

# Compiler Term Project

20190927 곽태환, 20191127 김두홍

## 목차

1. context-free grammar의 ambiguity 제거하기
2. non-ambiguous CFG를 이용하여 SLR 파싱 테이블 만들기
3. Linux에서 실행 가능하도록 파일 이름으로 파일 내용 저장하기
4. (1)에서 만든 non-ambiguous CFG를 문자열로 저장하기
5. (2)에서 만든 SLR 파싱 테이블을 딕셔너리 리스트로 저장하기
6. 메인 코드에서 사용할 Stack 클래스 만들기
7. (3)에서 저장한 input을 파싱하여 Accept, Reject 구분하기 - 메인 코드
8. input이 Accept인 경우, parse tree 출력하기
9. input이 Reject인 경우, error 출력하기
10. 예시 input으로 코드 실행하기

## 1. Context-free Grammar의 Ambiguity 제거하기

주어진 cfg 중에서 ambiguous한 cfg는 2개가 존재한다.

```
EXPR -> EXPR addsub EXPR | EXPR multdiv EXPR
      | lparen EXPR rparen | id | num
COND -> COND comp COND | boolstr
```

여기서 EXPR의 첫 번째, 두 번째 rule과 COND의 첫 번째 rule이 바로 ambiguity를 발생시킨다. EXPR의 경우, COND와 다르게 addsub와 multdiv 간의 우선순위가 존재하므로 ambiguity를 제거할 때 고려해줘야 한다.

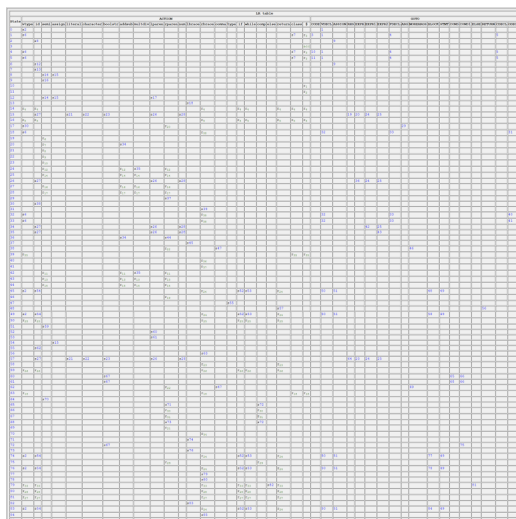
```
EXPR -> EXPR addsub EXPR1 | EXPR1
EXPR1 -> EXPR1 multdiv EXPR2 | EXPR2
EXPR2 -> lparen EXPR rparen | id | num
COND -> COND comp COND1 | COND1
COND1 -> boolstr
```

다음과 같이 rule을 수정해주면 의미는 동일하면서 ambiguity가 제거된 CFG가 만들어진다. 이제 이 non-ambiguous CFG를 이용하여 파싱 테이블을 만들어보자.

## 2. Non-ambiguous CFG를 이용하여 SLR 파싱 테이블 만들기

```
CODE -> VDECL CODE
CODE -> FDECL CODE
CODE -> CDECL CODE
CODE -> ''
VDECL -> vtype id semi
VDECL -> vtype ASSIGN semi
ASSIGN -> id assign RHS
RHS -> EXPR
RHS -> literal
RHS -> character
RHS -> boolstr
EXPR -> EXPR addsub EXPR1
EXPR -> EXPR1
EXPR1 -> EXPR1 multdiv EXPR2
EXPR1 -> EXPR2
EXPR2 -> lparen EXPR rparen
EXPR2 -> id
EXPR2 -> num
FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
ARG -> vtype id MOREARGS
ARG -> ''
MOREARGS -> comma type id MOREARGS
MOREARGS -> ''
BLOCK -> STMT BLOCK
BLOCK -> ''
STMT -> VDECL
STMT -> ASSIGN semi
STMT -> if lparen COND rparen lbrace BLOCK rbrace ELSE
STMT -> while lparen COND rparen lbrace BLOCK rbrace
COND -> COND comp COND1
COND -> COND1
COND1 -> boolstr
ELSE -> else lbrace BLOCK rbrace
ELSE -> ''
RETURN -> return RHS semi
CDECL -> class id lbrace ODECL rbrace
ODECL -> VDECL ODECL
ODECL -> FDECL ODECL
ODECL -> ''
```

위의 non-ambiguous CFG를 SLR Parser Generator (sourceforge.net)에 입력하여 SLR 파싱 테이블을 만든 결과, State0 ~ State85가 존재하는 파싱 테이블이 완성되었다. 해당 값들은 직접 코드에 입력해준다.



### 3. Linux에서 실행 가능하도록 파일 이름으로 파일 내용 저장하기

```
# LINUX에서 입력받은 파일이름으로 파일 내용을 input에 저장
# "python syntax_analyzer.py input.txt"로 실행 가능
input_file = sys.argv[1]
with open(input_file, 'r') as file:
    input = file.read()

# input을 tokenize하여 리스트에 저장
# input을 사용하면 외부 파일에서 input sequence을 받아와 파싱 가능
# input1, input2를 사용하면 위에 명시된 input sequence를 파싱 가능
tokens = input.split(' ')
tokens.append('$')
```

python syntax\_analyzer.py <파일 이름>으로 실행할 때, 파일 이름을 input\_file에 저장한다. 그리고 syntax\_analyzer.py와 같은 폴더에 존재하는 해당 파일을 읽기 모드로 열어 파일 내용 전체를 input에 저장한다. input은 이후 사용이 용이하도록 토큰화시키고 마지막에 '\$'를 추가한다.

### 4. (1)에서 만든 non-ambiguous CFG를 문자열로 저장하기

```
# SLR Parsing에 이용되는 non-ambiguous한 grammars
# 첫번째 grammar는 0번, 마지막 grammar는 38번 -> 총 39개의 grammar
slr_grammars = """CODE -> VDECL CODE
CODE -> FDECL CODE
CODE -> CDECL CODE
CODE -> epsilon
VDECL -> vtype id semi
VDECL -> vtype ASSIGN semi
ASSIGN -> id assign RHS
RHS -> EXPR
RHS -> literal
RHS -> character
RHS -> boolstr
EXPR -> EXPR addsub EXPR1
EXPR -> EXPR1
EXPR1 -> EXPR1 multdiv EXPR2
EXPR1 -> EXPR2
EXPR2 -> lparen EXPR rparen
EXPR2 -> id
EXPR2 -> num
FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
ARG -> vtype id MOREARGS
ARG -> epsilon
MOREARGS -> comma type id MOREARGS
MOREARGS -> epsilon
BLOCK -> STMT BLOCK
BLOCK -> epsilon
STMT -> VDECL
STMT -> ASSIGN semi
STMT -> if lparen COND rparen lbrace BLOCK rbrace ELSE
STMT -> while lparen COND rparen lbrace BLOCK rbrace
COND -> COND comp COND1
COND -> COND1
COND1 -> boolstr
ELSE -> else lbrace BLOCK rbrace
ELSE -> epsilon
RETURN -> return RHS semi
CDECL -> class id lbrace ODECL rbrace
ODECL -> VDECL ODECL
ODECL -> FDECL ODECL
```

```
ODECL -> epsilon""

# 39개의 grammar가 들어있는 리스트 생성
grammars = slr_grammars.split('\n')
```

39개의 CFG를 한 문자열로 slr\_grammars에 저장하고, 줄바꿈을 기준으로 split하여 리스트 grammars에 저장한다. 그리고 epsilon rule의 경우, "로 표현되는 것보다 epsilon으로 표현되는 것이 편하므로 epsilon으로 표현해준다.

## 5. (2)에서 만든 SLR 파싱 테이블을 딕셔너리 리스트로 저장하기

(2)에서 캡처파일을 첨부했듯이 총 86개의 State로 이루어진 파싱 테이블이 만들어진다. 이는 딕셔너리를 원소로 가지는 리스트로 표현해준다.

```
# table 각 열의 이름 = key
keys = ['vtype', 'id', 'assign', 'literal', 'character', 'boolstr',
        'addsub', 'multdiv', 'lparen', 'rparen', 'num', 'lbrace',
        'rbrace', 'comma', 'type', 'if', 'while', 'comp', 'else',
        'return', 'class', '$', 'CODE', 'VDECL', 'ASSIGN', 'RHS',
        'EXPR', 'EXPR1', 'EXPR2', 'FDECL', 'ARG', 'MOREARGS', 'BLOCK',
        'STMT', 'COND', 'COND1', 'ELSE', 'RETURN', 'CDECL', 'ODECL']

# table의 행의 index는 state를 표현
# table의 모든 값을 None으로 초기화
# table에서 값이 존재하는 경우에만 업데이트
terms = {}
for key in keys: terms[key] = None
table = [terms.copy() for _ in range(86)]
```

리스트 keys에는 파싱 테이블 각 열의 이름을 저장하고, None으로 value 값이 초기화된 딕셔너리를 만들어 각 state별로 딕셔너리를 가지게 만든다.

```
table[0].update({'vtype' : 's2', 'VDECL' : 1})
table[1].update({'vtype' : 's6', 'class' : 's7', '$' : 'r3', 'CODE' : 3, 'VDECL' : 1, 'FDECL' : 4, 'CDECL' : 5})
table[2].update({'id' : 's8', 'ASSIGN' : 9})
table[3].update({'$' : 'acc'})
table[4].update({'vtype' : 's6', 'class' : 's7', '$' : 'r3', 'CODE' : 10, 'VDECL' : 1, 'FDECL' : 4, 'CDECL' : 5})
table[5].update({'vtype' : 's6', 'class' : 's7', '$' : 'r3', 'CODE' : 11, 'VDECL' : 1, 'FDECL' : 4, 'CDECL' : 5})
table[6].update({'id' : 's12', 'ASSIGN' : 9})
table[7].update({'id' : 's13'})
table[8].update({'semi' : 's14', 'assign' : 's15'})
table[9].update({'semi' : 's16'})
table[10].update({'$' : 'r1'})
...
table[85].update({'vtype' : 'r32', 'id' : 'r32', 'rbrace' : 'r32', 'if' : 'r32', 'while' : 'r32', 'return' : 'r32'})
```

파싱 테이블에서 값이 존재하는 셀만 업데이트한다. 값이 존재하지 않는 경우 value가 None이고, 파싱 중에 None을 가리키게 되면 해당 input은 Reject된다.

## 6. 메인 코드에서 사용할 Stack 클래스 만들기

```
class Stack():
    # stack 생성 함수
```

```

def __init__(self):
    self.stack = []
# stack에 data를 쌓는 함수
def push(self, data):
    self.stack.append(data)
# stack 가장 위에 있는 data를 추출하는 함수
def pop(self):
    pop_object = None
    if self.isEmpty():
        print("Stack is Empty")
    else:
        pop_object = self.stack.pop()
    return pop_object
# stack 가장 위에 있는 data를 반환하는 함수
def top(self):
    top_object = None
    if self.isEmpty():
        print("Stack is Empty")
    else:
        top_object = self.stack[-1]
    return top_object
# stack이 비어있는지 반환하는 함수
def isEmpty(self):
    is_empty = False
    if len(self.stack) == 0:
        is_empty = True
    return is_empty
# stack의 사이즈를 반환하는 함수
def size(self):
    return len(self.stack)
# stack의 내용을 출력하는 함수
def printStack(self):
    for item in self.stack:
        print(item, end=' ')
    print()

```

Stack 클래스는 push()를 이용하여 어떤 데이터를 stack에 넣거나, pop()을 이용하여 stack 가장 위에 있는 데이터를 추출하거나, top()을 이용하여 stack 가장 위에 있는 데이터를 반환할 수 있다. size()를 이용하면 stack의 크기를 반환할 수 있고, printStack()을 사용하면 stack에 들어있는 데이터를 출력할 수 있다.

## 7. (3)에서 저장한 input을 파싱하여 Accept, Reject 구분하기

```

s1 = Stack()    # state 정보를 나타내는 state Stack
s2 = Stack()    # splitter의 왼쪽 symbol을 나타내는 node Stack
s1.push(0)      # 시작 state는 0이므로, 0을 s1에 push

```

우선, state 정보를 나타내는 state stack과 splitter의 왼쪽의 각 symbol을 나타내는 node stack을 생성하고, start state는 0이므로, s1에 0을 push한다.

```

cnt = 0        # input 내에서 splitter의 위치
accept = True   # 최종적으로 accept=True이면 Accept, accept=False이면 Reject
error = False   # error=True이면 코드 수정 필요
root = Node("CODE") # parse tree의 root node 생성

```

cnt는 input 내에서 splitter의 위치로, cnt가 증가하면서 splitter는 오른쪽으로 이동한다. accept는 이 input이 Accept될 지 구분하는 변수이며, error는 코드에 오류가 있는지 구분하는 변수이다. 또한, start symbol인 "CODE"를 root node로 생성하여 root에 저장해둔다.

```

# Main Parsing Code
while True:
    # Reject
    if table[s1.top()][tokens[cnt]] == None:
        accept = False
        break
    # Shift
    elif table[s1.top()][tokens[cnt]][0] == "s":
        new_node = Node(tokens[cnt])
        s1.push(int(table[s1.top()][tokens[cnt]][1:]))
        s2.push(new_node)
        cnt += 1
    # Reduce & Goto
    elif table[s1.top()][tokens[cnt]][0] == "r":
        index = int(table[s1.top()][tokens[cnt]][1:])
        temp = grammars[index].split(" -> ")
        left = temp[0]; right = temp[1].split()
        new_node = Node(left)
        child = []
        # epsilon를 reduce하는 경우, epsilon 노드 추가
        if right == ["epsilon"]:
            e_node = Node("ε", parent=new_node)
        # epsilon이 아닌 경우, s2에서 노드를 추출
        else:
            for _ in range(len(right)):
                s1.pop()
                child.append(s2.pop())
        # pop()으로 노드를 추출했으므로 순서가 역순
        # 노드를 parent와 연결할 때, 원래 순서대로 바꾸어 연결
        for i in range(len(child)-1, -1, -1):
            child[i].parent = new_node
        s2.push(new_node)
        s1.push(int(table[s1.top()][s2.top().name]))
    # Accept
    elif table[s1.top()][tokens[cnt]] == "acc":
        child = []
        for _ in range(s2.size()): child.append(s2.pop())
        # pop()으로 노드를 추출했으므로 순서가 역순
        # 노드를 parent와 연결할 때, 원래 순서대로 바꾸어 연결
        for i in range(len(child)-1, -1, -1):
            child[i].parent = root
        break
    # Error
    # None, s, r, acc 외의 값이 테이블에 존재하면 코드 수정이 필요
    else:
        accept = False
        error = True
        break

```

위의 코드는 메인 파싱 코드로서, shift, reduce, goto를 반복하다가 조건을 만족하면 Accept, 조건이 만족되지 못하면 Reject, 오류가 발생하면 Error로 분류되어 반복문을 탈출한다.

테이블값이 None이면, 인덱싱이 불가하므로 먼저 분류해준다.

테이블값이 “s숫자”이면, splitter를 이동시켜야 하므로 cnt를 1 증가시키고, s1에 “s숫자”의 숫자 부분을 push, splitter가 이동하며 splitter 왼쪽에 있게 된 symbol을 노드로 만들어 s2에 push한다.

테이블값이 “r숫자”이면, “r숫자”의 숫자 부분에 해당하는 production rule을 받아와서 화살표를 기준으로 왼쪽과 오른쪽으로 split한다. 이 때, epsilon이 reduce되는 경우에는 따로 처리를 해주고, s1과 s2에서 rule 오른쪽의 symbol의 개수 만큼 pop()을 해주고, s2에서 pop()된 symbol node들은 rule 왼쪽의 symbol로 symbol 노드를 만들어 parent로 연결해준다. 또한, reduce 이후에는 goto를 해야하므로, s1.top()과 s2.top().name 값을 이용하여 s1 가장 위에 해당 값을 추가해준다.

테이블값이 “acc”이면, s2에 있는 노드들이 아까 전에 만든 root node로 연결되는 것이므로, 모두 pop()하여 parent로 root를 연결시켜준다.

만약 테이블값이 이외의 값이라면, 테이블을 잘못 만들었을 확률이 높으니 코드를 확인해볼 필요가 있다.

## 8. Accept인 경우, parse tree 출력하기

```
# tree 출력
if accept == True:
    print("Accept!")
    for pre, _, node in RenderTree(root):
        print(f"{pre}{node.name}")
```

아까 테이블값이 "acc"이면 반복문을 그대로 탈출해서 accept는 초기값인 True일 것이다. 이 경우, "Accept!"를 출력하고, 이전에 만들었던 노드와 노드 사이 관계를 이용하여 parse tree를 출력해준다.

## 9. Reject인 경우, error 출력하기

```
# reject된 이유 출력
elif error == False:
    print("Reject!")
    print(f"There is no value in table[{s1.top()}][{s2.top().name}]")
else:
    print("Error!")
    print("You need to edit your code")
```

테이블값이 None이면 반복문을 탈출할 때, accept 값이 False로 바뀌고 반복문을 탈출했기 때문에 accept=False, error=False일 것이다. 이 경우, "Reject!"를 출력하고, 어디에서 reject 되었는지 출력해준다.

테이블값이 정해진 값들을 벗어날 경우, 코드를 수정해줄 필요가 있으므로 "Error!"를 출력하고, 코드를 수정하라는 경고문을 출력해준다.

## 10. 예시 input으로 코드 실행

```
# 과제에 제시된 input sequence, 결과 : Reject
input1 = "vtype id semi vtype id lparen rparen lbrace if lparen boolstr comp boolstr rparen lbrace rbrace"
# 직접 만든 input sequence, 결과 : Accept
input2 = "vtype id semi vtype id lparen vtype id comma type id rparen lbrace return literal semi rbrace"
```

위의 input들을 입력하여 .py 파일과 같은 폴더에 .txt로 파일을 생성했다. syntax\_analyzer.py와 input1.txt와 input2.txt는 같은 폴더에 위치한다.

```
python syntax_analyzer.py input1.txt
python syntax_analyzer.py input2.txt
```

powershell에서 저장해둔 파일들을 이용하여 코드를 실행시켰다.

```

PS D:\OneDrive\3학년 1학기 강의자료\3. 컴파일러\과제> python syntax_analyzer.py input1.txt
Reject!
There is no value in table[79][rbrace]
PS D:\OneDrive\3학년 1학기 강의자료\3. 컴파일러\과제> python syntax_analyzer.py input2.txt
Accept!
CODE
├── VDECL
│   ├── vtype
│   ├── id
│   └── semi
├── CODE
│   ├── FDECL
│   │   ├── vtype
│   │   ├── id
│   │   ├── lparen
│   │   ├── ARG
│   │   │   ├── vtype
│   │   │   ├── id
│   │   │   └── MOREARGS
│   │   │       ├── comma
│   │   │       ├── type
│   │   │       ├── id
│   │   │       └── MOREARGS
│   │   │           └── ε
│   │   ├── rparen
│   │   ├── lbrace
│   │   ├── BLOCK
│   │   │   └── ε
│   │   ├── RETURN
│   │   │   ├── return
│   │   │   ├── RHS
│   │   │   │   └── literal
│   │   │   ├── semi
│   │   └── rbrace
│   └── CODE
│       └── ε

```

input1의 경우, reject되어 어디서 reject 되었는지 출력되었고, input2의 경우, accept되어 parse tree가 출력되었다.