

Compiler Term-Project #2

02분반 26조

1) Non-ambiguous CFG

Index	LHS	->	RHS
accept	S	->	CODE
1	CODE	->	VDECL CODE
2	CODE	->	FDECL CODE
3	CODE	->	CDECL CODE
4	CODE	->	"
5	VDECL	->	vtype id semi
6	VDECL	->	vtype ASSIGN semi
7	ASSIGN	->	id assign RHS
8	RHS	->	EXPR
9	RHS	->	literal
10	RHS	->	character
11	RHS	->	boolstr
12	EXPR	->	EXPR' addsub EXPR
13	EXPR	->	EXPR'
14	EXPR'	->	EXPR'' multdiv EXPR'
15	EXPR'	->	EXPR''
16	EXPR''	->	lparen EXPR rparen
17	EXPR''	->	id
18	EXPR''	->	num
19	FDECL	->	vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
20	ARG	->	vtype id MOREARGS
21	ARG	->	"
22	MOREARGS	->	comma vtype id MOREARGS
23	MOREARGS	->	"
24	BLOCK	->	STMT BLOCK
25	BLOCK	->	"
26	STMT	->	VDECL
27	STMT	->	ASSIGN semi
28	STMT	->	if lparen COND rparen lbrace BLOCK rbrace ELSE

29	STMT	->	while lparen COND rparen lbrace BLOCK rbrace
30	COND	->	COND' comp COND boolstr
31	COND'	->	boolstr
32	ELSE	->	else lbrace BLOCK rbrace
33	ELSE	->	"
34	RETURN	->	return RHS semi
35	CDECL	->	class id lbrace ODECL rbrace
36	ODECL	->	VDECL ODECL
37	ODECL	->	FDECL ODECL
38	ODECL	->	"

Change in CFG: 기본으로 주어진 CFG에서 5번과 14번이 ambiguous함을 확인하였고, 이에 따라 CFG를 수정/추가하였습니다.

CFG	Before Change	After Change
5	EXPR→EXPR addsub EXPR EXPR multdiv EXPR	EXPR → EXPR' addsub EXPR EXPR'
		EXPR' → EXPR" multdiv EXPR' EXPR"
		EXPR" → lparen EXPR rparen id num
14	COND → COND comp COND boolstr	COND → COND' comp COND boolstr
		COND' → boolstr

- 5번 문법의 경우, 연산자의 우선순위 때문에 모호성이 발생합니다. 이를 해소하기 위해 기존 EXPR 토큰을 정의하던 CFG 5번과 6번을 통합하여 세 개의 다른 토큰으로 분리해 내었고, 이를 연산자의 우선 순위에 맞게 곱하기와 나누기 연산이 먼저 실행되도록 CFG를 재정의 하였습니다. 그 결과가 EXPR, EXPR' 그리고 EXPR"이고, 새로운 CFG는 위의 표에 기술되어 있습니다.
- 14번 문법의 경우, LHS에 있는 COND 토큰이 RHS에서 두 번이나 재귀적으로 등장하면서 순서 상으로 모호함이 생기는 것을 확인하였습니다. 이를 방지하기 위해 COND 토큰을 서로 다른 두 개의 토큰으로 분리하여 순서상으로 모호하게 호출되는 구조를 제거해 주었습니다. 그 결과가 COND와 COND'이고 새로운 CFG는 위의 표에 기술되어 있습니다.

2) SLR Parsing Table

[illegible]

*캡처화면의 화질이 좋지 않아 제출 파일에 추가적으로 SLR Table을 복사한 엑셀 파일을 첨부하였습니다

3) Change in Lexer

이번 2차 프로젝트를 진행하면서 지난 1차 제출했던 Lexical Analyzer에서의 변경점은 크게 세 가지가 있습니다.

첫번째, 토큰의 구분인자가 ';'에서 ':'와 '|'로 바뀌었습니다. 이는 구분인자로 ';'를 사용할 경우, 토큰 테이블이 생성된 후에 syntax analyzer에서 이를 받아들일 때, 구분인자로 적은 ';'가 토큰으로 인식되는 현상을 방지하기 위함입니다. 따라서 토큰의 시작을 '~'로, 끝을 '|'로 구분하였습니다.

	Before Change	After Change
Token Format	<Identifier,a>	~Identifier:a

두번째, 토큰의 분류가 끝나고 이를 파일에 입력할 때 입력된 코드에 맞춰 줄 바꿈을 삽입하였습니다. 이는 analyzer의 파싱 도중 오류가 생겼을 때, 로그를 효율적으로 출력하기 위함입니다. 이를 통해 오류가 생겼을 때, 몇 번째 줄, 어떤 토큰에서 오류가 발생하였는지 바로 확인할 수 있습니다.

```
if ((int)input == 10) //if next input is '\n', make line indexing to the answer
{
    line_num++;
    token_index = 1; //initialize token index for new line
    answer.push_back({ "newLine", "" });
}
```

마지막으로는, 기존의 Lexer와 달리, 변경된 Lexer에서는 에러가 났을 경우 이를 Log형태로 report를 띄워줍니다. 기존에는 단순히 '에러 발생'으로 처리하고 끝냈다면, 이제는 Syntax Analyzer와 마찬가지로 몇 번째 줄에서 어느 토큰을 파싱하던 중 에러가 발생했는지를 출력해줍니다.

The screenshot shows a code editor window titled 'input.txt' with the following C code:

```
1 int c(int d,int e,int f)
2 {
3     int g;
4     h=true;
5     string = "strll;
6     i=(j);
7     if (true)
8     {
9         while(true <= false)
10        {
11            i=h+j;
12        }
13    }
14    return i;
15 }
```

Below the code editor is a terminal window showing the execution of the lexical analyzer. The command executed is `./lexical_analyzer.exe input.txt`. The output shows a lexical error:

```
lexical_error : there can't be token named '"strll;' in 3th token of line number
5
```

다음은 Lexer의 변경 이후 test code를 받아 출력하는 방식의 전체적인 차이를 도표로 나타낸 것입니다.

Input Code	int a; b= c+d;
Token Result Before Change	<Variable_type,int> <Identifier,a> <Semicolon,;> <Identifier,b> <Assign_op,=> <Identifier,c> <Arithmetic_op,+> <Identifier,d> <Semicolon,;>
Token Result After Change	~Variable_type:int ~Identifier:a ~Semicolon:; ~Identifier:b ~Assign_op:= ~Identifier:c ~Arithmetic_op:+ ~Identifier:d ~Semicolon:;

4) Implementation Details

-Data Structure

1) token

각 데이터를 저장하는 기본 단위로, 구조체형을 사용했습니다. Token이 가지고 있는 인자로는,

data = token의 정보

type = token의 type (terminal number)

loc = token의 행렬정보 {행, 열}

가 있습니다.

2) S_R_table

Shift와 Reduce연산 수행을 위해 만들어진 2차원 배열로, 현재 상태와 토큰의 종류에 따라 다음에 취할 행동을 저장하고 있습니다. 만약 아무런 값이 지정되지 않은 index의 원소에 접근한다면 에러를 출력합니다.

4) Trans_table

token의 type(char*)을 int 형으로 변환하기 위한 벡터형 리스트입니다.

5) Reduce_Rule

Reduce 연산을 위해 필요한 CFG를 저장해놓은 배열입니다. CFG number를 index로 하여, {다음상태, 삭제해야 하는 원소 수}를 저장합니다.

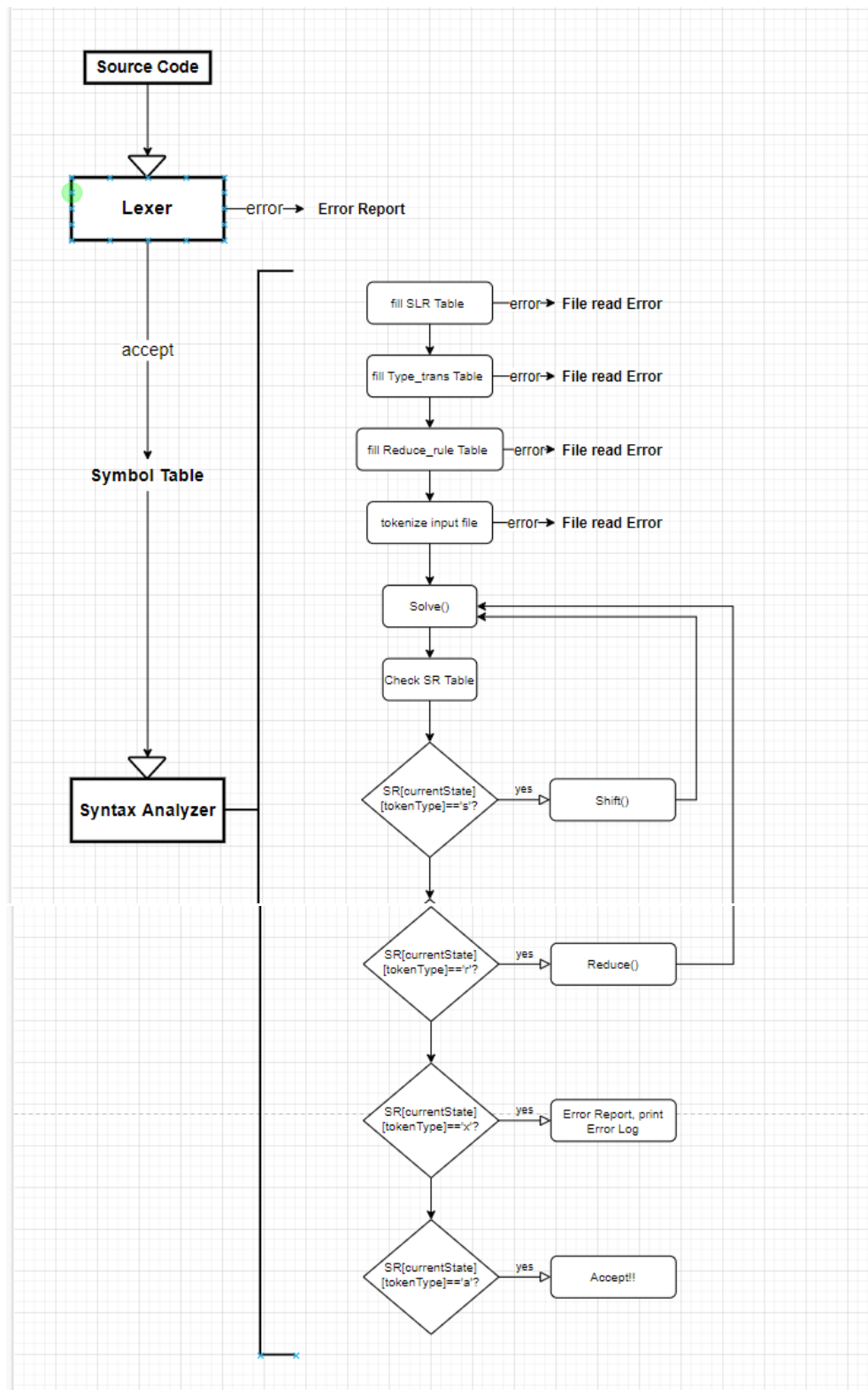
6) Token_list(vector)

프로그램 실행 시 인자로 받은 파일명을 읽어, 그 안에 있는 값을 토큰 단위로 모두 저장합니다. 마지막 문자 '\$'(terminal) 은 직접 넣어주어야 합니다.(output file 에는 포함되지 않습니다.)

7) local(stack)

첫 상태(0)를 푸쉬해서 초기화한 스택입니다. 그 이후 Token_List를 Shift Reduce 연산을 반복하며 파싱하면서 state들을 대상으로 push 혹은 pop을 수행합니다.

-Overall Procedure



-Algorithms

solve()

토큰을 차례대로 읽으며 next token의 상태에 따라 shift reduce 가 정해지며 S_R_table을 따라 Token을 왼쪽에서 오른쪽으로 parse 합니다. S_R_table을 따라 다음 실행(cmd)을 결정합니다

*cmd[0] 가 s 일때

- Shift 를 실행합니다. cmd[1~] 에서 다음 상태를 알 수 있습니다.

*cmd[0] 가 r 일때

- Reduce 를 실행합니다. cmd[1~] 에서 Reduce_Rule number를 알 수 있습니다.

*cmd[0] 가 a 일때

- acc 즉 parse가 성공하였으므로(accept) 정상종료합니다.

*cmd[0] 가 x 일때

- no next command, 다음동작이 없는 잘못된 접근이 일어났기 때문에 에러발생을 return 합니다.

Shift(token cur, string cmd)

다음상태를 stack의 top에 저장하고 종료합니다. 이 함수는 실패(잘못된 접근)가 없습니다.

Reduce(token cur, string cmd)

일단 주어진 Reduce Rule에 따라 규칙에서 지정한 원소수만큼 stack 에서 pop하여 Reduce 를 실행합니다. 그렇게 생성된 non terminal을 받아들이는 것으로 동작을 마칩니다. 이 함수는 새로운 non terminal을 받아들이는 과정에서 실패(잘못된 접근)이 일어날 수 있으며 실패가 일어날 경우 에러를 리턴합니다.

4) Result

Solve 함수가 종료되면 Shift Reduce의 결과와 그에 따른 Syntax analysis의 결과가 accept되었는지, 혹은 error report를 보냈는지 출력됩니다.

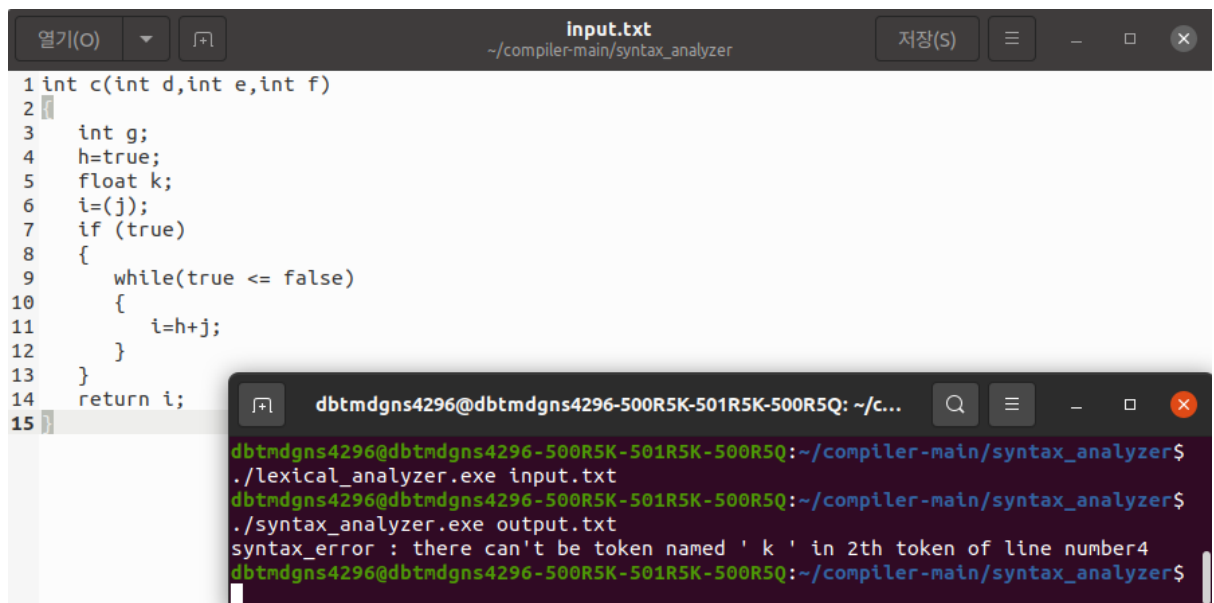
1) Accept



```
1 int c(int d,int e,int f)
2 {
3     int g;
4     h=true;
5     int k;
6     i=(j);
7     if (true)
8     {
9         while(true <= false)
10        {
11            i=h+j;
12        }
13    }
14    return i;
15 }
```

```
dbtmdgns4296@dbtmdgns4296-500R5K-501R5K-500R5Q: ~/c...
dbtmdgns4296@dbtmdgns4296-500R5K-501R5K-500R5Q:~/compiler-main/syntax_analyzer$ ./lexical_analyzer.exe input.txt
dbtmdgns4296@dbtmdgns4296-500R5K-501R5K-500R5Q:~/compiler-main/syntax_analyzer$ ./syntax_analyzer.exe output.txt
accept
dbtmdgns4296@dbtmdgns4296-500R5K-501R5K-500R5Q:~/compiler-main/syntax_analyzer$
```

2) Reject / Error Report



```
1 int c(int d,int e,int f)
2 {
3     int g;
4     h=true;
5     float k;
6     i=(j);
7     if (true)
8     {
9         while(true <= false)
10        {
11            i=h+j;
12        }
13    }
14    return i;
15 }
```

```
dbtmdgns4296@dbtmdgns4296-500R5K-501R5K-500R5Q: ~/c...
dbtmdgns4296@dbtmdgns4296-500R5K-501R5K-500R5Q:~/compiler-main/syntax_analyzer$ ./lexical_analyzer.exe input.txt
dbtmdgns4296@dbtmdgns4296-500R5K-501R5K-500R5Q:~/compiler-main/syntax_analyzer$ ./syntax_analyzer.exe output.txt
syntax_error : there can't be token named ' k ' in 2th token of line number4
dbtmdgns4296@dbtmdgns4296-500R5K-501R5K-500R5Q:~/compiler-main/syntax_analyzer$
```

에러가 발생할 경우, 몇 번째 줄의 어느 토큰을 파싱하던 중 에러가 발생했는지 출력해 줍니다.