프로젝트 : 재귀하강파서

과목명 : 프로그래밍 언어론

담당교수 : 김진성

학과 : 응용통계학과 / 소프트웨어학부

학번 : 20204885 / 20216946

이름 : 강태영 / 송재호

1. Internal

1.1. SetUp

```
import os
import sys
import keyword
# TOKEN CODE - jh
IDENTIFIER =1
ASSIGN_OP =2
ADD_OP =3
SUB_OP =4
MUL_OP =5
DIV_OP =6
SEMICOLON =7
LEFT_PAREN =8
RIGHT_PAREN =9
CONSTANT =10
EOF =-1
# global token - jh
next_token =None
token_string =""
op_count =0
errors = []
table = {}
paren =0
# 각 연산자에 해당하는 변수 맵핑 - jh
op_map = {
    '+': ADD_OP,
    '-': SUB_OP,
    '*': MUL_OP,
    '/': DIV_OP
}
# 괄호, 세미콜론에 해당하는 변수 맵핑 - jh
pl_map = {
    ';': SEMICOLON,
    '(': LEFT_PAREN,
    ')': RIGHT_PAREN
}
```

표준 라이브러리 호출, 토큰 코드 정의(상수) 및 전역 변수 초기화

* next_token : 현재 처리 중인 토큰 코드 저장
* token_string : 현제 토큰 코드의 실제 문자열 값을 저장
* op_count : 연산자의 개수 저장
* errors : 에러 메시지를 저장하는 리스트
* table : 심볼 테이블로 사용되는 딕셔너리(변수 : 값)
* paren : 괄호의 균형상태(“(”, “)”가 균형)를 저장

또한 연산자, 괄호 세미콜론 매핑

1.2. class SymbolTable

프로그램 내의 식별자와 해당 값을 관리하는 심볼 테이블을 구현하는 클래스

```
# global table 초기화 및 table 참조 - jh
class SymbolTable:
    def __init__(self):
        global table
        table = {}
    def add_symbol(self, ident, value="Unknown"):
        global table
        table[ident] = value
    def get_symbol(self, ident):
        global table
        return table.get(ident)    # 식별자가 존재하지 않을 경우 "Unknown" 반환 - jh
```

1.2.1. def __init__(self)

table 초기화

1.2.2. def add_symbol(self, ident, value = “Unknown”)

```
    def add_symbol(self, ident, value="Unknown"):
        global table
        table[ident] = value
```

table(딕셔너리)에 ident를 key, value를 value로 저장하며 parser에서 변수 선언 및 할당 시 호출

1.2.3. def get_symbol(self, ident)

```
    def get_symbol(self, ident):
        global table
        return table.get(ident)    # 식별자가 존재하지 않을 경우 "Unknown" 반환 - jh
```

특정 식별자의 값을 가져오며 parser에서 변수 참조 시 호출
식별자의 값을 반환하나, 식별자가 존재하지 않으면 “Unknown”반환

1.3. class LexicalAnalysis

어휘 분석기를 담당하는 클래스로서 주어진 코드에서 토큰을 분리하고, 해당 토큰의 타입과 값을 반환

```
class LexicalAnalysis:
```

1.3.1. def __init__(self, input_text, symbol_table)

```
    def __init__(self, input_text, symbol_table):
        global op_count, errors    # 구문분석 시 사용할 수 있도록 global로 - jh
        self.text = input_text
        self.pos =0
        self.current_char =self.text[self.pos]
        self.symbol_table = symbol_table
        self.ident_count =0
        self.const_count =0
        op_count =0
        errors = []
        self.read_line = []
        self.parse_success =True
```

객체 LexicalAnalysis를 초기화(분석할 코드의 텍스트(input_text)를 읽고, 심볼 테이블 정의, 현재 문자와 위치, 식별자 개수, 상수 개수, 전역 변수를 초기화)

1.3.2. def advance(self)

```
    def advance(self):
        """현재 text의 길이를 넘지 않을 때까지"""
        global errors
        self.pos +=1
        if self.pos <len(self.text):
            self.current_char =self.text[self.pos]
            if self.pos ==len(self.text) -1 and self.current_char !=';':    # 세미콜론 누락 확인
                errors.append("(Warning) ; 누락")
        else:
            self.current_char =None
```

현재 위치(pos)에서 한 칸 이동하며 코드의 끝에 도달할 때까지 이동한다. 코드 끝에 도착하지 않을 경우 current_char을 갱신함.

* 오류 처리

- “;” 누락 : 코드 끝에 세미콜론이 없을 경우 경고(warning) 메시지 추가

1.3.3. def skip_whitespace(self)

```
    def skip_whitespace(self):
        """빈 공간 건너뛰기 함수 종료 시 current_char는 빈 공간이 아닌 pos를 가리킴"""
        while self.current_char is not None and self.current_char.isspace():
            self.advance()
```

현재 위치가 공백인 경우 건너뜀, 해당 함수가 종료되면 current_char는 공백이 아닌 위치이다.

1.3.4. def get_number(self)

```
    def get_number(self):
        num_str =''
        while self.current_char is not None and self.current_char.isdigit():
            num_str +=self.current_char
            self.advance()
        self.const_count +=1
        return CONSTANT, int(num_str)
```

숫자(상수, CONSTANT)를 읽고 CONSTANT와 그 값을 반환한다. 또한 상수의 개수를 샌다.

1.3.5. def get_identifier(self)

```
    def get_identifier(self):
        global table, errors
        ident =''
        while self.current_char is not None and (self.current_char.isalpha() or self.current_char.isdigit()):
            ident +=self.current_char
            self.advance()
        if ident in keyword.kwlist:    # 예약어를 사용하였을 경우 error 발생
            errors.append("(Warning) 예약어가 사용되었음")

        self.ident_count +=1

        return IDENTIFIER, ident
```

식별자를 읽고 IDENTIFIER와 그 값을 반환한다. 또한 식별자의 개수를 샌다.
* 오류 처리
- 예약어 : 예약어 사용 시 경고(warning) 메시지 추가

1.3.6. def lexical(self)

```
    def lexical(self):
        """lexeme 얻기"""
        global op_count
        self.skip_whitespace()
        if self.current_char is None:
            return EOF, 'EOF'
        # 식별자 - jh
        if self.current_char.isalpha():
            token_type, ident =self.get_identifier()
            self.read_line.append(ident)
            return token_type, ident
        # 상수 - jh
        elif self.current_char.isdigit():
            token_type, number =self.get_number()
            self.read_line.append(str(number))
            return token_type, number
        # 연산자 - jh
        elif self.current_char in op_map:
            temp_char =self.current_char
            self.advance()
            self.read_line.append(temp_char)
            op_count +=1
            return op_map[temp_char], temp_char

        # 괄호 및 세미콜론 - jh
        elif self.current_char in pl_map:
            temp_char =self.current_char
            self.advance()
            self.read_line.append(temp_char)
            return pl_map[temp_char], temp_char
        # 할당 연산자 - jh
        elif self.current_char ==':':
            self.advance()
            if self.current_char =='=':
                self.advance()
                self.read_line.append(":=")
                return ASSIGN_OP, ':='

        elif self.current_char =='=':
            self.advance()
            self.read_line.append("=")
            return "=", "="
        else:
            return EOF, "EOF"
```

코드에서 하나의 토큰을 추출하고, 토큰의 타입과 값을 반환한다. 즉 어휘분석 함수이다. 식별자, 상수, 연산자, 괄호 및 세미콜론, 할당연산자가 그것이다. 또한 토큰 추출 시 위에서 정의한 op_map 및 pl_map을 사용하여 토큰을 매핑한다.

1.3.7. def analyze(self)

```
    # lexical을 통해 얻은 type과 string을 next_token, token_string에 저장 - jh
    def analyze(self):
        global next_token, token_string    # 구문분석 시 next_token으로 해당 token의 type을 비교 - jh
        next_token, token_string =self.lexical()
```

lexcial함수를 호출하여 next_token과 next_string에 현재 토큰의 타입과 값을 저장하며 parser에서 현재 토큰을 확인하는 데 사용된다.

1.3.8. def print_result(self)

```
    def print_result(self):
        if self.read_line:
            print(" ".join(self.read_line))
            print(f"ID: {self.ident_count}; CONST: {self.const_count}; OP: {op_count};")
            if self.parse_success and not errors:
                print('(OK)')
            else:
                for error in errors:
                    print(f"{error}")
            print()  # 줄 구분을 위해 추가 - jh
```

분석된 코드 결과를 출력한다. 출력 내용은 식별자 개수, 상수 개수, 연산자 개수 및 정상적으로 파싱이 되었는지를 출력한다. 또한 에러가 존재할 경우 에러도 출력한다.

1.4. class Parser

```
classParser:
```

구문 분석기를 담당하는 클래스로서, 문제에서 재시 된 구문 규칙에 따라 분석한다.

1.4.1. def __init__(self, lexer)

```
    def __init__(self, lexer):
        global next_token, token_string
        self.lexer = lexer
        self.errors = errors
        self.lexer.analyze() # 첫 토큰 가져옴
```

어휘 분석기 lexer 객체를 받아 구문 분석을 준비한다. 첫 토큰을 가져온다. 즉 구문 분석을 위한 준비를 하는 함수이다.

1.4.2. def program(self)

문법 : <program> -> <statements>

```
    # <program> -> <statements>
    def program(self):
        global next_token, token_string
        if next_token == EOF:
            return
        self.statements()
```

프로그램의 시작, 토큰이 끝에 도착하면 종료
* 오류 처리
- 프로그램이 비어 있는 경우 EOF처리

1.4.3. def statements(self)

문법 : <statements> -> <statement> | <statement><semi_colon><statements>

```
# <statements> -> <statement> ¦ <statement><semi_colon><statements>
def statements(self):
    global next_token, token_string
    if next_token == EOF:
        return
    # -> <statement>
    self.statement()

    # -> <statement> -> <semi_colon>
    if next_token == SEMICOLON:
        self.lexer.analyze()
        if token_string ==";":
            errors.append("(Warning) 이중 세미콜론")
        self.statements()
    else:
        pass
```

<statement>를 처리하며 ";"이 그다음에 있을 경우 다음 <statements>를 처리한다.

* 오류 처리

- 중복 세미콜론 : 세미콜론 중복사용 시 경고(Warning) 메시지 메시지 추가

1.4.4. def statement(self)

문법 : <statement> -> <ident><assignment_op><expression>

```
    # <statement> -> <ident><assignment_op><expression>
    def statement(self):
        global next_token, token_string, errors

        # -> <ident>
        if next_token == IDENTIFIER:
            ident = token_string
            self.lexer.analyze()
            # -> <assignment_op>
            if next_token == ASSIGN_OP:
                self.lexer.analyze()
                if token_string ==";":
                    errors.append("(Error) := 뒤에 표현식이 없음")
                result =self.expression()
                global_symbol_table.add_symbol(ident, result)
            elif next_token =='=': # <assign_op> 존재 안함
                errors.append("(Warning) assign_op_error")
                self.lexer.analyze()
                result =self.expression()    # :이 생략된 경우 그냥 값이 할당할 수 있도록 함
                global_symbol_table.add_symbol(ident, result)

            elif token_string in op_map:    # ident 후 바로 연산자가 나오게 된 경우
                temp_string = token_string
                self.lexer.analyze()
                if token_string =='=':
                    errors.append(f"(Error) := 나오기전 {temp_string} 바로 나옴 ")
                    errors.append("(Warning) assign_op_error")
                elif next_token == ASSIGN_OP:
                    errors.append(f"(Error) {temp_string} 이후에 ':=' 나옴")
                self.lexer.analyze()
                result =self.expression()
                global_symbol_table.add_symbol(ident, "Unknown")

        else: # <ident> 존재 안함
            errors.append("(Error) ident_error")
            self.lexer.analyze()
```

ident(식별자)가 존재하는지 확인하고 할당 연산자와 <expression>을 처리한 후 심볼 테이블에 값 저장

* 오류 처리

- 표현식 없음 : 할당 연산자 뒤에 표현식이 없는 경우 에러 메시지 추가
- 할당 연산자 이상 : “:=”이 아닌 할당 연산자를 처리할 경우 경고(Warning) 메시지 추가
- 할당 연산자 이전 이상 : “:=”이 나오기 전에 연산자가 나온 경우, “=”인 경우에는 이 에러 처리 후 위의 할당 연산자 이상 적용
- 식별자 없음 : 식별자가 없는 경우 에러 메시지 추가

1.4.5. def expression(self)

```
# <expression> -> <term><term_tail>
def expression(self):
    term =self.term()
    if term !="Unknown":
        result =self.term_tail(term)
    else:
        result ="Unknown"
        self.term_tail(term)
    return result
```

문법 : <expression> -> <term><term_tail>

전체 표현식을 처리하며 <term>, <term_tail>을 계산한다.

* 오류 처리

- 정의되지 않은 식별자가 연산에 사용된 경우 “Unknown”을 반환

1.4.6. def term_tail(self, term)

문법 : <term_tail> -> <add_op><term><term_tail> | E

```
# <term_tail> -> <add_op><term><term_tail> ¦ E
def term_tail(self, term):
    global next_token, token_string, errors, op_count
    term_tail_result = term
    # -> <add_op> : 덧셈연산실행
    if next_token == ADD_OP:
        self.lexer.analyze()
        if next_token == ADD_OP:
            errors.append('(Warning)\"중복 연산자(+) 제거\"')
            op_count -=1
            self.lexer.analyze()

        t =self.term()

        if t !="Unknown"and type(t) !=int:
            errors.append("(Error) (+)후 피연산자 누락")
            return "Unknown"
        self.term_tail(t) # 재귀
        if t !="Unknown"and term !="Unknown":
            term_tail_result = t + term
        else:
            return "Unknown"

    # -> <add_op> : 뺄셈연산실행
    elif next_token == SUB_OP:
        self.lexer.analyze()
        if next_token == SUB_OP:
            errors.append('(Warning)\"중복 연산자(-) 제거\"')
            op_count -=1
            self.lexer.analyze()
        t =self.term()
        if t !="Unknown"and type(t) !=int:
            errors.append("(Error) (-)후 피연산자 누락")
            return "Unknown"
        self.term_tail(t) # 재귀
        if t !="Unknown"and term !="Unknown":
            term_tail_result = term - t
        else:
            return "Unknown"
```

```
        elif next_token == CONSTANT or next_token == IDENTIFIER:
            errors.append("(Error) 연산자 (+, - *, /) 누락")
            return "Unknown"
        return term_tail_result
```

덧셈과 뺄셈 연산자 처리하고 재귀적으로 다음 term을 처리
* 오류 처리
- 중복연산자 : 중복연산자가 사용된 경우 경고(warning) 메시지 추가하고 복구
- 피연산자 없음 : 피연산자가 없는 경우 누락 에러 메시지 추가
- 연산자 누락 : 연산자가 없는 경우 에러 메시지 추가

1.4.7. def term(self)

문법 : <term> -> <factor><factor_tail>

```
    # <term> -> <factor><factor_tail>
    def term(self):
        factor =self.factor()
        if factor !="Unknown":
            term_result =self.factor_tail(factor)
        else:
            term_result ="Unknown"
            self.factor_tail(factor)
        return term_result
```

<factor>, <factor_tail>을 처리하며 곱셈과 나눗셈을 포함한 항을 처리
* 오류 처리
- 정의되지 않은 식별자가 연산에 사용된 경우 “Unknown” 반환

1.4.8. def factor_tail(self, factor)

문법 : <factor_tail> -> <mult_op><factor><factor_tail> | E

```
    # <factor_tail> -> <mult_op><factor><factor_tail> | E
    def factor_tail(self, factor):
        global next_token, token_string, op_count
        factor_tail_result = factor
        # -> <mult_op> : 곱셈 연산
        if next_token == MUL_OP:
            self.lexer.analyze()
            if next_token == MUL_OP:
                errors.append('(Warning)\"중복 연산자(*) 제거\"')
                op_count -=1
                self.lexer.analyze()
            f =self.factor()
            if f !="Unknown"and type(f) !=int:
                errors.append("(Error) (*)후 피연산자 누락")
                return "Unknown"
            self.factor_tail(f) # 재귀
            if f !="Unknown"and factor !="Unknown":
                factor_tail_result = f * factor
```

```
        # -> <mult_op> : 나눗셈 연산
        elif next_token == DIV_OP:
            self.lexer.analyze()

            if next_token == DIV_OP:
                errors.append('(Warning)\"중복 연산자(/) 제거\"')
                op_count -=1
                self.lexer.analyze()
            f =self.factor()
            if f !="Unknown"and type(f) !=int:
                errors.append("(Error) (/)후 피연산자 누락")
                return "Unknown"
            self.factor_tail(f) # 재귀
            if f !="Unknown"and factor !="Unknown":
                factor_tail_result = factor / f
        return factor_tail_result
```

곱셈과 나눗셈을 처리, 재귀적으로 처리

* 오류 처리

- 중복연산자 : 중복연산자가 사용된 경우 경고(warning) 메시지 추가하고 복구
- 피연산자 없음 : 피연산자가 없는 경우 누락 에러 메시지 추가

1.4.9. def factor(self)

문법 : <factor> -> <left_paren><expression><right_paren> | <ident> | <const>

```
    # <factor> -> <left_paren><expression><right_paren> ¦ <ident> ¦ <const>
    def factor(self):
        global next_token, token_string, paren
        # -> <left_paren>
        if next_token == LEFT_PAREN:
            paren +=1
            self.lexer.analyze()
            factor_result =self.expression()
            if next_token == RIGHT_PAREN:
                self.lexer.analyze()
            else:
                errors.append("(Error) ')' 없음")

            # <expr>다시 리턴이니까
            return factor_result

        # -> <ident>
        elif next_token ==  IDENTIFIER:
            ident =self.ident()
            self.lexer.analyze() # 터미널까지 옴
            if next_token == RIGHT_PAREN:
                paren -=1
                if paren <0:
                    errors.append("(Error) '(' 없음")
                    self.lexer.analyze()
                    return "Unknown"
            return ident
```

```
        # -> <const>
        elif next_token == CONSTANT:
            const =self.const()
            self.lexer.analyze()
            if next_token == RIGHT_PAREN:
                paren -=1
                if paren <0:
                    errors.append("(Error) '(' 없음")
                    self.lexer.analyze()
                    return "Unknown"
            return const
        else: # 아무것도 없는 경우
            errors.append("(Error) 예상치 못함")
            self.lexer.analyze()
```

괄호처리(괄호 내의 표현식 처리)

* 오류 처리

- “)”누락 : “)”누락 시 에러 메시지 추가
- “(”누락 : “(”누락 시 에러 메시지 추가
- 예상하지 못한 값 : 예상하지 못한 값을 입력받을 경우 에러 메시지 추가

1.4.10. def const(self)

```
    # <const> -> any decimal numbers
    def const(self):
        global token_string
        const = token_string
        return int(const)
```

십진수의 숫자를 처리하며, 현재의 token_string을 정수로 변환하여 반환

1.4.11. def ident(self)

```
    # <ident> -> any names conforming to C identifier rules
    def ident(self):
        global token_string, errors
        if token_string not in table:
            errors.append(f'(Error)\"정의되지 않은 변수 ({token_string})가 참조됨\"')
            global_symbol_table.add_symbol(token_string)
        ident = global_symbol_table.get_symbol(token_string)
        return ident
```

심볼 테이블에서 해당 식별자 규칙에 맞는 이름을 처리한다.

* 오류 처리

- 심볼테이블 정의 X : 식별자가 심볼 테이블에 정의되지 않았을 경우 에러 메시지 추가하고 심볼 테이블에 “Unknown”을 추가

1.5. __main__

```
if __name__ =="__main__":
    args = sys.argv[1:]
    txt_files = [f for f in args if f.endswith('.txt')]

    for file_name in txt_files:
        global_symbol_table = SymbolTable()
        print(f"'{file_name}' 파일 실행")
        with open(file_name, 'r') as f:
            for line in f:
                paren =0 # 괄호수 초기화 -ty
                line = line.strip()
                if line =="":
                    continue # 빈줄 처리 - ty
                lexer = LexicalAnalysis(line, global_symbol_table) # line.strip() 위로 가져와서 공백 처리 - ty
                parser = Parser(lexer)
                parser.program()
                lexer.print_result()

        # 결과 처리 - 앞줄로 당김 - ty
        sorted_dict = sorted(table.items())  # 'table'이 SymbolTable의 속성이라고 가정
        print('Result ==> ', end='')
        for i in range(len(sorted_dict)):
            print(sorted_dict[i][0], ':', sorted_dict[i][1], end='')
            if i != (len(sorted_dict)-1):
                print('; ', end='')
        print("\n")
```

입력한 .txt로 끝나는 모든 파일(소스코드)을 처리한다. .txt로 끝나는 평가파일들을 순차적으로 처리한다. 또한 양옆 공백을 제거하고 빈 줄은 건너뛴다. 괄호처리를 위해 괄호수를 0으로 초기화한다. 이후 한 줄씩 읽어 들어가면서 어휘분석과 구문분석을 진행한다. 이후 어휘분석결과를 출력한다. 변수에 할당된 값들이 Result이후에 출력된다.

2. External

2.1. 프로그램 실행 방법

평가파일.txt와 main.py를 동일한 디렉토리에 둔다. 이후 터미널에서 해당 디렉토리로 이동 후 python main.py (파일명).txt를 입력한다.

2.2. 요구사항 및 개발환경

개발환경 : VSCode / 파이썬 버전 : 3.13.0