

Report

Team 8. 임동영, 한태균

Changes to the lexical analyzer

'-' 기호의 처리

기존에는 '-' 기호의 위치를 연산자와 음수 둘 중 하나로 결정하는 데에 세 가지 기준점을 두었습니다.

1. 기호 앞에 '=' (w.s) 가 오는 경우
2. 기호 앞에 <ARITHMETIC OPERATER> (w.s) 가 오는 경우
3. 기호 앞에 ';' (w.s) 가 오는 경우

여기에 '('(left paren)과 '{'(left brace)가 오는 경우를 네번째 기준으로 추가 하였습니다.

4. 기호 앞에 '(' (w.s) 혹은 '{' (w.s) 가 오는 경우

```
if (  
    output[idx][0] == "ASSIGNMENT OPERATOR"  
    or output[idx][0] == "ARITHMETIC OPERATOR"  
    or output[idx][0] == "COMMA"  
    or output[idx][0] == "LPAREN"  
    or output[idx][0] == "LBRACE"  
):  
    output.pop()  
    value = "-" + value
```

토큰의 간략화

저희는 토큰을 표기할 때, 꺾쇠를 사용해 표기하고 있었습니다. Syntax analyzer의 자료처리 편의성을 위해 꺾쇠를 제거 하였습니다.

Before	After
<SIGNED INTEGER> 30	SIGNED INTEGER 30

Non-ambiguous CFG G and SLR parsing table

Ambiguity 제거 및 테이블 구현

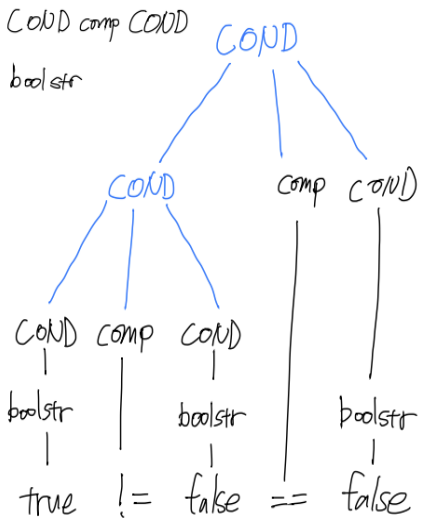
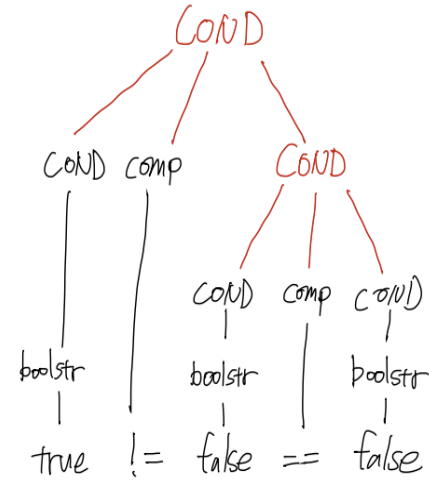
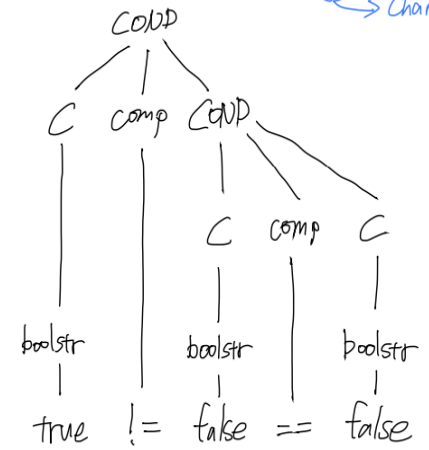
과제 공지대로, 2개의 ambiguity를 제거하기 위해, 먼저 ambiguity가 발생하는 CFG rule을 찾아보았습니다.

그 결과, 다음 2 개의 ambiguity 를 발견했습니다.

1. 덧셈과 곱셈 우선순위 반영되어 있지 않음
2. 조건식의 우선순위 반영되어 있지 않음

이를 고치기 위해, 다음의 과정을 거쳤습니다.

먼저, 덧셈과 곱셈에 대한 우선순위 문제는 강의록을 참고하여 그대로 수정하였습니다. 그리고, 해당 해결법이 꼬리를 무는 식의 CFG rule을 만들어서 해결한다는 점에 착안하여 조건식도 다음과 같이 수정하였습니다.

<p>$COND \rightarrow COND \text{ comp } COND$ $COND \rightarrow \text{boolstr}$</p>  <p>$COND \rightarrow C \text{ comp } COND \mid C$ $C \rightarrow \text{boolstr}$</p> 	<p>$\text{addsub} / \text{multdiv}$ 우선순위 문제는 강의록과 일치하면</p> <p>$EXPR \rightarrow T \text{ addsub } EXPR \mid T$ $T \rightarrow F \text{ multdiv } T \mid F$ $F \rightarrow \text{lparen } EXPR \text{ rparen} \mid \text{id} \mid \text{num}$</p> <p>$COND \rightarrow C \text{ comp } COND \mid C$ $C \rightarrow \text{boolstr}$</p>  <p>필요한 경우에만 COND를 불러서 해결.</p>
---	--

그렇게 하여, 저희 팀의 CFG 는 다음과 같이 작성할 수 있었습니다. S 는 dummy start 입니다.

Non-ambiguous CFG G

```
S -> CODE
CODE -> CDECL CODE
CODE -> FDECL CODE
CODE -> VDECL CODE
CODE -> "
VDECL -> vtype id semi
VDECL -> vtype ASSIGN semi
ASSIGN -> id assign RHS
RHS -> EXPR
RHS -> literal
RHS -> character
RHS -> boolstr
EXPR -> T addsub EXPR
EXPR -> T
T -> F multdiv T
T -> F
F -> lparen EXPR rparen
F -> id
F -> num
FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
ARG -> vtype id MOREARGS
ARG -> "
MOREARGS -> comma vtype 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 -> C comp COND
COND -> C
C -> 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 CFG 를 [SLR Parser Generator](http://jsmachines.sourceforge.net/machines/slr.html)¹에 돌려서 SLR Table 을 얻을 수 있었습니다.

해당 SLR Table 의 HTML 을 [HTML to JSON converter](https://toolslick.com/conversion/data/html-to-json)²를 활용하여 변환하여, JSON 파일을 얻었습니다. 그 후 이 파일을 파서에서 사용하기 쉽게, 약간의 변형을 거쳐서 다음의 형태로 가공하였습니다.

```
syntax_analyzer = {
  0: {
    "vtype": ["s", 6],
    "class": ["s", 5],
    "$": ["r", 4],
    "CODE": 1,
    "VDECL": 4,
    "FDECL": 3,
    "CDECL": 2,
  },
  1: {
    "$": "acc",
  },
  2: {
    "vtype": ["s", 6],
    "class": ["s", 5],
    "$": ["r", 4],
    "CODE": 7,
    "VDECL": 4,
    "FDECL": 3,
    "CDECL": 2,
  },
  3: {
    "vtype": ["s", 6],
    "class": ["s", 5],
    "$": ["r", 4],
    "CODE": 8,
    "VDECL": 4,
    "FDECL": 3,
    "CDECL": 2,
  },
}
```

SLR table 의 state 는 첫 Key 가 되게 했고, state 에 따른 value 에서는 다음 input 값 혹은 GOTO 에 따라서 처리해야할 동작을 정의하였습니다. Shift 해야 할 경우엔 s, Reduce 해야 할 경우엔 r 로 구분하였습니다. 파서는 동작을 수행하면서, 이 파이썬 Dictionary 형태의 테이블을 계속 참조하며 진행하게 됩니다.

JSON 파일은 slr.json, SLR table 은 Syntax_Analyzer/table.py 에서 확인하실 수 있습니다.

¹ SLR Parser Generator <http://jsmachines.sourceforge.net/machines/slr.html>

² HTML to JSON converter <https://toolslick.com/conversion/data/html-to-json>

Reduce 규칙은 다음과 같습니다. CFG G 를 역순으로 배치해 놓았습니다.

```
reduce = {
  0: {"CODE": "S"},
  1: {"CDECL CODE": "CODE"},
  2: {"FDECL CODE": "CODE"},
  3: {"VDECL CODE": "CODE"},
  4: {"": "CODE"},
  5: {"vtype id semi": "VDECL"},
  6: {"vtype ASSIGN semi": "VDECL"},
  7: {"id assign RHS": "ASSIGN"},
  8: {"EXPR": "RHS"},
  9: {"literal": "RHS"},
  10: {"character": "RHS"},
  11: {"boolstr": "RHS"},
  12: {"T addsub EXPR": "EXPR"},
  13: {"T": "EXPR"},
  14: {"F multdiv T": "T"},
  15: {"F": "T"},
  16: {"lparen EXPR rparen": "F"},
  17: {"id": "F"},
  18: {"num": "F"},
  19: {"vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace": "FDECL"},
  20: {"vtype id MOREARGS": "ARG"},
  21: {"": "ARG"},
  22: {"comma vtype id MOREARGS": "MOREARGS"},
  23: {"": "MOREARGS"},
  24: {"STMT BLOCK": "BLOCK"},
  25: {"": "BLOCK"},
  26: {"VDECL": "STMT"},
  27: {"ASSIGN semi": "STMT"},
  28: {"if lparen COND rparen lbrace BLOCK rbrace ELSE": "STMT"},
  29: {"while lparen COND rparen lbrace BLOCK rbrace": "STMT"},
  30: {"C comp COND": "COND"},
  31: {"C": "COND"},
  32: {"boolstr": "C"},
  33: {"else lbrace BLOCK rbrace": "ELSE"},
  34: {"": "ELSE"},
  35: {"return RHS semi": "RETURN"},
  36: {"class id lbrace ODECL rbrace": "CDECL"},
  37: {"VDECL ODECL": "ODECL"},
  38: {"FDECL ODECL": "ODECL"},
  39: {"": "ODECL"},
}
```

All about how syntax analyzer works

Syntax analyzer 동작 방식

파서는 크게 다음과 같이 동작합니다.

1. Lexical analyzer에서의 결과 파일을 받아옵니다.
2. 토큰 이름을 CFG G에 맞게 번역합니다.
3. 번역된 토큰을 input_token 데크에 순서대로 넣습니다.
4. 스택에 시작 위치인 0 을 삽입합니다.
5. 스택에서 현재 위치를 불러오고 SLR table을 참조해 이번 action을 받아옵니다.
 - A. 만약 action이 shift 혹은 reduce라면
 - i. Shift: input_token에서 stack으로 토큰을 하나 shift한 후, 현재 action(next state)을 push합니다.
 - ii. Reduce: 이번 action의 reduce rule을 확인한 뒤, stack에서 알맞은 만큼의 토큰을 pop한 후 reduce된 non-terminal을 push합니다.
 - B. 만약 action이 goto라면 현재 action(next state)를 stack에 push합니다.
 - C. 그렇지 않다면(action을 찾을 수 없다면) reject합니다.
6. Action이 accept가 나올때까지 5. 을 반복합니다.

먼저 각종 테이블들을 받아 왔습니다.

```
# Load reduce, slr and convert tables
reduce = table.reduce
syntax_analyzer = table.syntax_analyzer
convert_token = table.convert_token
input_token = deque()
```

변환을 위한 convert table은 다음과 같이 구성 했습니다. 전에 정의한 토큰 이름을 딕셔너리로 치환할 수 있게 하고, 산술 연산자와 키워드도 따로 처리할 수 있게 하였습니다.

```
# Replace tokens in the lexical analyzer with terms that match the terminal
convert_token = {
    "SINGLE CHARACTER": "character",
    "VARIABLE TYPE": "vtype",
    "SIGNED INTEGER": "num",
    "BOOLEAN STRING": "boolstr",
    "ARITHMETIC OPERATOR": ["addsub", "multdiv"],
    "ASSIGNMENT OPERATOR": "assign",
    "COMPARISON OPERATOR": "comp",
    "TERMINATING SYMBOL": "semi",
    "LPAREN": "lparen",
    "RPAREN": "rparen",
    "LBRACE": "lbrace",
    "RBRACE": "rbrace",
    "COMMA": "comma",
    "LITERAL STRING": "literal",
    "IDENTIFIER": "id",
    "KEYWORD": ["if", "else", "while", "class", "return"],
}
```

다음은 convert_table을 이용해서 input_token덱에 토큰 이름들을 치환해 넣는 과정입니다.

```
F = open(filePath.split(".")[0] + "_lexer_output.txt", "r")
for tokens in f:
    token = tokens.split(" ")
    # print(token)
    line = token.pop(0)
    value = token.pop().strip()
    # Separation of token and value
    while len(token) > 2:
        value = token.pop().strip()
        token = " ".join(token)
    # Handle arithmetic operators and keywords
    if token == "ARITHMETIC OPERATOR":
        if value == "+" or value == "-":
            input_token.append((int(line), convert_token[token][0]))
        else:
            input_token.append((int(line), convert_token[token][1]))
    elif token == "KEYWORD":
        input_token.append((int(line), value))
    else:
        input_token.append((int(line), convert_token[token]))
f.close()
```

마지막에 종료기호를 넣어준 뒤, stack에도 시작 위치인 0을 삽입해 줍니다.

```
# Insert terminating character
input_token.append((input_token[len(input_token) - 1][0] + 1, "$"))
# Insert starting state into the "stack"
stack = deque()
stack.append(0)
action = ""
```

스택에 계속 현재 위치를 삽입해 줄 것인데, 이를 통한 이점은 두 가지가 있습니다.

1. 왼쪽 문자열이 viable prefix인지 매번 검사할 필요가 없습니다.
2. 다음 action을 받아올 때 스택에서 현 위치를 확인해 바로 받아올 수 있습니다.

위치를 계속 넣어 준다면 SLR table에서 현 위치를 바로 얻어올 수 있고, 다음 input에 대한 action이 존재하지 않다면 viable prefix가 아니게 됩니다. 이는 다르게 말하면 action이 있다면 viable prefix임을 증명해 주는 것입니다.

다음은 action을 받아오는 코드입니다. stack에서 현 위치를 파악해 SLR table에서 다음 action을 가져옵니다. 만약 action이 없다면 reject합니다.

```
if str(stack[-1]).isdigit():
    if input_token[0][1] in syntax_analyzer[stack[-1]]:
        action = syntax_analyzer[stack[-1]][input_token[0][1]]
    else:
        print(
            "reject at line",
            input_token[0][0],
            "and input token is",
            input_token[0][1],
        )
        exit(1)
else:
    if stack[-1] in syntax_analyzer[stack[-2]]:
        action = syntax_analyzer[stack[-2]][stack[-1]]
    else:
        print(
            "reject at line",
            input_token[0][0],
            "and input token is",
            input_token[0][1],
        )
        exit(1)
```

그 후 받은 action을 바탕으로 알맞은 문자열을 stack에 넣습니다.


```

if isinstance(action, list):
    # we don't have to qualify left substring every time because we put in
    # state in stack
    # it imply left substring is viable prefix if it has rule in str table
    # do shift
    if action[0] == "s":
        stack.append(input_token.popleft()[1])
        stack.append(action[1])
    # do reduce
    else:
        if "" in reduce[action[1]]:
            stack.append(reduce[action[1]][""])
        else:
            tmp = deque()
            while " ".join(tmp) not in reduce[action[1]]:
                t = str(stack.pop())
                if t.isalpha():
                    tmp.appendleft(t)
            stack.append(reduce[action[1]][" ".join(tmp)])
# do "goto" action
else:
    stack.append(action)

```

이와 같은 과정을 action이 accept가 될 때까지 반복합니다. 만약 성공적으로 수행했다면 마지막에 accept!를 출력하게 됩니다.

에러 발생 시 원본 파일 라인 표시

Syntax analysis 과정에서 에러가 발생할 경우, 원본 파일의 라인을 표시하기 위해 Lexical analyzer 의 코드를 일부 변경하였습니다.

```

lineNumber = 1
for token, v in output:
    # skip white space
    if token == "WHITE SPACE":
        if "\n" in v:
            lineNumber += v.count("\n")
        continue
    f.write(str(lineNumber) + " " + token + " " + v + "\n")
    print(token, v)
f.close()

```

출력부에서, lineNumber라는 변수를 선언하였습니다. 그리고 WHITE SPACE를 만날 경우 기존에는 skip 하고 넘어갔었는데, 이제는 new line 문자의 수 만큼 lineNumber를 더하게 처리했고, 출력 시에도 lineNumber를 함께 출력하게 해서 Syntax analysis 중에 에러가 발생할 경우, 바로 원본 파일의 어느 라인인지 파악할 수 있게 했습니다.

모듈화

Lexical analyzer와 Syntax analyzer 모두 각각 동작할 수 있지만 동시에 수행할 수도 있도록 하였습니다. 다음은 실행 방법입니다.

```
lex_and_parse.py test2.java
```

Cmd :

```
C:\Users\WTG\Documents\GitHub\handmade-compiler>lex_and_parse.py test2.java  
accept!
```

Test 파일

다음과 같이 여러가지 문법들을 사용해 테스트 하였습니다. test2.java 파일입니다.

```
class test {  
    char ch='c';  
    int cl=a+b;  
    int main(int num1, String str) {  
        int _aa_a = -1;  
        char b_b123 = 'c';  
        boolean __cc = true;  
        String d__d_0 = "Hello World";  
        char _123if = ' '  
        int _123if0 = -1-0;  
  
        if(true) {  
            _aa_a = 3923 + 41 - -1- -1-1-1;  
        }else{  
            while(true){  
                _aa_a=_aa_a+1;  
            }  
            b_b123 = '3';  
        }  
  
        return 0;  
    }  
  
    int function(int num){  
        int a=3;  
        int b=4;  
        num=a+b;  
        return num;  
    }  
}  
class test2 {  
    int temp(int num1, String str) {  
        int t = -1;
```

```

char b_b123 = 'c';
boolean __cc = true;
String d__d_0 = "Hello World";
char _123if = ' ';
int _123if0 = -1-0;

if(true) {
    t = 3923 + 41 - -1- -1-1-1;
    if(true){

    }else{
        int d=5;
    }
}else{} if(true){
    b_b123 = '3';
}else{
    __cc=false;
}
return 0;
}

int function1(int num){
    int a=3;
    int b=4;
    num=((a+b)*(a-b))+b;
    b=b*(a+b);
    while(true!=false){
        a=a+1;
        while(true){
            int c=2;

        }
    }
    String str1="STR";
    return num+a*b;
}

String function2(String str, String str1, String str2, int num1, int
num2, int num3){
    return str;
}

int function3(){
    return 0+1*-1;
}
}
class test3{
    int function4(){
        return 0;
    }
    int num=-1231;
}

```

에러를 잡는지 확인하기 위해 31번째 줄의 '}'를 제거했을 때의 결과입니다.

```
...
    int function(int num){
        int a=3;
        int b=4;
        num=a+b;
        return num;
    }
class test2 {
    int temp(int num1, String str) {
        int t = -1;
        char b_b123 = 'c';
    }
}
```

Cmd :

```
C:\Users\WTG\Documents\GitHub\handmade-compiler>lex_and_parse.py test2.java
reject at line 31 and input token is class
```