Compiler

- #Proj2 Syntax Analyzer –

과목명	컴파일러
제출일	2022.06.21
학 번	20175371
학 과	컴퓨터공학과
이 름	강명석

목차

Ambiguos Grammer

기존 제시된 CFG 문법에서 2가지의 모호한 문법을 찾을 수 있었다. 하나는 연산자 우선순위에 따른 모호함이고, 다른 하나는 Dangling Else이다.

연산자 우선순위

기존 제시된 CFG에서는 다음과 같이 EXPR를 표현한다.

```
EXPR → EXPR addsub EXPR | EXPR multdiv EXPR
EXPR → lparen EXPR rparen | id | num
```

위의 문법대로 Parse Tree를 생성하면, 1+2*3과 같은 input에 대해서 2개의 Parse Tree가 생성된다.

따라서 이를 모호하지 않도록 다음과 같이 변환해 준다.

```
EXPR -> EXPR addsub TERM | TERM
TERM -> TERM multdiv FACTOR | FACTOR
FACTOR -> lparen EXPR rparen | id | num
```

Dangling Else

다음의 문법에서 Dangling Else와 같은 문제가 나오는 거라고 판단된다.

```
STMT -> if lparen COND rparen lbrace BLOCK rbrace ELSE
STMT -> while lparen COND rparen lbrace BLOCK rbrace
```

하지만 해당 문법의 모호성을 제거하지는 못해서 본 프로젝트에 사용된 SLR 테이블의 경우 둘 중에 하나의 선택을 하도록 테이블이 생성되었다.

SLR Parsing Table

주어진 CFG 문법을 통해서 http://jsmachines.sourceforge.net/machines/slr.html 사이트에서 파싱 테이블을 생성했다. 생성한 파싱 테이블은 실제 구현에서 다음과 같은 자료구조로 이용했다. (상세 테이블은 table.py를 참조)

```
SLR_table = [
    {'vtype' : 's2','VDECL':'G1'},
    {'vtype' : 's6','class' : 's7','$' : 'r3','CODE':'G3','VDECL':'G1','FDECL':'G4','CDECL':'G5'},
    {'id' : 's8','ASSIGN':'G9'},
    {'$' : 'acc'},
    {'vtype' : 's6','class':'s7','$':'r3','CODE':'G10','VDECL':'G1','FDECL':'G4','CDECL':'G5'},
    {'vtype' : 's6','class':'s7','$':'r3','CODE':'G11','VDECL':'G1','FDECL':'G4','CDECL':'G5'},
    {'id':'s12','ASSIGN':'G9'},
    {'id':'s13'},
    {'semi':'s14','assign':'s15'},
    {'semi':'s16'},
    {'$':'r1'}, #10
    {'$':'r2'},
    {'semi':'s14','assign':'s15','lparen':'s17'},
    {'lbrace':'s18'},
    ...
    // 생략
    ...
    {'vtype':'r28','id':'r28','rbrace':'r28','if':'r28','while':'r28','return':'r28'},
    {'vtype':'r27','id':'r27','rbrace':'r27','if':'r27','while':'r27','return':'r27'}, #80
    {'lbrace':'s82'},

{'vtype':'s2','id':'s54','rbrace':'r24','if':'s52','while':'s53','return':'r24','VDECL':'G50','ASSIGN':
'G51','BLOCK':'G83','STMT':'G49'},
    {'rbrace':'s84'},
    {'vtype':'r31','id':'r31','rbrace':'r31','if':'r31','while':'r31','return':'r31'}
]
```

Implement Syntax Analyzer

Table.py

Lexical Analyzer, Syntax Analyzer에서 사용하기 위한 테이블 4개를 정의하였다.

1. Transition Table : Lexical Analyzer에서 현재 상태에서 들어온 Input을 통해 다음 State를 정의하는 테이블이다.
2. State Table : Lexical Analyzer에서 현재 상태에서 종료된다면, 토큰의 종류가 무엇인지를 담고 있는 테이블이다.
3. SLR Table : Syntax Analyzer에서 사용하는 SLR 파싱 테이블이다. 현재 state에서 next token을 통해 다음 state를 정의하는 테이블이다.
4. Reduce Table : Syntax Analyzer에서 SLR 파싱 테이블을 통해서 Reduce를 수행하기 위해 CFG Production 정보를 담은 테이블이다.

Lex.py

이전 프로젝트에서 수행한 Lexical Analyzer 파일이다. 단독으로 실행하여 Lexical Analyzer 기능만을 수행할 수도 있고, Syntax Analyzer에서는 내부적으로 호출하여 파일을 읽고 Token Stream으로

변환하여 Syntax Analyzer의 Input으로 들어간다.

Syntax.py

Syntax Analyzer는 내부적으로 lex.py의 lexical()함수를 호출하여 input 파일을 토큰화 시킨 결과를 이용한다. 이때, 이전 프로젝트에서 진행했던 토큰들의 이름과 Syntax Analyzer의 CFG에서 받아들이는 터미널들의 Token 이름이 다르기 때문에 이들을 변환하는 작업을 먼저 수행한다.

```
def convert_token_name(token_list) -> list:
    convert_token_list = []
    convert_table = {
        'Type' : 'vtype',
        'Integer' : 'num',
        'Char' : 'character',
        'Bool' : 'boolstr',
        'String' : 'literal',
        'Id' : 'id',
        'Assign' : 'assign',
        'Compare' : 'comp',
        'SCOLON' : 'semi',
        'COMMA' : 'comma',
        'LPAREN' : 'lparen',
        'RPAREN' : 'rparen',
        'LBRACE' : 'lbrace',
        'RBRACE' : 'rbrace',
        'LBRAKET' : 'lbraket',
        'RBRAKET' : 'rbraket'
    }
```

토큰들을 Input으로 받아 수행되는 메인 로직은 다음과 같다.

```
    while(True):
        try:
            next_input_symbol = token_list[0]
            current_state = state_stack[-1]
            next_step = SLR_table[current_state][next_input_symbol]

            if(next_step[0] == 's'):
                # shift and goto
                state_stack.append(int(next_step[1:]))
                token_list.pop(0)

            elif(next_step[0] == 'r'):
                # Reduce
                # RHS 갯수만큼 스택 pop
                reduce_table_idx = int(next_step[1:])
                reduce_LHS = Reduce_table[reduce_table_idx]['LHS']
                for i in range(len(reduce_RHS)):
                    state_stack.pop()

                # GOTO(current state,LHS) into the stack
                current_state = int(state_stack[-1])
                next_GOTO = reduce_LHS
                next_step = SLR_table[current_state][next_GOTO]

                state_stack.append(int(next_step[1:]))

            if(next_step == 'acc'):
                print("success")
                break
```

구현은 스택을 이용하여 state를 저장하고, SLR 테이블에 따라서 Shift and go 인 경우, Reduce인

경우에 대해서 처리를 해주었다.

0. 매 반복문 시작 시

```
next_input_symbol = token_list[0]
current_state = state_stack[-1]
next_step = SLR_table[current_state][next_input_symbol]
```

토큰 리스트에서 그 다음 토큰을 가져온다. 현재 상태정보(current_state)와 다음 토큰 정보를 통해서 SLR_table을 이용하여 그 다음 action/goto 정보를 받아온다. SLR_table에서 shift and go로 이동해야 하는 경우 s5와 같은 정보가 리턴되고, Reduce 해야 하는 경우는 r5와 같이 리턴된다.

1. Shift and go

```
if(next_step[0] == 's'):
    # shift and goto
    state_stack.append(int(next_step[1:]))
    token_list.pop(0)
```

Next_step에는 s17과 같은 SLR 파싱테이블의 정보가 들어있다. 파싱된 결과를 통해서 다음 state를 스택에 push 하고, 토큰 리스트에서 가장 앞에있는 원소를 하나 pop 함으로써 shift 해준다.

2. Reduce

```
elif(next_step[0] == 'r'):
    # Reduce
    # RHS 갯수만큼 스택 pop
    reduce_table_idx = int(next_step[1:])
    reduce_LHS = Reduce_table[reduce_table_idx]['LHS']
    for i in range(len(reduce_RHS)):
        state_stack.pop()

    # GOTO(current state,LHS) into the stack
    current_state = int(state_stack[-1])
    next_GOTO = reduce_LHS
    next_step = SLR_table[current_state][next_GOTO]

    state_stack.append(int(next_step[1:]))
```

Next_step 에는 r20과 같은 SLR 파싱테이블의 정보가 들어있다. reduce_table을 통해서 해당 Reduce 정보(ex : {'LHS' : 'VDECL' , 'RHS' : ['vtype','id','semi']})를 불러온 뒤, RHS 개수만큼 스택에서 state를 pop 한다. 그 다음 LHS 정보를 통해서 SLR 파싱 테이블에서 GOTO 정보를 가져온 뒤, state stack에 push를 한다.

3. State == acc

```
if(next_step == 'acc'):
    print("success")
    break
```

이렇게 반복문을 수행하다가 Reduce 결과가 acc(처음 start symbol)로 돌아온다면, syntax analyzer는 success를 출력하고 종료한다.

Usage

Lexical Analyzer

```
$ python3 lex.py [FILE]
```

[첨부한 예제 소스코드]

```
kms00129@ldap:~/Desktop/syntax_analyzer$ ls lex_code/
error1.txt  error3.txt  error5.txt  error7.txt  test2.txt
error2.txt  error4.txt  error6.txt  test1.txt   test3.txt
```

Lex.py의 예제 코드는 lex_code/ 경로 아래에 존재한다.

[정상 코드 실행]

```
kms00129@ldap:~/Desktop/syntax_analyzer$ python3 lex.py test1.txt
[!] Lexical analyzer for tokenizing simple java code.

[+] Finish Lexical analyzr
[+] Save Result in [lex] test1.txt_output.txt
```

콘솔 정보를 출력하고 [lex] filename_output.txt 로 tokenizing 정보를 저장.

[오류 코드 실행]

```
kms00129@ldap:~/Desktop/syntax_analyzer$ python3 lex.py error1.txt
[!] Lexical analyzer for tokenizing simple java code.

Line[3] Integer Start With 0 is not permitted.
Lexeme : 022
```

몇 번째 라인에서 어떠한 오류가 발생했는지 출력

Syntax Analyzer

```
$ python3 syntax.py [FILE] [--debug]
```

[첨부한 예제 소스코드]

```
kms00129@ldap:~/Desktop/syntax_analyzer$ ls syntax_code/
error1.txt  error3.txt  error5.txt  test2.txt
error2.txt  error4.txt  test1.txt   test3.txt
```

Syntax.py의 예제 코드는 syntax_code/ 경로 아래에 존재한다.

[정상 코드 실행]

```
kms00129@ldap:~/Desktop/syntax_analyzer$ python3 syntax.py test1.txt
[!] Syntax analyzer for tokenizing simple java code.

success
```

정상적으로 파싱이 끝나면 success를 출력

[오류 코드 실행]

```
kms00129@ldap:~/Desktop/syntax_analyzer$ python3 syntax.py error2.txt
[!] Syntax analyzer for tokenizing simple java code.

Error Occur at line number 3

1 : int x = 0;
2 : char y = '2';
3 : int z = 4;
4 : boolean func(int x, int y int z){

current Stack for syntax analyzer : ['VDECL', 'VDECL', 'VDECL', 'vtype', 'id', '
lparen', 'vtype', 'id', 'comma', 'vtype', 'id']
next token : {'token': 'Type', 'lexeme': 'int', 'line_num': 4}
```

오류가 발생하면 전체 소스코드 중, 오류가 발생한 라인과 어떤 토큰을 읽을 때 오류가 발생했는지 출력한다.

Debug Stack Trace

추가적으로 파싱에 성공한 경우에 대해서 디버그 기능을 추가하였다. 코드를 실행할 때 –debug 옵션을 지정하여 다음과 같이 Stack 정보와 Reduce가 어떻게 되는지 확인 가능하다.

Test Code

정상적인 문법을 가진 예제 코드는 3개를 작성하였으며, Error 코드는 4개를 작성하였다.

정상 코드

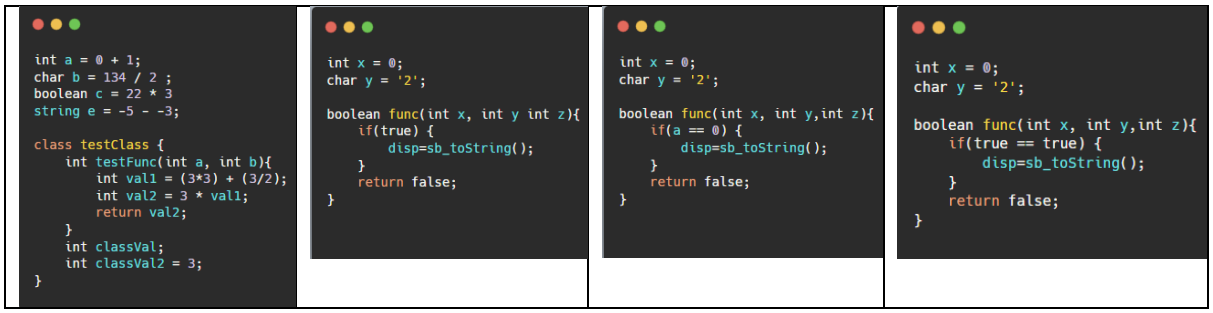
```
int a = 0 + 1;
char b = 134 / 2 ;
boolean c = 22 * 3;
string e = -5 - -3;

class testClass {
    int testFunc(int a, int b){
        int val1 = (3*3) + (3/2);
        int val2 = 3 * val1;
        return val2;
    }
    int classVal;
    int classVal2 = 3;
}
```

```
int x = 0;
char y = '2';
int z = 4;

boolean func(int x, int y, int z){
    if(true) {
        disp=sb_toString;
        if(true == true){
            }
        else{
            if(false){}
        }
    }else {
        disp=sb_toString;
    }
    if(false){
        result= (3 + 3) * 4;
    }else{
        if(true == true){
            result = result - 3;
        }
        if(true != false){
            result = true;
        }
    }
    return false;
}
```

```
int x = 3 + 3 * 4;

int y = 3 * 4 + 3;
```

가장 왼쪽은 Class 선언에 대한 코드, 가운데는 Function 선언, 변수 선언에 대한 테스트 코드, 마지막은 Ambiguous를 제거한 연산자 관련 테스트 코드이다. 이를 디버깅 기능을 이용하여 스택을 분석해 보면 우선순위를 고려하여 Parse Tree가 형성됨을 확인할 수 있다.

Error Case



```
int a = 0 + 1;
char b = 134 / 2 ;
boolean c = 22 * 3
string e = -5 - -3;

class testClass {
    int testFunc(int a, int b){
        int val1 = (3*3) + (3/2);
        int val2 = 3 * val1;
        return val2;
    }
    int classVal;
    int classVal2 = 3;
}
```

```
int x = 0;
char y = '2';

boolean func(int x, int y int z){
    if(true) {
        disp=sb_toString();
    }
    return false;
}
```

```
int x = 0;
char y = '2';

boolean func(int x, int y,int z){
    if(a == 0) {
        disp=sb_toString();
    }
    return false;
}
```

```
int x = 0;
char y = '2';

boolean func(int x, int y,int z){
    if(true == true) {
        disp=sb_toString();
    }
    return false;
}
```

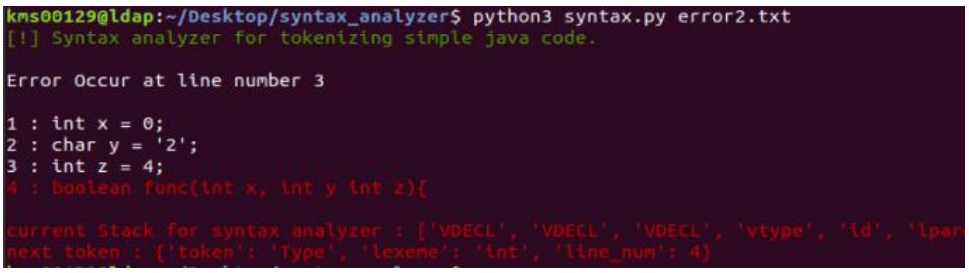
1) 첫 번째 에러 케이스는 세미콜론이 누락되었으며, 다음과 같이 에러 정보가 출력된다.

```
kms00129@ldap:~/Desktop/syntax_analyzer$ python3 syntax.py error1.txt
[!] Syntax analyzer for tokenizing simple java code.

Error Occur at line number 3

1 : int a = 0 + 1;
2 : char b = 134 / 2 ;
3 : boolean c = 22 * 3
4 : string e = -5 - -3;

current Stack for syntax analyzer : ['VDECL', 'VDECL', 'vtype', 'id', 'assign', 'TERM', 'multdiv', 'num']
next token : {'token': 'Type', 'lexeme': 'string', 'line_num': 4}
```

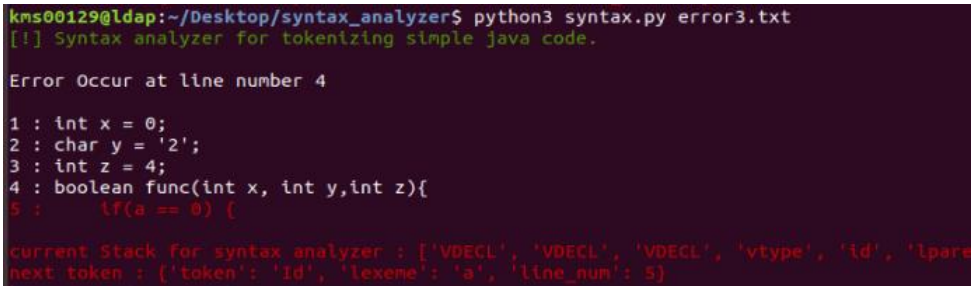
토큰의 위치를 기준으로 라인 수를 계산했기 때문에 세미콜론이 누락되면 다음 string을 읽을

때 오류가 발생하여 4번째 라인에서 오류가 발생했다고 출력한다.

2) 두 번째 에러는 매개변수 입력 부분에서 콤마(,)가 누락되었다.



3) 세 번째 에러는 if 문 내에서 조건(condition)에 대한 내용이다. 원래 java 문법이라면 해당 구문이 문제가 없겠지만, 우리의 프로젝트 CFG 문법에서는 boolstr comp boolstr만 가능하기 때문에 위의 내용은 문법상 오류가 난다.



4) 마지막은 함수 호출에 대한 에러이다. 이 예제 코드 역시 원래 java 문법에서는 가능하지만, 현재 CFG에서는 함수 호출에 대한 문법이 없기 때문에 오류가 난다.