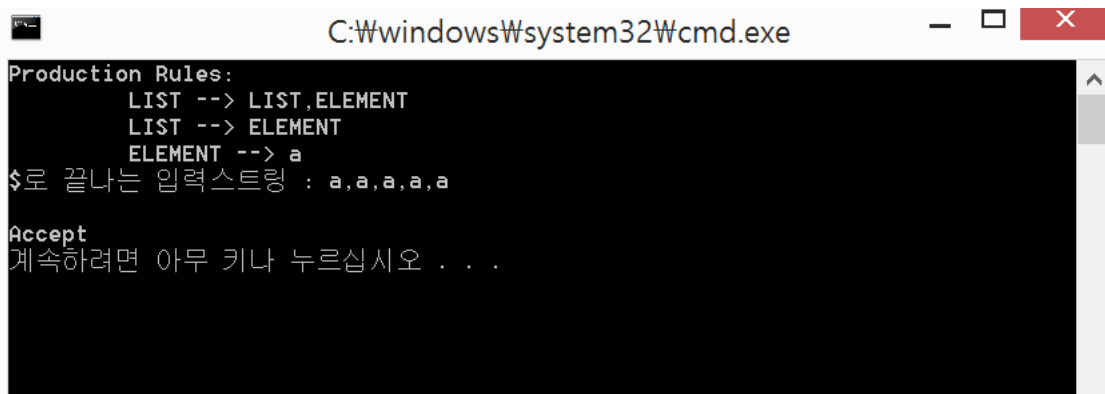
LIST -> LIST , ELEMENT

LIST -> ELEMENT

ELEMENT -> a

입력 예: "a,a,a,a,a"

### <실행결과>



```
C:\Windows\system32\cmd.exe

Production Rules:
    LIST --> LIST, ELEMENT
    LIST --> ELEMENT
    ELEMENT --> a
$로 끝나는 입력스트링 : a,a,a,a,a

Accept
계속하려면 아무 키나 누르십시오 . . .
```

→ 입력 스트링 a,a,a,a,a 를 제대로 인식하여 Accept를 출력한 모습이다.

### <코드구현>

우선, LIST -> LIST, ELEMENT는 left recursion을 형성하고 있으므로 left recursion을 없애준다.

없애준 후 : LIST-> ELEMENTLIST'

LIST' -> ,ELEMENTLIST'

ELEMENT-> a

### \*자료형 정의

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>

char input[20];
int i, error;
void LIST();
void LISTprime();
void ELEMENT();
```

Input[20]은 입력스트링.

error는 error일 경우 1을 대입시켜 준다.

i는 input의 입력 스트링 하나하나의 위치를 가르킨다.

LIST,LISTprime,ELEMENT은 nonterminal이다.

### \*LIST, LISTPRIME, ELEMENT 함수

```
void LIST()
{
    ELEMENT();
    LISTprime();
}
void LISTprime()
{
    if (input[i] == ',') {
        i++;
        ELEMENT();
        LISTprime();
    }
}
void ELEMENT() {
    if (input[i] == 'a') {
        i++;
    }
    else {
        error = 1;
    }
}
```

-LIST() :

LIST-> ELEMENTLIST' 이기 때문에

nonterminal인 ELEMENT를 인식한 후에  
nonterminal인 LIST'를 인식하는 함수이다.

-LISTprime() :

LIST'-> ,ELEMENTLIST' 이기 때문에

terminal인 ,를 인식한 후에 인식한 스트링의  
길이 만큼 i를 더해준 후 nonterminal인  
ELEMENT를 인식한 후에 LIST'를 인식하는 함수이다.

. -ELEMENT() :

ELEMENT-> a 이기 때문에

terminal인 a를 인식한 후에 인식한 스트링의  
길이 만큼 i를 더해준다. a가 아니면 error이  
기 때문에 error에 1을 대입한다.

\*main 함수

```
void main()
{
    i = 0;
    error = 0;
    puts("Production Rules:");
    puts("### LIST --> LIST,ELEMENT");
    puts("### LIST --> ELEMENT");
    puts("### ELEMENT --> a");
    printf("$로 끝나는 입력스트링 : ");

    gets(input);
    LIST();
    if (strlen(input) == i && error == 0)
        printf("###Accept###");
    else
        printf("###FAIL###");
}
```

Puts를 이용하여 화면에 문법을 정리해서 나타내준다.

Gets를 이용하여 input을 받아오고 시작 심벌인 LIST함수를 사용하여 파서를 시작한다.

스트링의 길이인 i만큼 끝까지 인식하고 error가 발생하지 않아 0이라면 accept를 출력하고 그렇지 않으면 FAIL을 출력한다.

2) 덧셈, 곱셈 수식을 인식하는 문법

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

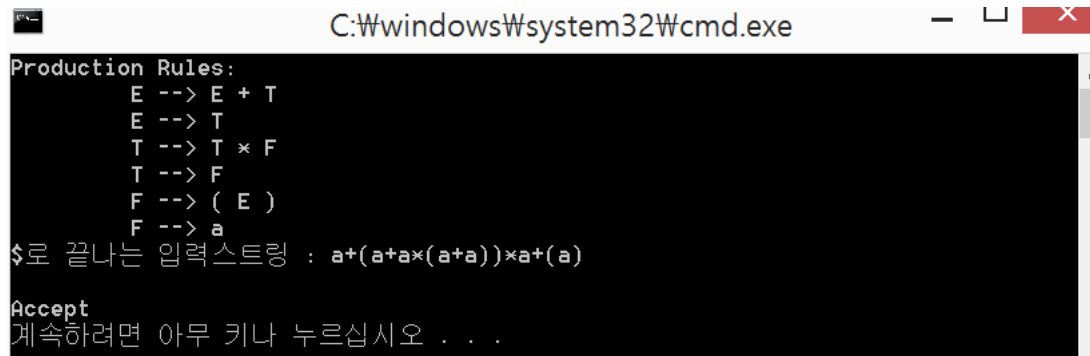
$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow a$

입력 예: "a+(a+a\*(a+a))\*a+(a)"

### <실행결과>



```
C:\Windows\system32\cmd.exe

Production Rules:
    E --> E + T
    E --> T
    T --> T * F
    T --> F
    F --> ( E )
    F --> a
$로 끝나는 입력스트링 : a+(a+a*(a+a))*a+(a)

Accept
계속하려면 아무 키나 누르십시오 . . .
```

➔ 입력 스트링  $a+(a+a*(a+a))*a+(a)$  를 제대로 인식하여 Accept를 출력한 모습이다.

### <코드구현>

우선, 문법이 left recursion을 형성하고 있으므로 left recursion을 없애준다.

없애준 후 :  $E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow \text{epsilon}$

$T \rightarrow FT'$

$T' \rightarrow *FT'$

$T' \rightarrow \text{epsilon}$

$F \rightarrow (E)$

$F \rightarrow a$

## \*자료형 정의

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
```

```
char input[20];
int i, error;
void E();
void T();
void Eprime();
void Tprime();
void F();
```

Input[20]은 입력스트링.

error는 error일 경우 1을 대입시켜 준다.

i는 input의 입력 스트링 하나하나의 위치를 가르킨다.

E, T, E', T', F는 nonterminal이다.

## \*E, T, Eprime, Tprime, F 함수

```
void E()
{
    T();
    Eprime();
}
void Eprime()
{
    if (input[i] == '+')
    {
        i++;
        T();
        Eprime();
    }
}
void T()
{
    F();
    Tprime();
}
void Tprime()
{
    if (input[i] == '*')
    {
        i++;
        F();
        Tprime();
    }
}
```

-E() :

E-> TE'이기 때문에

nonterminal인 T를 인식한 후에 nonterminal인 E'를 인식하는 함수이다.

-Eprime() :

E'-> +TE' 이기 때문에

terminal인 +를 인식한 후에 인식한 스트링의 길이 만큼 i를 더해준 후 nonterminal인 T를 인식한 후에 E'를 인식하는 함수이다.

-T() :

T-> FT' 이기 때문에

nonterminal인 F를 인식한 후에 nonterminal인 T'를 인식하는 함수이다.

-Tprime() :

T'-> \*FT' 이기 때문에

terminal인 \*를 인식한 후에 인식한 스트링의 길이 만큼 i를 더해준 후 nonterminal인 F를 인식한 후에 T'를 인식하는 함수이다.

```

void F()
{
    if (input[i] == 'a') i++;
    else if (input[i] == '(')
    {
        i++;
        E();
        if (input[i] == ')')
            i++;

        else error = 1;
    }

    else error = 1;
}

```

-F() :

$F \rightarrow (E) \mid a$  이기 때문에

terminal인 'a'를 인식한 후에 인식한 스트링의 길이만큼 i를 더해주거나, terminal인 '('를 인식한 후에 i를 1더해준 후 nonterminal인 E를 인식한 후에 terminal인 ')'를 인식하고 i를 1더해주는 함수이다.

만약 이렇게 입력되지 않았다면 error를 발생시키기 위해 1을 대입해준다.

\*main 함수

```

void main()
{
    i = 0;
    error = 0;
    puts("Production Rules:");
    puts("\tE -> E + T");
    puts("\tE -> T");
    puts("\tT -> T * F");
    puts("\tT -> F");
    puts("\tF -> ( E )");
    puts("\tF -> a");
    printf("$로 끝나는 입력스트링 : ");

    gets(input);
    E();
    if (strlen(input) == i && error == 0)
        printf("\nAccept\n");
    else
        printf("\nFAIL\n");
}

```

Puts를 이용하여 화면에 문법을 정리해서 나 타내준다.

Gets를 이용하여 input을 받아오고 시작심벌인 E함수를 사용하여 파서를 시작한다.

스트링의 길이인 i만큼 끝까지 인식하고 error가 발생하지 않아 0이라면 accept를 출력하고 그렇지 않으면 FAIL을 출력한다.

## 2. Predictive 파서.

1) Identifier list를 인식하는 문법

LIST -> LIST , ELEMENT

LIST -> ELEMENT

ELEMENT -> a

입력 예: "a,a,a,a,a"

<실행결과>

```
Input a,a,a,a,a : a,a,a,a,a
Input string : a,a,a,a,ac (c is an endig mark)
[CFG 0] A -> CB
[CFG 3] C -> a
[CFG 1] B -> ,CB
[CFG 3] C -> a
[CFG 1] B -> ,CB
[CFG 3] C -> a
[CFG 1] B -> ,CB
[CFG 3] C -> a
[CFG 1] B -> ,CB
[CFG 3] C -> a
[CFG 1] B -> ,CB
[CFG 3] C -> a
[CFG 2] B ->
[accept]
계속하려면 아무 키나 누르십시오 . . .
```

\*자료형 정의

```
#include <stdio.h>
#include <string.h>
#pragma warning(disable: 4996)

#define NONTERMINALS 3
#define TERMINALS 2
#define RULESIZE 3
#define MAX 100

#define INPUT_END_MARK ('a'+TERMINALS)
#define STACK_BOTTOM_MARK '_'

char create_rule[RULESIZE][MAX];
int parsing_table[NONTERMINALS][TERMINALS + 1];
char stack[999];
int top;
```

Nonterminal은 A,B,C로 3개고  
terminal은 ',', 'a'로 2개다.

정의된 생성규칙은 3개이다.

끝마크 문자로 터미널개수만큼 더  
해준 다음의 문자를 지정해준다.

Parsing table은 열에 끝마크문자를  
포함하여 terminal+1만큼, 행에  
nonteminal을 넣어준다.



### \*load\_create\_rule 함수

```
void load_create_rule()
{
    strcpy(create_rule[0], "CB");
    strcpy(create_rule[1], ",CB");
    strcpy(create_rule[2], "");
    strcpy(create_rule[3], "a");
}
```

기존의 문법의 left recursion을 제거 하여 LL(1)문법으로 만든 후 문자를 인덱스로 표시하기 위하여 아래와 같이 차례대로 A, B..nonterminal 이름 변경..

0. A-> CB
1. B-> ,CB
2. B-> epsilon
3. C-> a

각 배열에 생성규칙을 각각 넣어준다.

### \*load\_parsing\_table 함수

Vn/Vt	a	,	\$(c)
A	0	-1	-1
B	-1	1	2
C	3	-1	-1

```
void load_parsing_table()
{
    parsing_table[0][0] = 0;
    parsing_table[1][1] = 1;
    parsing_table[1][2] = 2;
    parsing_table[2][0] = 3;
}
```

위와 같이 만들어 놓은 파싱 테이블을 2차원 배열에 넣어준다.

### \*predictive\_parsing 함수

```
else { // expand -- nonterminal
    if (*p == 'a' || *p == 'c') {
        index = parsing_table[stack_top() - 'A'][*p - 'a'];
    }
    else {
        index = parsing_table[stack_top() - 'A'][1];
    }
    if (index != -1) {
        str_p = rule[index];
        len = strlen(str_p);
        printf("[CFG %d] %c -> %s\n", index, stack_top(), str_p);
        pop();
        for (i = len - 1; i >= 0; i--) {
            push(str_p[i]);
        }
    }
    ...
}
```

->만약 input이 a,c라면 index를 구해주기 위해 'a'를 빼고 ','라면 parsing table의 열이 1이다.

2) 덧셈, 곱셈 수식을 인식하는 문법

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow a$

입력 예: "a+(a+a\*(a+a))\*a+(a)"

<실행결과>

```
Input(a+(a+a*(a+a))*a+(a) : a+(a+a*(a+a))*a+(a)
Input string : a+(a+a*(a+a))*a+(a)f (f is an endig mark)
[CFG 0] A -> CB
[CFG 3] C -> ED
[CFG 7] E -> a
[CFG 5] D ->
[CFG 1] B -> +CB
[CFG 3] C -> ED
[CFG 6] E -> (A)
[CFG 0] A -> CB
[CFG 3] C -> ED
[CFG 7] E -> a
[CFG 5] D ->
[CFG 1] B -> +CB
[CFG 3] C -> ED
[CFG 7] E -> a
[CFG 4] D -> *ED
[CFG 6] E -> (A)
[CFG 0] A -> CB
[CFG 3] C -> ED
[CFG 7] E -> a
[CFG 5] D ->
[CFG 1] B -> +CB
[CFG 3] C -> ED
[CFG 7] E -> a
[CFG 5] D ->
[CFG 2] B ->
[CFG 5] D ->
[CFG 2] B ->
[CFG 4] D -> *ED
[CFG 7] E -> a
[CFG 5] D ->
[CFG 1] B -> +CB
[CFG 3] C -> ED
[CFG 6] E -> (A)
[CFG 0] A -> CB
[CFG 3] C -> ED
[CFG 7] E -> a
[CFG 5] D ->
[CFG 2] B ->
[CFG 5] D ->
[CFG 2] B ->
[accept]
계속하려면 아무 키나 누르십시오 . . .
```

## \*자료형 정의

```
#include <stdio.h>
#include <string.h>
#pragma warning(disable: 4996)

#define NONTERMINALS 5
#define TERMINALS 5
#define RULESIZE 8
#define MAX 100

#define INPUT_END_MARK ('a'+TERMINALS)
#define STACK_BOTTOM_MARK '_'

char create_rule[RULESIZE][MAX];
int parsing_table[NONTERMINALS][TERMINALS + 1];
char stack[999];
int top;
```

Nonterminal은 A,B,C,D,E로 5개고  
terminal은 +, \*, (, ), a로 5개다.

정의된 생성규칙은 8개이다.

끝마크 문자로 터미널개수만큼 더  
해준 다음의 문자를 지정해준다.

Parsing table은 열에 끝마크문자를  
포함하여 terminal+1만큼, 행에  
nonteminal을 넣어준다.

## \*load\_create\_rule 함수

```
void load_create_rule()
{
    strcpy(create_rule[0], "CB");
    strcpy(create_rule[1], "+CB");
    strcpy(create_rule[2], "");
    strcpy(create_rule[3], "ED");
    strcpy(create_rule[4], "*ED");
    strcpy(create_rule[5], "");
    strcpy(create_rule[6], "(A)");
    strcpy(create_rule[7], "a");
}
```

기존의 문법의 left recursion을 제거 하여 LL(1)문법  
으로 만든 후 문자를 인덱스로 표시하기 위하여 아  
래와 같이 차례대로 A, B..nonterminal 이름 변경..

0. A-> CB
1. B-> +CB
2. B-> epsilon
3. C-> ED
4. D-> \*ED
5. D-> epsilon
6. E-> (A)
7. E-> a

각 배열에 생성규칙을 각각 넣어준다.

## \*load\_parsing\_table 함수

Vn/Vt	a	(	)	*	+	\$(f)
A	0	0	-1	-1	-1	-1
B	-1	-1	2	-1	1	2
C	3	3	-1	-1	-1	-1
D	-1	-1	5	4	5	5
E	7	6	-1	-1	-1	-1

```

void load_parsing_table()
{
    parsing_table[0][0] = 0;
    parsing_table[0][1] = 0;
    parsing_table[1][2] = 2;
    parsing_table[1][4] = 1;
    parsing_table[1][5] = 2;
    parsing_table[2][0] = 3;
    parsing_table[2][1] = 3;
    parsing_table[3][2] = 5;
    parsing_table[3][3] = 4;
    parsing_table[3][4] = 5;
    parsing_table[3][5] = 5;
    parsing_table[4][0] = 7;
    parsing_table[4][1] = 6;
}

```

위와 같이 만들어 놓은 파싱 테이블을 2차원 배열에 넣어준다.

\*stack\_pop, push, pop, init\_stack 함수

```

char stack_top()
{
    return stack[top - 1];
}

void push(char ch)
{
    stack[top++] = ch;
}

void pop()
{
    top--;
}

void init_stack()
{
    top = 0;
    push(STACK_BOTTOM_MARK);
    push('A');
}

```

-stack\_top() :

Stack의 top의 값을 반환한다.

-push(char ch) :

Stack의 top에 stack을 쌓고 다음 stack을 가르킨다.

-pop() :

Stack을 삭제한다.

-init\_stack() :

Stack을 초기화하는 함수로 top=0이며 시작 심벌 이자 nonterminal인 'A'를 넣어준다.

### \*is\_nonterminal, is\_terminal 함수

```
int is_nonterminal(char ch)
{
    if ('A' <= ch && ch <= 'Z') return 1;
    else return 0;
}

int is_terminal(char ch)
{
    return (is_nonterminal(ch) == 0);
}
```

-is\_nonterminal(char ch) :  
Nonterminal인 A,B,C,D,E가 입력되면  
1을 return  
-is\_terminal(char ch) :  
Terminal이 입력되면 nonterminal은  
0을 return

### \*predictive\_parsing 함수

```
void predictive_parsing(int table[NONTERMINALS][TERMINALS + 1], char rule[RULESIZE][MAX], char *input)
{
    char *p = input;
    char *str_p;
    int i, index = 0, len;

    while (1) {
        if (stack_top() == STACK_BOTTOM_MARK) {
            if (*p == INPUT_END_MARK)
                printf("[accept]\n"); // parsing OK
            else printf("[error] -- Input is not empty!\n");
            return;
        }

        if (is_terminal(stack_top())) { // pop -- terminal symbol
            if (*p == stack_top()) {
                pop();
                p++;
            }
        }
        else {
            printf("[error] -- Stack is not empty!\n");
            return;
        }
    }
}
```

\*P에 input 스트링을 넣어준다.

while문을 돌리면서 stack도 비었고 input스트링도 비었다면 제대로 인식하였기 때문에 accept를 출력하고 그렇지 않다면 FAIL을 출력한다.

만약 terminal symbol이 stack의 top과 input스트링과 같다면 pop해준다.

```

else { // expand -- nonterminal
    if (*p == 'a' || *p == 'f') {
        index = parsing_table[stack_top() - 'A'][*p-'a'];
    }
    else {
        index = parsing_table[stack_top() - 'A'][*p - '(' + 1];
    }
    if (index != -1) {
        str_p = rule[index];
        len = strlen(str_p);
        printf("[CFG %d] %c -> %s\n", index, stack_top(), str_p);
        pop();
        for (i = len - 1; i >= 0; i--) {
            push(str_p[i]);
        }
    }
    else { // error
        printf("[error] %c -> %c\n", stack_top(), *p);
        return;
    }
}
}
}
}

```

입력 스트링이 a이거나 끝마크인 f라면 파싱테이블의 열이 0또는 5이므로 인덱스로 바뀌  
주기 위해 'a'를 빼준다.

(, \*, + 라면 아스키코드가 28,29,30,31이므로 제일 작은 순서대로 인덱스를 1,2,3,4 로  
지정해주기 위해 '(' 을 빼준 후 1을 더해준다.

생성규칙이 있다면 생성규칙을 적용한 후 스택을 pop해준다.

## \*main 함수

```

int main()
{
    char input[100];

    load_create_rule();
    load_parsing_table();

    input_data(input);
    init_stack();

    predictive_parsing(parsing_table, create_rule, input);
    return 0;
}

```

생성규칙과 parsing table을 정  
의해주고 입력스트링을 받은  
후 stack을 초기화 시켜준 후  
parsing을 시작한다.