Compiler Term Project - 2

Team 44. 20181594 김형기, 20180500 장훈석

Index

1. Non-ambiguous CFG G

프로젝트에서 주어진 CFG G는 not ambiguous하고 non-left recursive한 CFG이기 때문에 CFG를 수정하진 않고 bottom-up parser를 만들기 위해
dummy start S'을 추가하였습니다.

```
S' -> CODE
CODE -> VDECL CODE
CODE -> FDECL CODE
CODE -> ε
VDECL -> vtype id semi
FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
ARG -> vtype id MOREARGS
ARG -> ε
MOREARGS -> comma vtype id MOREARGS
MOREARGS -> ε
BLOCK -> STMT BLOCK
BLOCK -> ε
STMT -> VDECL
STMT -> id assign RHS semi
STMT -> if lparen COND rparen lbrace BLOCK rbrace else lbrace BLOCK rbrace
STMT -> while lparen COND rparen lbrace BLOCK rbrace
RHS -> EXPR
RHS -> literal
EXPR -> TERM addsub EXPR
EXPR -> TERM
TERM -> FACTOR multdiv TERM
TERM -> FACTOR
FACTOR -> lparen EXPR rparen
FACTOR -> id
FACTOR -> num
```

```
COND -> FACTOR comp FACTOR
RETURN -> return FACTOR semi
```

위의 CFG를 *SLR Parser Generator(Link)* 에 돌려서 SLR Table을 얻었습니다.

2. SLR Parsing Table

얻은 SLR Table은 아래와 같습니다.

먼저 **ACTION Table**입니다.

SR Parsing Table.<Action>

State	SLR TABLE ACTION																			
	ε	vtype	id	semi	lparen	rparen	lbrace	rbrace	comma	assign	if	else	while	literal	addsub	multdiv	Num	somp	return	$
0	s4	s5																		
1																				acc
2	s4	s5																		
3	s4	s5																		
4																				r3
5			s8																	
6																				r1
7																				r2
8				s9	s10															
9	r4	r4	r4								r4		r4							
10	s13	s12																		
11						s14														
12			s15																	
13						r7														
14							s16													
15	s19								s18											
16	s22	s27	s24								s25		s26							
17						r6														
18		s28																		
19						r9														
20																			s30	
21	s22	s27	s24								s25		s26							
22								r11											r11	
23	r12	r12	r12								r12		r12							
24										s32										
25					s33															
26					s34															
27			s35																	
28			s36																	
29								s37												
30			s40		s39												s41			
31								r10											r10	
32			s40		s39									s44			s41			
33			s40		s39												s41			
34			s40		s39												s41			
35				s9																
36	s19								s18											
37	r5	r5																		
38				s51																
39			s40		s39												s41			
40				r23		r23									r23	r23		r23		
41				r24		r24									r24	r24		r24		
42				s53																
43				r16																
44				r17																
45				r19		r19									s54					
46				r21		r21									r21	s55				
47						s56														

48																		s57		
49						s58														
50						r8														
51								r26												
52						s59														
53	r13	r13	r13								r13		r13							
54			s40		s39												s41			
55			s40		s39												s41			
56							s62													
57			s40		s39												s41			
58							s64													
59				r22		r22									r22	r22		r22		
60				r18		r18														
61				r20		r20									r20					
62	s22	s27	s24								s25		s26							
63						r25														
64	s22	s27	s24								s25		s26							
65								s67												
66								s68												
67												s69								
68	r15	r15	r15								r15		r15							
69							s70													
70	s22	s27	s24								s25		s26							
71								s72												
72	r14	r14	r14								r14		r14							

다음으로는 **GOTO Table**입니다.

SR Parsing Table.<GOTO>

SLR TABLE														
State	GOTO													
	S'	CODE	VDECL	FDECL	ARG	MOREARGS	BLOCK	STMT	RHS	EXPR	TERM	FACTOR	COND	RETURN
0		1	2	3										
1														
2		6	2	3										
3		7	2	3										
4														
5														
6														
7														
8														
9														
10					11									
11														
12														
13														
14														
15						17								
16			23				20	21						
17														
18														
19														
20														29
21			23				31	21						
22														
23														
24														
25														
26														
27														
28														
29														
30												38		
31														
32									42	43	45	46		
33												48	47	
34												48	49	
35														
36						50								
37														
38														
39										52	45	46		
40														
41														
42														
43														
44														
45														
46														
47														
48														
49														
50														
51														
52														
53														
54										60	45	46		
55											61	46		
56														
57												63		
58														
59														
60														
61														
62			23				65	21						
63														
64			23				66	21						
65														
66														
67														
68														
69														
70			23				71	21						
71														
72														

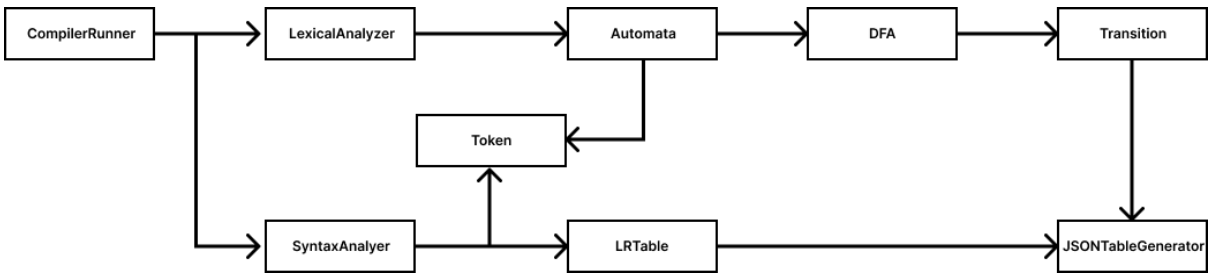
이런식으로 SLR Table을 이용하여 LRTable.json 파일에 옮겨 주었습니다.

```
{
  "0": {
    "e": "s4",
    "vtype": "s5",
    "CODE": "1",
    "VDECL": "2",
```

```
    "FDECL": "3"
  },
  "1": {
    "$": "acc"
  },
  "2": {
    "e": "s4",
    "vtype": "s5",
    "CODE": "6",
    "VDECL": "2",
    "FDECL": "3"
  },
  "3": {
    "e": "s4",
    "vtype": "s5",
    "CODE": "7",
    "VDECL": "2",
    "FDECL": "3"
  },
  "4": {
    "$": "r3"
  },
  "5": {
    "id": "s8"
  },
....생략

  "70": {
    "e": "s22",
    "vtype": "s27",
    "id": "s24",
    "if": "s25",
    "while": "s26",
    "VDECL": "23",
    "BLOCK": "71",
    "STMT": "21"
  },
  "71": {
    "rbrace": "s72"
  },
  "72": {
    "e": "r14",
    "vtype": "r14",
    "id": "r14",
    "if": "r14",
    "while": "r14"
  }
}
```

3. Project Structure



- CompilerRunner : 매개변수로 전달된 컴파일 대상 파일에 대해 프로그램 실행을 담당합니다. I/O 작업에 필요한 객체를 담당합니다.
  - LexicalAnalyzer : 대상 파일에서 다음 글자를 읽어와서 오토마타가 해당하는 상태로 이동하는 것을 담당합니다.
    - Automata : 각 토큰에 해당하는 DFA를 모두 가지고 있으며, Lexical accept/reject의 판단을 담당합니다.
    - DFA : 다음 문자에 대해 해당하는 State로 이동합니다.
    - Transition : DFA가 이동할 다음 State를 JSONObject로부터 반환합니다.
    - Token : LexicalAnalyzer의 결과를 담기 위한 클래스입니다.
  - SyntaxAnalyzer : LexicalAnalyzer에서 생성한 Token의 리스트가 올바른 구문인지 판단합니다.
    - LRTable : 토큰 입력에 대해 다음 State 및 production RHS의 개수 등을 반환합니다.
  - JSONTableGenerator : 문법에 대해 나타낸 .json 파일을 객체로 파싱합니다.

4. Implementation

Token

LexicalAnalyzer에서 정의된 토큰 이름과 SyntaxAnalyzer에서의 Terminal 이름과 다르기 때문에 이를 변환하는 메소드 `toTerminal()`를 추가하였습니다.

```
// Convert to terminal
  public String toTerminal() {

      if (this.tokenName.equals("OP")) {
          // addsub
          if (this.lexeme.equals("+") || this.lexeme.equals("-"))
              return "addsub";
          // multdiv
          if (this.lexeme.equals("*") || this.lexeme.equals("/"))
              return "multdiv";
      }

      // Convert to terminal
      String mapped = mapTerminal.get(this.tokenName);
      if (mapped != null)
          return mapped;

      return this.tokenName.toLowerCase();

  }
```

`OP` 토큰의 경우 lexeme을 이용해서 `addsub` 또는 `multdiv` 로 구분합니다.

이외에 다른 이름을 가지는 토큰은 정의된 정적 상수를 이용해서 매핑합니다.

```
// Map that convert token to terminal
    private final static Map<String, String> mapTerminal = Collections.unmodifiableMap(new HashMap
<String, String>() {
        {
            put("INTEGER", "num");
            put("STRING", "literal");
            put("COMPARISON", "comp");
        }
    });
```

또한 대문자로 표현되었던 토큰을 소문자 형태로 변경하여 CFG의 terminal로 사용하도록 구현하였습니다.

LRTable

SLR parsing table의 정보를 담은 LRTable.json 파일과 Production 정보를 담은 Production.json 파일로부터 불러와서 JSONObject 객체로 메모리에 적재합니다.

```
public static void init() throws FileNotFoundException {
       // json file
        JSONTableGenerator lrTableGenerator = new JSONTableGenerator(FILE_NAME_LR_TABLE);       //
LR Table
        JSONTableGenerator lhsGenerator = new JSONTableGenerator(FILE_NAME_PROD);
 // LHS of productions

        lrTable = lrTableGenerator.getJSONTable();
        productions = lhsGenerator.getJSONTable();
    }
```

SLR parser의 진행 과정에서 현재 state와 다음 input symbol에 대응하는 ACTION, GOTO를 얻기 위한 정적 메소드를 제공합니다. 이는 LRTable.json으로부터 파싱한 JSONObject로부터 얻을 수 있습니다.

```
@Nullable
public static String getAction(final int curState, final String input) {
    JSONObject state = (JSONObject) lrTable.get(""+curState);

    return (String) state.get(input);
}

@NotNull
public static int getGoto(final int curState, final String input) {
    JSONObject state = (JSONObject) lrTable.get(""+curState);
    String res = (String) state.get(input);

    if (isNumber(res))
        return Integer.parseInt(res);
    return -1;
}
```

Reduce 과정에서 스택 pop의 개수를 알기 위해서 Production의 RHS 개수를 반환하는 정적 메소드를 제공합니다. 또한 GOTO 테이블에서 다음 state를 얻기 위해 필요한 LHS의 Non-terminal을 반환하는 정적 메소드도 제공합니다.

```
@Nullable
public static String getLHS(final int prodNum) {
    JSONObject prod = (JSONObject) productions.get(""+prodNum);
    return (String) prod.get("LHS");
}

@Nullable
public static long getNumOfRHS(final int prodNum) {
    JSONObject prod = (JSONObject) productions.get(""+prodNum);
    return (Long) prod.get("RHS");
}
```

SyntaxAnalyzer

현재 구현에서는 사용하지 않았지만, 멀티스레딩 환경에서 여러 파일을 병렬 처리할 수 있도록 Runnable로 구현하였습니다.

- 필드 목록
  - `fileName` : 구문 오류가 발생했을 때, 해당 파일을 콘솔에 표시할 때 사용합니다.
  - `symbolTable` : LexicalAnalyzer 결과로 얻은 Token의 리스트입니다.
  - `stateStack` : SLR parsing 과정에서 사용되는 스택입니다. State 번호로 구성되기 때문에 Integer 타입입니다.
  - `accepted` : SyntaxAnalysis 결과 성공적으로 accept되었는지 결과입니다.

```
private final String fileName;

private final ArrayList<Token> symbolTable;

private final Stack<Integer> stateStack = new Stack<Integer>();

private boolean accepted = true;
```

`splitter` 는 현재 보고 있는 Token의 위치입니다.

처음 시작할 때 dummy start인 0을 push합니다.

reject이 되지 않았으며, `splitter`가 끝에 도달하지 않은 동안 반복합니다.

- `nextSymbol` : 현재 `splitter`가 가리키는 토큰을 SLR parser에서 사용할 terminal로 변환한 것입니다.
- `decision` : 현재 스택의 top과 `nextSymbol`을 이용해서 LRTable로부터 Action을 불러옵니다.

```
@Override
public void run() {
    int splitter = 0;

    // init
    stateStack.push(0);

    while(accepted &&  splitter < symbolTable.size()) {
        boolean epsilonMoved = false;

        // next input symbol
        String nextSymbol = symbolTable.get(splitter).toTerminal();
        String decision = LRTable.getAction(stateStack.peek(), nextSymbol);
```

만약 해당하는 Action이 없다면, epsilon move가 가능한지를 다시 한 번 확인합니다.

만약 epsilon move로 Shift를 할 경우에는 `splitter` 를 이동하지 않기 위해서 `epslionMoved` 에 flag를 기록합니다.

epsilon move도 불가능한 경우에는 reject되고 해당 파일에 대해 구문 분석이 종료됩니다.

```
        // epsilon move
        if (decision == null) {
            decision = LRTable.getAction(stateStack.peek(), "e");
            epsilonMoved = true;
        }

        // invalid transition -> reject
        if (decision == null || decision.length() < 1) {
            Token symbol = symbolTable.get(splitter);
            System.out.println("Error occurred in file " + fileName);
            System.out.println("Syntax error : '" + symbol.getLexeme() +
"' in Line " + symbol.getLineNumber() + "\n");
            accepted = false;
        }
```

만약 LRTable로부터 “acc”라는 결과를 얻었다면 해당 파일에 대해서 accepted를 반환하고, 구문 분석을 종료합니다.

```
        else {

            // accept
            if (decision.equals("acc")) {
                accepted = true;
                break;
            }
```

`decision` 을 `op` 와 `value` 로 분리합니다.

```
            char op = decision.charAt(0);                                                   // s or c
            int value = Integer.parseInt(decision.substring(1));                // [num]
```

shift and goto 일 경우 현재 LRTable로부터 얻은 state인 `value` 를 스택에 push하고, splitter를 다음 토큰으로 이동합니다.

```
            // action : shift and goto
            if (op == 's') {
                stateStack.push(value);   // push the next state into the stack
                if (!epsilonMoved)
                    splitter++;           // move the splitter to the right
            }
```

reduce일 경우 LRTable로부터 해당 Production의 RHS의 개수를 불러와서 스택에서 그 수만큼 pop합니다.

현재 스택의 top과 Production의 LHS를 이용해 GOTO 테이블에서 다음 state를 불러옵니다.

만약 GOTO 테이블에 해당하는 state가 없다면 reject시키고, 있다면 스택에 push 합니다.

```
            // action : reduce
            else if (op == 'r') {

                // pop number of RHS items from stack
                for (int i = 0; i < LRTable.getNumOfRHS(value); i++)
                    stateStack.pop();

                int nextState = LRTable.getGoto(stateStack.peek(), LRTable.getLHS(value));
// goto table
                // rejected
                if (nextState == -1)
                    accepted = false;
                else
                    stateStack.push(nextState);
            }
        }
...
```

5. Test

No Syntax Error Test Case

먼저 Syntax Error가 없이 돌아가는 문법을 test한 경우입니다.

```
int a;
int bd;
```

```
int main(char args, int bd) {
    int result;

    bd = -23;
    a = args + bd / 2;
    if ((bd / 2) == 0) {
        result = -1;
    }
    else {
        result = bd + a;
    }
    while(a!=0){
        a = a - 1;
    }

    return result;
}
char test1() {
    if(a >= 0){
        a = b+c;
    }
    else{
        a = a+c;
    }
    hoon = asdf;
    return hoon;
}

int test2(int aa){
    int a;
    int b;
    int c;
    int gimanki;
    a = aa;
    b = aa * aa;
    c = (aa*aa)/aa;
    gimanki = a + b + c;
    return gimanki;
}
```

이 경우엔 주어진 CFG에 적합한 문법이기 때문에 이 test case를 test해보면

이런식으로 “Successfully accepted for file function_test1.c” 라고 Console에 출력됩니다.

Syntax Error Detect

Syntax Error가 발생한 Test Case들을 알아보겠습니다.

1. if문 안에 return이 있는 경우

```
int isEven(int num) {
    if ((num / 2) == 0) {
        return 1;
```

```
    }
    else {
        return 0;
    }
}
```

이 Test Case의 경우 if 안에 return이 있기 때문에 Syntax Error가 발생하는 Test Case입니다.

이 경우에 저희는

이런식으로 Line 3에서 return이라는 input때문에 Syntax Error가 발생했다는 문구를 Console에 출력하였습니다.

2. 함수 안에 return이 없는 경우.

```
int nonblock(){
    return 1;
}
char block(char a, char b, char c){
    if((a/b)==c) {
        a = b + c;
    }
    else {
        a = b - c;
    }
}
```

이 경우 block 함수에 return이 없기 때문에 Syntax Error가 Detect 됩니다.

함수 블록 마지막에는 return이 등장해야 하는데, '}' input이 인식되었기 때문에 해당 라인 11에서 Syntax error가 발생하였다는 문구를 Console에 출력하였습니다.

3. 변수 선언문에서 초기화를 한 경우

```
int a;
int b;
int c = 0;
```

변수 선언문에서 초기화를 진행하는 경우 Syntax Error가 발생하는 Test Case입니다.

```
java -jar Simple-C-Compiler.jar initialize.c
Error occurred in file initialize.c
Syntax error : '=' in Line 3
```

3번 라인에서 '='으로 assign을 진행하는 부분에서 Syntax erorr가 발생했다는 문구를 Console에 출력하였습니다.

4. if문에 else문이 Match되지 않은 경우.

```
int nonelse(int a, int b){
    if(a != b){
        a = b;
    }
    return a;
}
```

if문만 있고 else문은 없는 경우 Syntax Error가 발생하는 Test Case입니다.

```
~/gi/Simple-C-Compiler | @d43d4fa7 *1 !3 ?4  java -jar Simple-C-Compiler.jar if_nonelse.c
Error occurred in file if_nonelse.c
Syntax error : 'return' in Line 5

Execution is complete for 1 files.%
```

if 문 뒤에는 else문이 등장해야 하는데 'return' input이 인식되었기 때문에 라인 5에서 Syntax Error가 발생하였다는 문구를 Console에 출력하였습니다.

More Develope

현재 구현 상황에서의 한계점과 개선 방법에 대해서 서술하고자 합니다.

위의 Test Case에서 Error가 발생하는 부분 중 2번째 Case인 함수안에 return이 없는 경우를 살펴보겠습니다.

```
int nonblock(){
    return 1;
}
char block(char a, char b, char c){
    if((a/b)==c) {
        a = b + c;
    }
    else {
        a = b - c;
```

```
    }
}
```

이 경우 block 함수에 return이 없기 때문에 Syntax Error가 Detect 됩니다.

함수 블록 마지막에는 return이 등장해야 하는데, '}' input이 인식되었기 때문에 해당 라인 11에서 Syntax error가 발생하였다는 문구를 Console에 출력하였습니다.

이때 Error Detect 문구로 "Syntax error : "}" in Line 11"이라는 문구를 출력합니다.

하지만, 사용자 입장에서 구문 오류의 이유를 알기 어려운 단점이 존재합니다.

함수의 블록이 '}'으로 종료되는 것은 올바르기 때문입니다.

따라서 상용 IDE처럼 Expected : 'return' 같은 문구를 출력한다면 사용자가 구문 오류를 파악하고 고칠 때 용이할 것으로 기대합니다.

만약 이런 기능을 추후에 추가한다면 SLR parsing table에 현재 상태에서의 Follow symbol을 기입하여 사용하는 방법으로 개선할 수 있을 것입니다.