2020 COMPILER Project

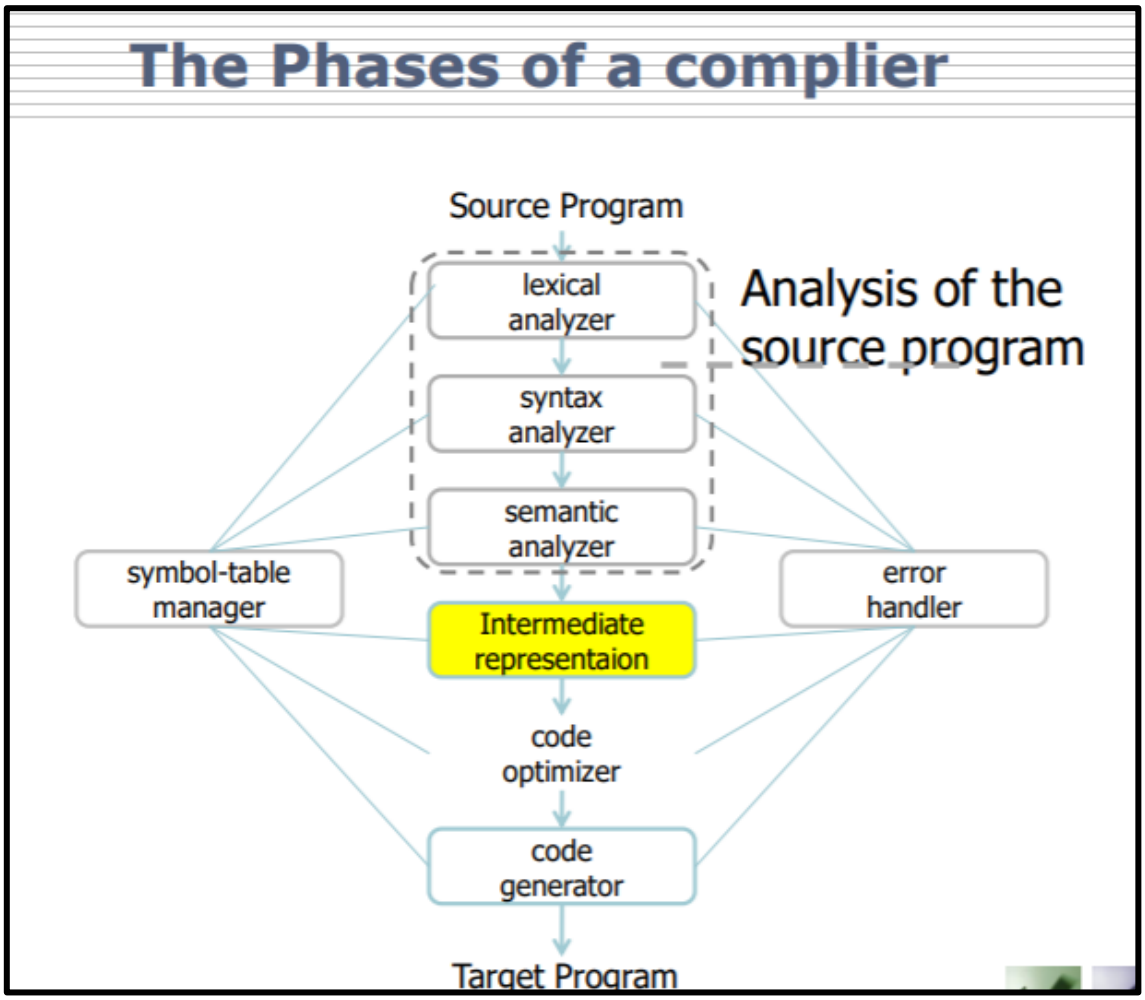
Team name : 141517

김민하, 신성국, 임정한

I. 서론

1. 과제 개요

본 보고서는 간단한 컴파일러를 python언어를 사용하여 구현하는 프로그래밍언어 팀 프로젝트의 일환이다. 컴파일러는 아래의 그림과 같이 여러 가지 Analyzer들로 구분하여 진행한다. 본 작성자들은 아래의 그림에 나와있는 것과 같이 단계로 나누어 설명하고자 한다.



2. 실행 방법

본 코드는 Unix 환경에서 실행이 되도록 진행하였으며, 컴파일러를 실행하는 코드인 Compiler141517.py와 입력할 input 파일로 testfile1, testfile2, testfile3을 명령어로

받도록 처리한다. 아래의 캡처본으로 해당 컴파일러를 실행할 수 있다.

```
$ python3 Compiler141517.py testfile1
```

II. 본론

1. Lexical Analyzer

```
def lexical(self):
    i=0

    while i<self.length:
        if self.code[i].isdigit(): # number(length >=1)
            token = self.get_digit(i)
            self.tokens.append(['number', token])
            i+= len(token)

        elif self.code[i].isalpha(): #word, type, statement(length>=1)
            token = self.get_str(i)
            if token in self.type:
                self.tokens.append(['type : ', token])
            elif token in self.statement:
                self.tokens.append(['statement : ', token])
            elif token in self.exit:
                self.tokens.append(['EXIT : ', token])
            else:
                self.tokens.append(['word : ', token])
            i+=len(token)

        elif self.code[i] == ' ' or self.code[i] == '\n' or self.code[i] == 9 or
self.code[i] == '    ':
            i += 1

        else: #bracket, operator or error(length =1) (for operator '==' : length =2)
            if self.code[i] in self.bracket:
                self.tokens.append(['bracket : ', self.code[i]])
                i+=1
            elif self.code[i] in self.operator:
                if self.check_equality(i):
                    self.tokens.append(['operator : ', '=='])
                    i+=2
                else:
                    self.tokens.append(['operator : ', self.code[i]])
                    i+=1
            elif self.code[i] in self.special_character:
                self.tokens.append(['comma, semicolon : ', self.code[i]])
                i+=1
            else:
                self.tokens.append(0)
                print("Error in Lexical Analysis: wrong input {}".format(i))
                sys.exit()
```

Lexical.py는 input file에 있는 코드들을 token화하는 작업을 구현한다. 주어진 grammar type들을 기준으로 토큰을 분류하여 token화가 진행되면 해당 input file에 대한 전체 token을 출력하도록 구현하였다. 위의 코드는 input 코드를 여러가지 type들로 나누어 self.tokens에 저장하는 방법을 구현한 것이다. 숫자는, 예를 들어 '151'의 경우, 하나의 input으로 토큰을 처리하면 전혀 다른 값을 token화하여 저장하는 문제가 발생되고, 이를 방지하기 위해 숫자와 영어단어의 경우 처음에 숫자와 영어로 인식되면 숫자와 영어가 나오지 않을 때까지 계속해서 그 다음 input에 대한 정보를 인식한다. 이는 get_digit과 get_str 함수를 통해 구현한다.

```
def get_digit(self, i):
    j=1
    while i+j < self.length:
        if self.code[i+j].isdigit():
            j+=1
        else:
            break
    return self.code[i:i+j]

def get_str(self, i):
    j=1
    while i+j < self.length:
        if self.code[i+j].isalpha():
            j+=1
        else:
            break
    return self.code[i:i+j]
```

그 외 괄호, 쉼표, 세미콜론, 연산기호 등도 따로 token화하여 저장하게 한다.

위의 lexical.py를 통해 testfile1.txt의 코드를 토큰화하는 결과는 다음과 같다.

Lexical.py를 통해 나온 토큰 결과 :

```
From Lexical Analyzer, tokens:

['word : ', 'main']
['bracket : ', '(']
['bracket : ', ')']
['bracket : ', '{']
['type : ', 'int']
['word : ', 'a']
['comma, semicolon : ', ';']
['type : ', 'int']
['word : ', 'b']
['comma, semicolon : ', ';']
['type : ', 'int']
['word : ', 'c']
```

```
['comma, semicolon : ', ';']
['type : ', 'char']
['word : ', 'd']
['comma, semicolon : ', ';']
['type : ', 'char']
['word : ', 'e']
['comma, semicolon : ', ';']
['word : ', 'b']
['operator : ', '=']
['number', '3']
['comma, semicolon : ', ';']
['word : ', 'a']
['operator : ', '=']
['word : ', 'b']
['operator : ', '+']
['number', '14']
['comma, semicolon : ', ';']
['word : ', 'd']
['operator : ', '=']
['number', '4']
['operator : ', '*']
['number', '3']
['operator : ', '+']
['number', '2']
['operator : ', '*']
['number', '1']
['comma, semicolon : ', ';']
['statement : ', 'IF']
['word : ', 'a']
['operator : ', '+']
['number', '1']
['operator : ', '>']
['word : ', 'c']
['statement : ', 'THEN']
['bracket : ', '{']
['word : ', 'e']
['operator : ', '=']
['number', '5']
['comma, semicolon : ', ';']
['bracket : ', '}']
['statement : ', 'ELSE']
['bracket : ', '{']
['word : ', 'e']
['operator : ', '=']
['number', '3']
['comma, semicolon : ', ';']
['bracket : ', '}']
['EXIT : ', 'EXIT']
['number', '0']
['comma, semicolon : ', ';']
['bracket : ', '}']
```

2. Parser

```
def make_LLgrammar(self):
    pre_grammar = self.read_from_grammar_path()
    grammar_without_lr = self.resolve_left_recursion(pre_grammar)
    LLgrammar = self.resolve_left_factoring(grammar_without_lr)
    for key, value in LLgrammar.items():
        idx = 0
        while idx < len(value):
            for idx2 in range(len(value[idx])):
                value[idx][idx2] = value[idx][idx2].strip("\"")
            idx += 1
    return LLgrammar

def read_from_grammar_path(self):
    grammar = dict()
    with open(self.grammar_path, 'r') as g:
        pre_grammar = g.read()
    pre_grammar_list = pre_grammar.strip().split(';\n')
    for i in pre_grammar_list:
        LHS = i.split('->')[0].strip()
        RHS = i.split('->')[1].strip()
        RHS_list = []
        if '|' in RHS and RHS.startswith('(') is False:
            RHS = RHS.split('|')
            for j in range(len(RHS)):
                RHS[j] = RHS[j].strip()
            for j in RHS:
                j = j.split(' ')
                for k in range(len(j)):
                    j[k] = j[k].strip()
                RHS_list.append(j)
        else:
            if RHS.startswith('(') is False:
                RHS = RHS.split(' ')
                for j in range(len(RHS)):
                    RHS[j] = RHS[j].strip()
            else:
                temp = RHS
                RHS = []
                RHS.append(temp)
            RHS_list.append(RHS)
        grammar[LHS] = RHS_list
    return grammar

def resolve_left_recursion(self, grammar):
    lr_removed_grammar = grammar.copy()
    for key, value in grammar.items():
        leftmost = []
        if len(value) > 1:
            for j in value:
                leftmost.append(j[0])
        else:
            leftmost.append(value[0])
```

```
for k in leftmost:
            if key == k:
                temp = dict()
                temp[key] = lr_removed_grammar[key]
                del lr_removed_grammar[key]
                if temp[key][1] != ['']:
                    lr_removed_grammar[key] = [[temp[key][1][0], k+'\'']]
                else:
                    lr_removed_grammar[key] = [[k + '\'']]
                new_value_list = [[temp[key][0][1], k + '\''], ['']]
                lr_removed_grammar[k + '\''] = new_value_list

    grammar = lr_removed_grammar.copy()
    return grammar

def resolve_left_factoring(self, grammar):
    lf_removed_grammar = grammar.copy()
    for key, value in grammar.items():
        if len(value) > 1:
            lf_index = 0
            leftmost = value[0][0]
            for j in value:
                if j[0] == leftmost:
                    continue
                else:
                    lf_index = 1
                    break;
            if lf_index == 0:
                temp = dict()
                temp[key] = lf_removed_grammar[key]
                del lf_removed_grammar[key]
                lf_removed_grammar[key] = [[leftmost, key + '\'']]
                if len(temp[key][0][1:]) == 0:
                    new_value_list = [temp[key][1][1:], ['']]
                else:
                    new_value_list = [temp[key][0][1:], temp[key][1][1:]]
                lf_removed_grammar[key + '\''] = new_value_list

    grammar = lf_removed_grammar.copy()
    return grammar
```

먼저 LL(1) 파서를 만들기 위해서는 **ambiguity, left-recursion, left-factoring**의 문제를 해결한 LL(1) 문법을 만들어야 한다. 따라서 위 코드를 바탕으로 left-recursion과 left-factoring의 문제를 해결한 새로운 문법을 만들 수 있었다. 두 문제를 해결하고 파싱 테이블을 만들어본 결과, **표의 여러 엔트리에 문법이 두개 이상씩 삽입되어 LL(1) 문법에 아직 ambiguity가 해소되지 않았음**을 알게 되었다. 과제에서 주어진 문법을 최대한 변형하지 않기 위해 주어진 문법으로 LL(1) 파서보다 더 강력한 LR(1) 파서를 만들어 보았지만, 여전히 ambiguity가 해소되지 않아 shift/reduce conflict이 나타남을 확인하였다. 따라서, ambiguity를 제거하고 LL(1) 파서를 만들기 위해서는 **반드시 주어진 문법을 변형해야 한다고 판단하여, 다음과 같이 변경하였다**.

```
prog -> word "(" ")" block ;
decls -> decls decl
| ;
decl -> vtype word ";" ;
vtype -> int | char ;
block -> "{" decls slist "}"
| ;
slist -> slist stat
| ;
stat -> IF cond THEN block ELSE block
| WHILE cond block
| word "=" expr ";"
| EXIT expr ";" ;
cond -> expr ">" expr
| expr "==" expr ;
expr -> term
| term "+" term ;
term -> fact
| fact "*" fact ;
fact -> num
| word ;
word -> ([a-z] | [A-Z])* ;
num -> [0-9]*
```

우리 조는 ambiguity를 제거하기 위하여,

1. (vtype -> int | char | '') → ***(vtype -> int | char)***

2. (stat -> IF cond THEN block ELSE block | WHILE cond block
| word = expr ; | EXIT expr ; | '')
→ (stat -> IF cond THEN block ELSE block | WHILE cond block
| word = expr ; | EXIT expr ;)

3. (slist -> slist stat | stat) → ***(slist -> slist stat | '')***

로 문법을 변경하였다.

일단 변수를 선언할 때 type을 지정하지 않고 변수를 선언할 수 없기 때문에, (vtype -> '') 은 제거하여도 괜찮다고 생각하였고, 계속 중복된 엔트리를 가지는 stat에서 마찬가지로 (stat -> '')를 제거하였다. 대신 stat을 입실론으로 바꾸는 문법의 의미를 그대로 유지하기 위하여 (slist -> stat)을 (slist -> '')로 변경하였다. 다음과 같이 문법을 바꾸었더니 ambiguity가 제거되어 성공적으로 LL(1) 파서를 만들 수 있었다.

```
def get_first(self, key):
    first_list = []
    if len(self.grammar[key]) == 1:
        value = self.grammar[key][0]
        if value[0] in self.terminal:
            first_list.append(value[0])
        else:
            first_list.extend(self.get_first(value[0]))
    else:
        for i in range(len(self.grammar[key])):
            value = self.grammar[key][i][0]
            if value in self.terminal:
                first_list.append(value)
            else:
                first_list.extend(self.get_first(value))
    return first_list
```

```
def get_follow(self, target_key):
    follow_list = []
    if target_key == 'prog':
        follow_list.append('$')

    for key, value in self.grammar.items():
        if len(value) == 1:
            if target_key in value[0]:
                index = value[0].index(target_key)
                # not the last position
                if index != len(value[0]) - 1:
                    if value[0][index+1] in self.terminal:
                        follow_list.append(value[0][index+1])
                    else:
                        temp_first = self.first[value[0][index+1]].copy()
                        if '' not in temp_first:
                            follow_list.extend(temp_first)
                        else:
                            temp_first.remove('')
                            follow_list.extend(temp_first)
                            follow_list.extend(self.get_follow(key))
                # the last position
                else:
                    if target_key != key:
                        follow_list.extend(self.get_follow(key))
                    else:
                        continue
            else:
                continue
        else:
            for i in range(len(value)):
                if target_key in value[i]:
                    index = value[i].index(target_key)
                    # not the last position
                    if index != len(value[i]) - 1:
                        if value[i][index + 1] in self.terminal:
                            follow_list.append(value[i][index + 1])
                        else:
                            temp_first = self.first[value[i][index + 1]].copy()
                            if '' not in temp_first:
                                follow_list.extend(temp_first)
                            else:
                                temp_first.remove('')
                                follow_list.extend(temp_first)
                                follow_list.extend(self.get_follow(key))
                    # the last position
                    else:
                        if target_key != key:
                            follow_list.extend(self.get_follow(key))
                        else:
                            continue
                else:
                    continue
    return follow_list
```

위 두 함수를 사용하여 전처리된 문법에서 각 non-terminal의 FIRST, FOLLOW set

을 구하고 이를 통하여 predictive parsing table을 만들 수 있었다. 우리 조가 만든 predictive parsing table은 다음과 같다.

```
================Table==============
     } > IF + * ) ([a-z] | [A-Z])* WHILE { ( = EXIT ELSE THEN char ; [0-9]* == int $

prog 0 0 0 0 0 0 ['word', '(', ')', 'block'] 0 0 0 0 0 0 0 0 0 0 0 0 0

decl 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ['vtype', 'word', ';'] 0 0 0 ['vtype', 'word', ';'] 0

vtype 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ['char'] 0 0 0 ['int'] 0

block [''] 0 [''] 0 0 0 [''] [''] ['{', 'decls', 'slist', '}'] 0 0 [''] [''] 0 0 0 0 0 0 ['']

stat 0 0 ['IF', 'cond', 'THEN', 'block', 'ELSE', 'block'] 0 0 0 ['word', '=', 'expr', ';']
['WHILE', 'cond', 'block'] 0 0 0 ['EXIT', 'expr', ';'] 0 0 0 0 0 0 0 0

fact 0 0 0 0 0 0 ['word'] 0 0 0 0 0 0 0 0 0 ['num'] 0 0 0

word 0 0 0 0 0 0 ['([a-z] | [A-Z])*'] 0 0 0 0 0 0 0 0 0 0 0 0 0

num 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ['[0-9]*'] 0 0 0

decls [''] 0 [''] 0 0 0 [''] [''] 0 0 0 [''] [''] 0 ["decls'"] 0 0 0 ["decls'"] ['']

decls' [''] 0 [''] 0 0 0 [''] [''] 0 0 0 [''] [''] 0 ['decl', "decls'"] 0 0 0 ['decl', "decls'"]
['']

slist [''] 0 ["slist'"] 0 0 0 ["slist'"] ["slist'"] 0 0 0 ["slist'"] 0 0 0 0 0 0 0 0

slist' [''] 0 ['stat', "slist'"] 0 0 0 ['stat', "slist'"] ['stat', "slist'"] 0 0 0 ['stat',
"slist'"] 0 0 0 0 0 0 0 0

cond 0 0 0 0 0 0 ['expr', "cond'"] 0 0 0 0 0 0 0 0 0 ['expr', "cond'"] 0 0 0

cond' 0 ['>', 'expr'] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ['==', 'expr'] 0 0

expr 0 0 0 0 0 0 ['term', "expr'"] 0 0 0 0 0 0 0 0 0 ['term', "expr'"] 0 0 0

expr' [''] [''] [''] ['+', 'term'] 0 0 [''] [''] [''] 0 0 [''] 0 [''] 0 [''] 0 [''] 0 0

term 0 0 0 0 0 0 ['fact', "term'"] 0 0 0 0 0 0 0 0 0 ['fact', "term'"] 0 0 0

term' [''] [''] [''] [''] ['*', 'fact'] 0 [''] [''] [''] 0 0 [''] 0 [''] 0 [''] 0 [''] 0 0
================Table Ends==============
```

3. Parser Tree

이전에 만들어진 parse table을 이용하여 tree를 빌드한다. 트리를 빌드하는 알고리즘은 다음과 같다.

1. 선처리

파싱을 하기 위해 선처리를 한다. 선처리는 크게 두 가지 작업을 하였다.

1. string으로 들어온 인풋 파일, terminal과 nonterminal이 들어있는 딕셔너리, 그리고 인풋 파일과 비교할 스택이다. 또한, parse_tree를 초기화한다. Node의 root는 $로 한다.

Input_txt	인풋 파일	첫 시작을 위해 $를 append한다.
Terminal Nonterminal	터미널, 넌터미널 딕셔너리	모든 terminal과 nonterminal의 word와 key(integer)를 저장한다.
stack	스택	첫 시작을 위해 $를 append한다.

```
def preprocess_parsing(self, input_txt):
        input_txt.append("$")
        terminal = {key : word for word, key in enumerate(self.terminal)}
        non_terminal = {key: word for word, key in
enumerate(self.nonterminal)}
        stack = list()
        stack.append(self.nonterminal[0])
        stack.append('$')

        return input_txt, terminal, non_terminal, stack


    def parsing(self, input_txt):
        #Get preprocessed data of input_txt(list),
terminal&non_terminal(dict), stack(list)
        input_txt, terminal, non_terminal, stack =
self.preprocess_parsing(input_txt)

        self.parse_tree = Node(stack[0], None, 0)
```

2. token을 가공한다.
   대부분의 Token은 가공할 필요가 없지만, 일부 Token에 대해서는 Token을 가공해야 한다. 바꾸어야 할 토큰들은 다음과 같다.

바꾸기 전	바꾼 후
String	([a-z] | [A-Z])*
Integer	[0-9]*

```
def tokens_to_input(self, tokens):
        temp_list = []
        for token_type, token in tokens:
            if token_type == "word : ":
                temp_list.append("([a-z] | [A-Z])*")
                self.word_tokens.append(token)
            elif token_type == "number":
                temp_list.append("[0-9]*")
                self.num_tokens.append(token)
            else:
                temp_list.append(token)
        return temp_list
```

2. 트리 구성

알고리즘은 다음과 같다.

스택이 비기 전까지 다음 Action 을 취한다:

Input_txt 와 문자를 비교하여 다음 행동을 취한다.

ㄱ. 입력 문자열과 스택을 비교하여 같으면 삭제한다

ㄴ. 입력 문자열과 스택이 모두 비었으면($) 작동을 종료한다

ㄷ. 문자열과 스택이 다르면 파싱 테이블에서 해당 문자열과 스택이 해당된 테이블에 있는 문법에 따른다.

```
while len(stack) > 1:
            comp = stack.pop(0)
            symbol = input_txt[0]


            if comp in terminal:
                if symbol == comp.strip("\""):
                    input_txt.pop(0)

                    if symbol == '([a-z] | [A-Z])*':
                        self.parse_tree.id = self.word_tokens.pop(0)

                    elif symbol == '[0-9]*':
                        self.parse_tree.id = self.num_tokens.pop(0)

                    self.parse_tree = self.parse_tree.advance()

                #Finished.
                else:
                    check = True
                    break

            if comp not in terminal:
                row = 0
                col = self.table[0].index(symbol)

                for idx in range(len(self.nonterminal)):
                    if self.nonterminal[idx] == comp:
                        row = idx + 1
                        break

                push = self.table[row][col]
                if type(push) == str:
                    push = [push]

                if push is not 0:
                    if push != ['']:
                        stack = push + stack
                        #set push
                        self.parse_tree.set_child(push)
                        #Move cur
                        self.parse_tree = self.parse_tree.children[0]
                    else:
                        #Child goes to e
                        self.parse_tree.set_child([''])
                        #
                        self.parse_tree = self.parse_tree.advance()

                #Finished
                else:
                    check = True
                    break
        if check:
            return None
        else:
            self.parse_tree = self.parse_tree.get_root()
            return self.parse_tree
```

4. Symbol table and Semantic analyzing

```
def set_symbol_table(self):
    node = self
    scope = ["global"]
    symbol_table = []
    while len(node.children) != 0:
        node = node.children[0]
    symbol_table.append([node.id, "function", list(scope)])
    scope.append(node.id)
    node = node.get_next()
    while node.parent is not None:
        if node.data in ["char", "int"]:
            node_type = node.data
            node = node.get_next()
            if node.data == "([a-z] | [A-Z])*":
                symbol_table.append([node.id, node_type, list(scope)])
                node = node.get_next()
                continue
        elif node.data in ["IF", "WHILE", "ELSE"]:
            scope.append(node.data)
        elif node.data == "}":
            scope.pop()
        node = node.get_next()
    return symbol_table
```

LL(1) 파서를 만들고 lexical analyzer로부터 토큰을 받아 Intermediate representation을 만들기 전에, symbol table을 구성하여 semantic analyze의 과정을 거쳤다. 다음 코드를 바탕으로 소스 코드에 나타난 각 함수와 변수의 type과 scope를 저장하였다. testfile1로 생성한 symbol table은 다음과 같다.

```
main(){
  int a;
  int b;
  int c;
  char d;
  char e;

  b = 3;
  a = b + 14;
  d = 4 * 3 + 2 * 1;

  IF a + 1 > c  THEN {
     e = 5;
  }
  ELSE {
     e = 3;
  }

  EXIT 0;
}
```

```
Symbol Table, testfile1
name : main
{  type : function
   scope : global }

name : a
{  type : int
   scope : global -> main }

name : b
{  type : int
   scope : global -> main }

name : c
{  type : int
   scope : global -> main }

name : d
{  type : char
   scope : global -> main }

name : e
{  type : char
   scope : global -> main }
```

semantic analyzing 과정을 거친 후, Code Generator을 통하여 assembly-like 코드를 만들기 이전에 Intermediate representation을 만들기 위하여 우리 조는 Three address code를 사용하였다.

```
def construct_IR(self, node):
    self.cur = node
    ir = []
    last_leafnode = node
    while last_leafnode.children:
        last_leafnode = last_leafnode.children[-1]

    while self.cur != last_leafnode:
        if self.cur.data == "stat":
            n = self.cur.children[0]

            if n.data in ["word", "EXIT"]:
                ir.append([self.WORD_or_EXIT(n), self.cur])
            elif n.data == "WHILE":
                if self.cur.children[2].children:
                    ir += self.WHILEcode()
            else:
                ir += self.IFcode()
        self.cur = self.cur.search_inorder()
    return ir
```

다음 코드를 통해 Intermediate representation 코드를 만들 수 있었다. 제작된 parse-tree를 인풋으로 받아서 각 트리 노드를 탐색하며 stat non-terminal을 마주치면 IR를 만든다. stat을 IR로 만들 때, 자식 노드의 수와 cond의 존재 유무에 따라 word, EXIT/

WHILE/ IF로 나누어 IR을 따로 생성하였다. testfile1로 만들어진 IR은 다음과 같다.

```
=======Intermediate representation========
0. BEGIN main
1. b = 3
2. a = b  + 14
3. d = 4 * 3 + 2 * 1
4. if a + 1 > c THEN goto L1
5. e = 3
6. goto L2
7. L1
8. e = 5
9. L2
10. EXIT 0
11. END main
```

5. Code Generator

Code_generator() 중 w_code() 함수 코드

```
class code_generator():
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~생략
def w_code(self,ir,node):
    nid = "Reg#" + str(node.id)

    if node.children:
        left = node.children[0]
        right = node.children[1]
        lid = "Reg#" + str(left.id)
        rid = "Reg#" + str(right.id)
        if not left.children:
            if node.data != "=":
                self.w_code("",left)
            self.w_code("",right)

            if node.data == "+":
                self.code.append("  ADD   {},   {},   {}".format(nid,lid,rid))
            elif node.data == "*":
                self.code.append("  MUL   {},   {},   {}".format(nid,lid,rid))
            elif node.data == "=":
                self.code.append("  ST    {},   {}".format(rid, left.data))

            else:
                self.code.append("  LT    {},   {},   {}".format(nid,lid,rid))
                self.code.append("  JUMPT {},   {}".format(nid, ir.split("goto")[-1][1:]))
        elif not right.children:
            self.w_code("",left)
            if node.data == "+":
                self.code.append("  ADD   {},   {},   {}".format(nid,lid,rid))
            elif node.data == "*":
                self.code.append("  MUL   {},   {},   {}".format(nid,lid,rid))
            elif node.data == "=":
                self.code.append("  ST    {},   {}".format(rid, left.data))

            else:  # >
                self.code.append("  LT    " + nid + ",   " + rid + ",   " + lid)
                self.code.append("  JUMPT " + nid + ",   " + ir.split("goto")[-1][1:])

        else:
            self.w_code("",left)
            self.w_code("",right)
            if node.data == "+":
                self.code.append("  ADD   " + nid + ",   " + lid + ",   " + rid)
            elif node.data == "*":
                self.code.append("  MUL   " + nid + ",   " + lid + ",   " + rid)
            elif node.data == "=":
                self.code.append("  ST    {},   {}".format(rid, left.data))

            else:  # >
                self.code.append("  LT    {},   {},   {}".format(nid,lid,rid))
                self.code.append("  JUMPT {},   {}".format(nid, ir.split("goto")[-1][1:]))
    elif ir[:4] == "EXIT":
        self.code.append("  LD    rax,   -1")
    elif str(node.id).isdigit():
        self.code.append("  LD    " + "Reg#{}".format(node.id) + ",   "+node.data)
```

Code_generator.py는 생성한 Intermediate Representations statement를 바탕으로

각 statement에 대한 코드를 생성하는 작업을 구현한 파일이다. input으로 노드(트리) 형태의 intermediate representation을 받고, 해당 노드에 저장된 값에 따라 세분화하여 구현하도록 하였다. 해당 결과는 Compiler141517.py.code에 저장되어 있다.

아래의 결과는 testfile1로 code generating 결과를 보여준 것이다. 이와 더불어 해당 프로그램을 실행하는데 필요한 최소 레지스터 개수를 마지막 줄에 출력하였다. Testfile3.txt의 경우 3개의 레지스터로 컴파일러를 실행할 수 있다.

Compiler141517.py.code 결과 내용

```
Generating codes, testfile1
BEGIN main
  LD    Reg#1,   3
  ST    Reg#1,   b
  LD    Reg#2,   b
  LD    Reg#1,   14
  ADD   Reg#1,   Reg#2,   Reg#1
  ST    Reg#1,   a
  LD    Reg#2,   4
  LD    Reg#1,   3
  MUL   Reg#1,   Reg#2,   Reg#1
  LD    Reg#3,   2
  LD    Reg#2,   1
  MUL   Reg#2,   Reg#3,   Reg#2
  ADD   Reg#1,   Reg#1,   Reg#2
  ST    Reg#1,   d
  LD    Reg#2,   a
  LD    Reg#1,   1
  ADD   Reg#1,   Reg#2,   Reg#1
  LT    Reg#1,   Reg#2,   Reg#1
  JUMPT Reg#1,   L1
  LD    Reg#1,   3
  ST    Reg#1,   e
  JUMP          L2
L1
  LD    Reg#1,   5
  ST    Reg#1,   e
L2
  LD    rax,   -1
END main

Number of Used Register (except Rax) :3
```