

CAU PL - LL(1) parser - Internal Document

20224566 김동우

LL(1)파서 제작 과제의 Internal Document 이다.

Grammar

```
<program> → <statements>
<statements> → <statement> | <statement><semi_colon><statements>
<statement> → <ident><assignment_op><expression>
<expression> → <term><term_tail>
<term_tail> → <add_op><term><term_tail> | ε
<term> → <factor> <factor_tail>
<factor_tail> → <mult_op><factor><factor_tail> | ε
<factor> → <left_paren><expression><right_paren> | <ident> | <const>
<const> → any decimal numbers
<ident> → any names conforming to C identifier rules
<assignment_op> → :=
<semi_colon> → ;
<add_operator> → + | -
<mult_operator> → * | /
<left_paren> → (
<right_paren> → )
```

File List

main.py - 진입점(entry point)

Lexer.py - lexical() 함수와 오류처리를 위한 함수들을 포함하는 Lexer 클래스의 .py file

Parser.py - 각 non-terminal들에 대한 함수를 포함하는 Parser 클래스의 .py file

TokenType.py - 토큰 타입들을 상수로 나타낸 클래스의 .py file

main.py

```
import sys
import os
from Parser import Parser

if __name__ == "__main__":
    v = "-v" in sys.argv # -v 옵션 여부
    t = "-t" in sys.argv # -t 테스트용 트리 출력 여부, 변수에 대입할 값 출력 여부
    file = sys.argv[-1] # 파서가 처리할 파일 명

    # 파일이 존재하는지 확인
    if not os.path.exists(file):
        print(f'{file} 파일이 존재하지 않습니다.')
        exit(1)

    #파일 읽기
    with open(file, "r") as in_fp:
        code = in_fp.read()
        if len(code)>=1 and code[-1] == '\n': code = code[:-1] #마지막에 개행문자가 있으면 제거-오류방지
        p = Parser(code, verbose=v, test=t) #verbose: -v 옵션, test: 트리, 변수에 대입할 값 출력
        p.run()
```

커맨드 라인에서 입력된 인자들을 인식한다.

Parser 객체를 생성(Parser 클래스는 Lexer 클래스를 상속받음)한다.

Parsing 과정을 실행(Parser 객체의 run 메소드 실행)한다.

Lexer.py - Lexer Class

- 어휘분석기(lexical analyzer)의 소스 코드는 정수 변수 `next_token`, 문자열 변수 `token_string`, 함수 `lexical()`을 포함하여야 한다. ⇒ 이 처리 조건의 `next_token`, `token_string`, `lexical()`은 Lexer Class의 Member Variable 또는 Member Method로 구현되어있다.

Member Variable list

```
self.token_string = "" # 현재 토큰의 문자열
self.next_token = None # 다음 토큰의 타입
self.before_token = None # 이전 토큰의 타입
self.source = input_source # 입력받은 소스코드
self.index = 0 # 현재 읽고 있는 문자의 인덱스
self.id_cnt = 0 # 각 statement에서의 id, const, op의 개수
self.const_cnt = 0
self.op_cnt = 0
self.verbose = verbose # -v 옵션에 따라 출력 방식 결정
self.symbol_table = {} # 변수의 이름과 값을 저장하는 심볼 테이블
self.is_error = False # 에러가 발생했는지 여부
self.is_warning = False # 경고가 발생했는지 여부
self.id_of_now_stmt = None # 현재 파싱 중인 statement의 LHS
self.now_stmt = "" # 현재 파싱 중인 statement
self.list_message = [] # 에러, 경고 메시지를 저장하는 리스트
```

Member Method list

`__init__(self, input_source, verbose=False)` - member variable들을 선언하고 initial value를 assign한다.

`lexical(self)` - 입력 스트림을 분석하여 하나의 lexeme을 찾아낸 뒤, 그것의 token type을 `next_token`에 저장하고, lexeme 문자열을 `token_string`에 저장한다.

c언어 규칙에 맞지 않는 identifier(직전에 인식된 토큰이 식별자이며 직전토큰 다음 부분이 공백이 아닐 경우에는 식별자 이름 규칙을 따르는 부분까지만 토큰으로 인식이 되고 따르지 않는 부분을 lexer가 아직 인식하지 않은 경우이다.)나 ! @같은 Grammar상 등장할 일이 없는 문자가 등장한 경우에 대한 오류를 처리한다. 에러메시지와 `token_string`에 오류가 있는 부분을 추가하기 위해 소스코드를 한글자씩 탐색하며 오류가 없는 부분까지 이동한다. 이때 입력된 source code에 대해 토큰이 인식된 부분을 버리면서 진행하는 것이 아닌 index를 증가시키며 진행한다. 이 작업이 끝난 이후에는 해당 statement의 파싱을 종료한다.

아래의 함수들은 필요한 경우에 아래의 작업들을 수행한다. (중복되는 내용을 생략하기 위해 먼저 설명)

`next_token` 값이 바뀔 때 마다 -v 옵션이 사용되었을 경우 `token_string`값을 출력한다.

또한 오류나 경고사항을 발견하였을때 `is_error` 또는 `is_warning`을 True로 설정하며 무시하는 토큰이 아니면 `now_stmt`의 뒤에 해당 토큰을 붙인다. 또한 인식한 토큰의 길이만큼 index값을 증가 시킨다.

필요한 경우에는 다음 토큰에 대한 정보가 `next_token`이나 `token_string`에 반영되도록 `lexical()`함수를 호출한다.

각 statement에 있는 상수, 식별자, 연산자의 갯수를 기록하기 위한 `const_cnt`, `id_cnt`, `op_cnt`도 상수, 식별자, 연산자를 발견하였을 경우 무시되어야하는 토큰이 아닐 경우 증가시킨다.

그리고, 오류나 경고 메시지들은 `list_message`에 저장된 이후 한 stament에 대한 메시지들이 한번에 출력된다.

오류 중 선언되지 않은(정상적으로 값이 할당되지 않은) 식별자가 사용된 경우를 제외하고 다른 오류들은 발견될 경우 해당 statement의 파싱을 중지하고 `go_to_next_statement()`함수를 호출한다. 이 경우 오류나 경고 중 Parser에 의해 발견되는 일부 오류나 경고들은 출력되지 않는다.

`detect_EOF(self)` - 파일의 끝을 index와 source의 길이를 기준으로 판단 후 `token_string`을 인식된 식별자로, `next_token`을 `TokenType.END`로 바꾼다. 리턴 값은 파일의 끝을 감지하였을 때 True, 아닐 때 False이다.

`detect_id(self)` - 식별자를 정규표현식을 이용해 인식한 이후 `token_string`을 인식한 식별자로, `next_token`을 `TokenType.IDENT`로 바꾼다. 리턴값은 식별자를 감지하였을 때 True, 아닐 때 False이다.

또한 식별자가 연속해서 나올때(before_token이 IDENT일때)는 첫번째 식별자를 제외한 식별자들을 무시한 이후 경고 메시지를 저장한다. 그리고 식별자 이후에 나온 문자가 공백이나 사칙연산자, 괄호, 세미콜론, assignment 연산자가 아닐 경우

`check_id_with_invalld_char()` 함수를 이용해 뒷쪽의 잘못된 문자열들까지 하나의 연산자로 인식한 이후 잘못된 식별자이름이라는 오류 메시지를 저장한다.

여기서 연속하여 나온 식별자가 인식되었을 경우 뒤에 나온 식별자에 대해서는 정의되었는지 여부는 확인하지 않으며 확인을 진행하지 않으니 해당 에러도 출력되지 않는다. 연산자가 연속하여 나왔을 경우 해당 연산자 이후에 `+-*/()`나 공백이 나오기 전까지는 연속된 연산자로 보고 계속 무시한다.

정의(해당 식별자에 값이 정상적으로 할당됨)되지 않은 식별자가 등장할경우 `symbol_table`에 저장된 해당 식별자에 대한 값은 Unknown이 되며 정상적인 LHS인 식별자일 경우 추후 Parser에 의해 정상적인 십진수 값이 할당될 것이다.

`detect_const(self)` - 상수를 정규표현식을 이용해 인식한 이후 `token_string`을 인식한 상수로, `next_token`을 `TokenType.CONST`로 바꾼다. 리턴 값은 상수를 감지하였을 때 `True`, 아닐 때 `False`이다. 이때 상수는 소수는 포함하지만 음수는 포함하지 않는다. 음수가 포함되었을 경우 음수가 아닌 -연산자와 상수로 인식이 되며 문법에 따른 적절한 오류 또는 경고 메시지가 출력될 것이다.

상수를 감지한 이후 해당 상수가 c언어의 식별자 규칙을 따르지 않는 식별자의 앞부분인지(숫자로 시작하는 식별자인지) 여부를 `check_id_start_with_digit()` 함수를 이용하여 확인한다. 이때 `detect_const()` 함수가 정규표현식으로 인식한 부분이 상수가 아닌 식 자의 분앞부부지 라지 `detect_const()` 함수의 리턴값은 `True`이다. (`lexical()` 함수가 종료될 수 있도록 하기 위함) 다만 `next_token`은 `TokenType.IDENT`로 바꾼다.

또한 상수가 연속해서 나올때(`before_token`이 `CONST`일때)는 첫번째 상수를 제외한 상수들을 무시한 이후 경고 메시지를 저장한다. 소수점이 1개 이상일경우 오류가 아닌 경고 메시지를 출력하며 두번째 소수점 이후는 무시된다.

`detect_two_char_op(self)` - 인덱스를 기준으로 2글자를 슬라이싱하여 `:=`를 인식한 이후 `token_string`을 인식한 `:=`로, `next_token`을 `TokenType.ASSIGN_OP`로 바꾼다. 리턴값은 `:=`를 감지하였을 때 `True`, 아닐 때 `False`이다.

`assign` 연산자 이후에 연산자가 오는 것을 확인하고 오류 메시지를 출력하기 위해 `:=`가 인식된 경우에는 `op_after_assign_op()` 함수를 호출한다.

`detect_one_char_op(self)` - 인덱스를 기준으로 1글자를 슬라이싱하여 1글자 연산자들 인식한 이후 `token_string`을 인식한 연산자로, `next_token`을 인식한 연산자에 맞는 `TokenType`로 바꾼다. 리턴 값은 연산자를 감지하였을 때 `True`, 아닐 때 `False`이다.

해당 함수에서 인식하는 연산자들과 각 연산자들을 나타내는 `TokenType`은 다음과 같다.

: 또는 = `ASSIGN_OP`

; `SEMI_COLON`

+또는- `ADD_OP`

*또는/ `MULT_OP`

(`LEFT_PAREN`

) `RIGHT_PAREN`

(가 인식되었을 경우에는 `before_token`으로 이전 토큰이 식별자인지 확인하고 식별자 다음 왼쪽 괄호가 나왔다는 오류 메시지를 출력하며 해당 `statement`의 파싱을 종료한다.

`assign` 연산자 이후에 연산자가 오는 것을 확인하고 오류 메시지를 출력하기 위해 :나 =가 인식된 경우에는 `op_after_assign_op()` 함수를 호출한다.

: =)를 제외한 나머지 연산자들이 인식되었을 경우에는 `ignore_multiple_op()`를 호출하여 여러 연산자가 연속하여 등장하였을 경우에는 경고 메시지를 출력하고 첫번째 연산자를 제외한 이후에 나온 연산자들은 무시될 수 있도록 한다.

:나 =가 인식되었을 경우에는 `op_after_assign_op()`를 호출하여 assignment operator 이후에 연산자들이 나올 경우 경고 메시지를 출력하고 :나 = 이후에 나온 연산자들은 무시될 수 있도록 한다.

`ignore_blank(self)` - 인덱스가 소스코드의 끝이 아니고 인덱스가 가르키는 문자의 아스키코드가 32이하인 동안 소스코드를 한글자씩 살펴봐서 해당 인덱스의 문자가 공백이면 `now_stmt`에 공백을 추가하고 공백이 아닐경우 소스코드에서 해당 문자를 제거한다.

`ignore_multiple_op(self)` - 인덱스가 소스코드 길이 이내이고 해당 인덱스의 문자가 `+-*/;:=` 중 하나인 동안 인덱스를 증가시키며 문자들을 무시하고 연산자가 연속해서 나왔다는 경고 메시지를 출력한다.

오류가 아닌 경고이므로 해당 `statement`의 파싱을 중지하지는 않는다.

`op_after_assign_op(self)` - `assign_op` 이후에 온 문자가 `+-*/;:=` 중 하나인 동안 인덱스를 증가시키며 문자들을 무시하고 `assign_op` 뒤에 연산자가 나왔다는 오류 메시지를 출력하고 해당 `statement`의 파싱을 중지한다.

`check_id_start_with_dight(self,const)` - 이 함수는 상수가 인식된 직후에 호출된다. 인덱스가 소스코드 길이 이내이고 `+-*/();:=` 중 하나나 공백이 나오기 전까지 지금 인덱스에 해당하는 문자를 `now_stmt`와 `token_string`에 추가하며 인덱스를 증가시킨다. 이후 `token_string`을 `symbol_table`에 키값으로 추가시키고 `value`는 `invalid identifier name`으로 한다. 또한 `next_token`을 `TokenType.CONST`에서 `TokenType.IDENT`로 변경한다.

오류 메시지를 출력하며 오류가 있을땐 `True`를 리턴하고 리턴값이 `True`일때는 `detect_const()` 함수에서 해당 `statment`의 파싱을 중지시킨다.

`check_id_with_invalidd_char(self)` - 이 함수는 식별자가 인식되었으며 해당 식별자 직전 토큰이 식별자가 아닐 때 호출된다. 인덱스가 소스코드 길이 이내이고 `+-*/();:=` 중 하나나 공백이 나오기 전까지 지금 인덱스에 해당하는 문자를 `now_stmt`와 `token_string`에 추가하며 인덱스를 증가시킨다. 이후 `token_string`을 `symbol_table`에 키값으로 추가시키고 `value`는 `invalid identifier name`으로 한다. 이 함수가 인덱스를 증가시키며 작업을 진행하는 것은 식별자에 !, @같은 문법상 등장할 일이 없는 문자가 포함되었음을 의미한다. 식별자가 정규표현

식으로 파싱되었으므로 !, @같은 문자가 식별자 이름에 포함되었을 경우 정규표현식을 통한 파싱 결과는 해당 invalid한 문자 직전까지 이기 때문이다.

오류 메시지를 출력하며 오류가 있을땐 True를 리턴하고 리턴값이 True일때는 detect_id()함수에서 해당 statment의 파싱을 중지시킨다.

go_to_next_statement(self) - 해당 statement의 파싱을 오류로 인해 중지시켰을때를 위한 함수이다. lexical()함수와 동작 방식은 거의 유사하지만 lexical()함수는 토큰 하나를 발견하면 멈추는 반면 이 함수는 해당 statement가 끝날때까지 계속 lexical함수가 수행하는 일을 수행한다.

또한 이 함수가 호출되었다는 건 파싱이 종료되었다는 것이므로 Parser에 의해 출력될 경고 중 세미콜론이 나왔는데 파일의 끝인 경우는 이 함수에서 경고 메시지를 처리한다. 이것을 제외한 일부 Parser에 의해 발견되는 오류는 출력되지 않을 수 있다.

after_invalid_char(self, error) - lexical()이나 go_to_next_statement에서 !나 @같은 문법상 등장할 수 없는 문자가 등장할 경우 호출된다. 해당 잘못된 문자 이후에 이어지는 문법상 허용되지 않는 문자열들을 인자로 받은 error메시지에 포함시켜 반환한다.

오류 메시지에 잘못 입력된 문자열들을 추가시키고 잘못입력된 문자열만큼 인덱스를 증가시키는 작업을 수행한다.

end_of_stmt(self) - 각 statement의 끝에서 호출되는 함수이다.

현재 파싱한 statement에서 오류가 발생했을 경우 LHS에 있는 id의 값을 Unknown으로 설정한다.

-v 옵션이 없을 경우 저장되어있는 오류나 경고 메시지들을 출력하고 오류나 경고가 없다면 (OK)를 출력한다.

출력한 이후에는 print()를 이용해 한줄을 띄워 가독성을 높인다.

다음 statement로 넘어가기 전에 now_stme, id_cnt, const_cnt, op_cnt, is_error, is_warning, list_message를 초기화한다.

아래는 Lexer.py의 코드이다.

```
from TokenType import TokenType
import re

class Lexer:
    def __init__(self, input_source, verbose=False):
        self.token_string = "" # 현재 토큰의 문자열
        self.next_token = None # 다음 토큰의 타입
        self.before_token = None # 이전 토큰의 타입
        self.source = input_source # 입력받은 소스코드
        self.index = 0 # 현재 읽고 있는 문자의 인덱스
        self.id_cnt = 0 # 각 statement에서의 id, const, op의 개수
        self.const_cnt = 0
        self.op_cnt = 0
        self.verbose = verbose # -v 옵션에 따라 출력 방식 결정
        self.symbol_table = {} # 변수의 이름과 값을 저장하는 심볼 테이블
        self.is_error = False # 에러가 발생했는지 여부
        self.is_warning = False # 경고가 발생했는지 여부
        self.id_of_now_stmt = None # 현재 파싱 중인 statement의 id
        self.now_stmt = "" # 현재 파싱 중인 statement
        self.list_message = [] # 에러, 경고 메시지를 저장하는 리스트

    def lexical(self): # 다음 토큰을 읽어오는 함수
        self.before_token = self.next_token

        self.ignore_blank() #공백 무시

        check = self.detect_EOF() #파일의 끝을 감지
        if check: return

        check = self.detect_id() #식별자를 감지
        if check: return

        check = self.detect_const() #상수를 감지
        if check: return

        check = self.detect_two_cahr_op() #두 글자 연산자를 감지
        if check: return

        check = self.detect_one_char_op() #한 글자 연산자를 감지
        if check: return

        # ! @ 같은 이상한 문자가 포함되었을때
        # c언어 식별자 규칙에 맞지 않을 때
        tmp_flag = False #위의 두 경우를 구분하기 위한 임시 플래그
        if self.before_token == TokenType.IDENT and self.source[self.index - 1] != " ":
            #식별자에 허용되지 않은 문자가 포함되었을 때 - error
            #직전 토큰이 식별자로 인식되었고 그 식별자 다음 공백이 존재하지 않으며 식별자 다음 온 문자가 연산자와 같은 문법상 가능한 문자가 아닌 경우
            error = "(Error) Unknown token - invalid identifier(Does not follow the identifier name rules for language c) (" + self.token_string + ")"
            self.next_token = TokenType.IDENT #식별자로 인식
```

```

else:
    # ! @ 같은 이상한 문자가 포함되었을때
    tmp_flag = True
    error = "(Error) Unknown token - invalid character(!, @, etc.) or string with invalid character ("
    self.next_token = TokenType.UNKNOWN #식별자가 아닌 경우 UNKNOWN으로 설정
    while self.index < len(self.source) and self.source[self.index] not in "+-*/();:= ": #연산자나 공백이 나올 때까지 invalid한 문자열이나
        error += self.source[self.index]
        self.token_string += self.source[self.index]
        if self.verbose: print(self.token_string)
        self.now_stmt += self.source[self.index]
        self.index += 1
    if tmp_flag: error = self.after_invalid_char(error)
    error += ")"
    self.list_message.append(error)
    self.is_error = True
    self.go_to_next_statement()
    if self.next_token != TokenType.SEMI_COLON: self.lexical()
def detect_EOF(self): # 파일의 끝을 감지하는 함수
    if self.index >= len(self.source):
        self.token_string = "EOF"
        self.next_token = TokenType.END
        if self.verbose: print(self.token_string)
        return True
    else:
        return False

def detect_id(self): # 식별자를 감지하는 함수
    ident_match = re.match(r"[a-zA-Z_][a-zA-Z0-9_]*", self.source[self.index:]) #정규표현식으로 식별자를 감지
    if ident_match: #식별자가 나왔을 때
        self.token_string = ident_match.group() #token_string에 식별자 문자열 저장
        if self.before_token == TokenType.IDENT: # 식별자가 연속해서 나올 때 - warning
            # 식별자가 연속해서 나올 때 - warning
            warning = "(Warning) Continuous identifiers - ignoring identifiers"+"("

        if self.index < len(self.source) and self.source[self.index] not in "+-*/();:= ": #식별자가 끝날때 까지
            while self.index < len(self.source) and self.source[self.index] not in "+-*/();:= ":
                warning += self.source[self.index]
                self.index += 1

        warning += ")" #warning 메시지 저장

        self.list_message.append(warning)
        self.is_warning = True #warning 발생 여부 플래그 설정
        # 뒤에 나온 식별자는 무시
        # 뒤에 나온 식별자가 정의되었는지 여부는 확인하지 않음 - 해당 에러도 출력하지 않음
        self.index += len(self.token_string)
        self.ignore_blank()
        self.lexical()
        return True
    else: # 식별자가 연속해서 나오지 않을 때
        self.next_token = TokenType.IDENT #다음 토큰을 식별자로 설정

        if (self.verbose): print(self.token_string) #verbose 옵션에 따라 출력
        self.now_stmt += self.token_string #현재 파싱 중인 statement에 식별자 추가

        self.index += len(self.token_string) #인덱스 증가

        if self.check_id_with_invalld_char(): #식별자에 허용되지 않은 문자가 포함되었을 때 - error
            self.go_to_next_statement()
            if self.next_token != TokenType.SEMI_COLON: self.lexical()
        else:
            self.id_cnt += 1

        # 식별자가 정의되지 않았을 때 "Unknown"으로 설정
        if self.token_string not in self.symbol_table and not self.is_error: self.symbol_table[self.token_string] = "Unknown"

        return True
    else:
        return False

def detect_const(self): # 상수를 감지하는 함수
    const_match = re.match(r'[-?]\d+(\.\d+)?', self.source[self.index:]) #정규표현식으로 상수를 감지
    if const_match: #상수가 나왔을 때
        self.token_string = const_match.group() #token_string에 상수 문자열 저장

        if self.check_id_start_with_digit(self.token_string): #식별자가 숫자로 시작하는지 확인
            # 식별자가 숫자로 시작하면 - error
            # 오류 메시지는 check_id_start_with_digit()에서 저장
            self.go_to_next_statement() #다음 statement로 이동
            if self.next_token != TokenType.SEMI_COLON: self.lexical() #세미콜론이 아니면 다음 토큰을 읽어옴
            return True

        if self.before_token == TokenType.CONST: #상수가 연속해서 나올 때 - warning

```

```

        warning = "(Warning) Continuous constants - ignoring constants"+"("+self.token_string+)" #warning 메시지 저장
        self.list_message.append(warning) #warning 메시지 저장
        self.is_warning = True
        # 뒤에 나온 상수는 무시
        self.index += len(self.token_string)
        self.ignore_blank()
        self.lexical()
        return True
    else:#상수가 연속해서 나오지 않을 때
        self.next_token = TokenType.CONST #다음 토큰을 상수로 설정

        if (self.verbose): print(self.token_string) #verbose 옵션에 따라 출력
        self.now_stmt += self.token_string #현재 파싱 중인 statement에 상수 추가

        self.index += len(self.token_string) #인덱스 증가
        self.const_cnt += 1 #상수 개수 증가
        if self.index < len(self.source) and self.source[self.index] == "." and "." in self.token_string:
            # 소수점이 여러개일 때 - warning
            # 두번째 소수점 이후 - 앞으로 파싱할 부분이므로 수정 가능
            # 두번째 소수점 이하는 무시
            warning = "(Warning) Multiple decimal points - ignoring decimal points and digits after the second decimal point("
            self.is_warning = True
            while self.index < len(self.source) and (
                self.source[self.index] == "." or self.source[self.index].isdigit()):
                warning += self.source[self.index]
                self.index += 1
            warning += ")"
            self.list_message.append(warning)
            self.ignore_blank()
        return True
    else:
        return False

def detect_two_cahr_op(self): # 두 글자 연산자를 감지하는 함수
    two_char_op = self.source[self.index:self.index + 2] #두 글자 연산자를 감지
    if two_char_op == "!=": #!=가 나왔을 때
        self.token_string = two_char_op #token_string에 != 저장
        self.next_token = TokenType.ASSIGN_OP #다음 토큰을 ASSIGN_OP으로 설정

        if (self.verbose): print(self.token_string) #verbose 옵션에 따라 출력
        self.now_stmt += self.token_string #현재 파싱 중인 statement에 != 추가

        self.index += 2 #인덱스 증가

        self.op_after_assign_op() #:= 다음에 연산자가 나올 때 - error
        return True
    else:
        return False

def detect_one_char_op(self): # 한 글자 연산자를 감지하는 함수
    one_char_op = self.source[self.index] #한 글자 연산자를 감지

    if one_char_op in "+-*/();:=": #연산자가 나왔을 때
        self.token_string = one_char_op #token_string에 연산자 저장
        self.index += 1 #인덱스 증가
        if one_char_op == "+": #연산자에 따라 다음 토큰 설정
            self.next_token = TokenType.ADD_OP #다음 토큰을 ADD_OP으로 설정

            if (self.verbose): print(self.token_string) #verbose 옵션에 따라 출력
            self.now_stmt += self.token_string #현재 파싱 중인 statement에 연산자 추가

            self.op_cnt += 1 #연산자 개수 증가

            self.ignore_multiple_op() #연산자가 여러개 연속해서 나올 때 - warning
        elif one_char_op == "-": #연산자에 따라 다음 토큰 설정
            self.next_token = TokenType.ADD_OP #다음 토큰을 ADD_OP으로 설정

            if (self.verbose): print(self.token_string) #verbose 옵션에 따라 출력
            self.now_stmt += self.token_string #현재 파싱 중인 statement에 연산자 추가

            self.op_cnt += 1 #연산자 개수 증가

            self.ignore_multiple_op() #연산자가 여러개 연속해서 나올 때 - warning
        elif one_char_op == "*": #연산자에 따라 다음 토큰 설정
            self.next_token = TokenType.MULT_OP #다음 토큰을 MULT_OP으로 설정

            if (self.verbose): print(self.token_string) #verbose 옵션에 따라 출력
            self.now_stmt += self.token_string #현재 파싱 중인 statement에 연산자 추가

            self.op_cnt += 1 #연산자 개수 증가

            self.ignore_multiple_op() #연산자가 여러개 연속해서 나올 때 - warning
        elif one_char_op == "/": #연산자에 따라 다음 토큰 설정
            self.next_token = TokenType.MULT_OP #다음 토큰을 MULT_OP으로 설정

```

```

        if (self.verbose): print(self.token_string) #verbose 옵션에 따라 출력
        self.now_stmt += self.token_string #현재 파싱 중인 statement에 연산자 추가

        self.op_cnt += 1 #연산자 개수 증가

        self.ignore_multiple_op() #연산자가 여러개 연속해서 나올 때 - warning
    elif one_char_op == ";": # 연산자에 따라 다음 토큰 설정
        self.next_token = TokenType.SEMI_COLON #다음 토큰을 SEMI_COLON으로 설정

        if (self.verbose and self.index < len(self.source)): print(self.token_string) # verbose 옵션에 따라 출력
        self.now_stmt += self.token_string #현재 파싱 중인 statement에 연산자 추가

    elif one_char_op == "(": # 연산자에 따라 다음 토큰 설정
        self.next_token = TokenType.LEFT_PAREN #다음 토큰을 LEFT_PAREN으로 설정

        if (self.verbose): print(self.token_string) #verbose 옵션에 따라 출력
        self.now_stmt += self.token_string #현재 파싱 중인 statement에 연산자 추가

        if self.before_token == TokenType.IDENT: # 식별자 다음에 (가 나올 때 - error
            error = "(Error) There is left parenthesis after identifier" #error 메시지 저장
            self.list_message.append(error) #error 메시지 저장
            self.is_error = True #error 발생 여부 플래그 설정

            self.ignore_multiple_op() #연산자가 여러개 연속해서 나올 때 - warning
            self.go_to_next_statement() #다음 statement로 이동
            if self.next_token != TokenType.SEMI_COLON: self.lexical() #세미콜론이 아니면 다음 토큰을 읽어옴
            return True

        self.ignore_multiple_op() #연산자가 여러개 연속해서 나올 때 - warning
    elif one_char_op == ")": # 연산자에 따라 다음 토큰 설정
        self.next_token = TokenType.RIGHT_PAREN #다음 토큰을 RIGHT_PAREN으로 설정

        if (self.verbose): print(self.token_string) #verbose 옵션에 따라 출력
        self.now_stmt += self.token_string #현재 파싱 중인 statement에 연산자 추가

        self.ignore_blank() #공백 무시
    elif one_char_op == "=" or one_char_op == "(": # =나 :가 나올 때
        self.next_token = TokenType.ASSIGN_OP #다음 토큰을 ASSIGN_OP으로 설정
        # :=를 =로 쓰 경우 - warning
        # :=를 :로 쓰 경우 - warning
        # :=로 썼다고 가정하고 계속 진행
        self.token_string = ":@"

        if (self.verbose): print(self.token_string)
        self.now_stmt += self.token_string

        if one_char_op == "=": # :=를 =로 쓰 경우 - warning
            warning = "(Warning) Using = instead of := ==> assuming :="
            self.list_message.append(warning)
        elif one_char_op == "(": # :=를 :로 쓰 경우 - warning
            warning = "(Warning) Using : instead of := ==> assuming :="
            self.list_message.append(warning)
        self.is_warning = True

        self.op_after_assign_op() #:= 다음에 연산자가 나올 때 - error
        return True
    else:
        return False

def ignore_blank(self):
    while self.index < len(self.source) and ord(self.source[self.index]) <= 32:
        # 공백이면 self.index를 1 증가시키고 self.now_stmt에 공백 추가
        # 공백이 아니면 self.source에서 self.source[self.index]를 제거
        if self.source[self.index] == " ":
            self.now_stmt += " "
            self.index += 1
        else:
            self.source = self.source[:self.index] + self.source[self.index + 1:]

def ignore_multiple_op(self): # 연산자가 여러개 연속해서 나올 때 - warning
    self.ignore_blank()
    if self.index < len(self.source) and self.source[self.index] in "+-*/(=)":
        # 연산자가 여러개 연속해서 나올 때 - warning
        # 다음에는 당연히 연산자가 나올 수 있으므로 )가 아닐 때만 경고
        warning = "(Warning) Using multiple operators(operator or left_paren) ==> ignoring multiple operators except the first one"
        self.is_warning = True
        while self.index < len(self.source) and self.source[self.index] in "+-*/(=)":
            self.index += 1
            warning += self.source[self.index - 1]

        # ignore_blank() 대응
        while self.index < len(self.source) and ord(self.source[self.index]) <= 32:
            self.now_stmt += " "

```

```

        warning += " "
        self.index += 1
    warning += ")"
    self.list_message.append(warning)

def op_after_assign_op(self): #:= 다음에 연산자가 나올 때 - error
    self.ignore_blank()
    if self.index < len(self.source) and self.source[self.index] in "+-*/(,;:=)":
        # 대입 연산자 이후 다른 연산자가 나올때 - error
        error = "(Error) Operator(operator or right_paren, semi_colon, assign_op) after assignment operator"
        self.list_message.append(error)
        self.is_error = True
        self.go_to_next_statement()
        if self.next_token != TokenType.SEMI_COLON: self.lexical()
        return True
    else:
        return False

def check_id_start_with_digit(self, const): #식별자가 숫자로 시작하는지 확인하는 함수
    # 식별자가 숫자로 시작하면 - error
    if self.index + len(const) < len(self.source) and self.source[self.index + len(const)] not in "+-*/(,;:= ":
        self.now_stmt += const
        self.token_string = const
        self.index += len(const)
        error = "(Error) Identifier starts with digit (" + const
        while self.index < len(self.source) and self.source[self.index] not in "+-*/(,;:= ":
            error += self.source[self.index]
            self.now_stmt += self.source[self.index]
            self.token_string += self.source[self.index]

        if self.verbose: print(self.token_string)
        self.index += 1
        error += ")"
        self.list_message.append(error)
        self.is_error = True

        self.id_cnt += 1
        self.symbol_table[self.token_string] = "invalid identifier name"
        self.next_token = TokenType.IDENT
        return True
    else:
        return False

def check_id_with_invalild_char(self): #식별자에 허용되지 않은 문자가 포함되었는지 확인하는 함수
    # 식별자에 허용되지 않은 문자가 포함되었을 때 - error
    if self.index < len(self.source) and self.source[self.index] not in "+-*/(,;:= ":
        while self.index < len(self.source) and self.source[self.index] not in "+-*/(,;:= ":
            self.now_stmt += self.source[self.index]
            self.token_string += self.source[self.index]
            if self.verbose: print(self.token_string)
            self.index += 1
        self.is_error = True

        self.symbol_table[self.token_string] = "invalid identifier name"
        error = "(Error) Unknown token - invalid identifier(Does not follow the identifier name rules for language c) (" + self.token_string + ")"
        self.list_message.append(error)
        self.next_token = TokenType.IDENT
        self.id_cnt += 1
        return True
    else:
        return False

def go_to_next_statement(self): # 다음 statement로 이동 - error발생시 파싱을 계속 할수 없으므로 lexer를 변형한 이 함수를 사용
    while self.index < len(self.source) and self.next_token != TokenType.SEMI_COLON and self.next_token != TokenType.END:
        self.before_token = self.next_token
        self.ignore_blank() # 공백 무시
        check = self.detect_EOF() # 파일의 끝을 감지
        if check: continue

        check = self.detect_id() # 식별자를 감지
        if check:
            # 선언되지 않은 식별자가 나왔을 때 - error
            # 원래는 파서가 발견해야하는 오류 이지만 이미 앞에서 오류가 발생하여 파싱이 중단되었을 경우에는 Lexer가 발견해야함
            if self.token_string not in self.symbol_table:
                error = "(Error) Using undeclared identifier(" + self.token_string + ")"
                self.list_message.append(error)
                self.is_error = True
            continue

        check = self.detect_const() # 상수를 감지
        if check: continue

        check = self.detect_two_cahr_op() # 두 글자 연산자를 감지
        if check: continue

```



```

        check = self.detect_one_char_op() # 한 글자 연산자를 감지
        if check: continue

        # ! @ 같은 이상한 문자가 포함되었을때
        # c언어 식별자 규칙에 맞지 않을 때
        tmp_flag = False
        if self.before_token == TokenType.IDENT and self.source[self.index - 1] != " ":
            error = "(Error) Unknown token - invalid identifier(Does not follow the identifier name rules for language c) (" + self
            self.next_token = TokenType.IDENT
        else:
            tmp_flag = True
            error = "(Error) Unknown token - invalid character(!, @, etc.) or string with invalid character ("
            self.next_token = TokenType.UNKNOWN
        while self.index < len(self.source) and self.source[self.index] not in "+-*/();:= " :
            error += self.source[self.index]
            self.now_stmt += self.source[self.index]
            self.token_string += self.source[self.index]
            if self.verbose: print(self.token_string)
            self.index += 1
        if tmp_flag: error = self.after_invalid_char(error)
        else:
            self.symbol_table[self.token_string] = "invalid identifier name"
            self.id_cnt += 1
            error += ")"
            self.list_message.append(error)
            self.is_error = True

        if self.next_token == TokenType.SEMI_COLON and self.index == len(self.source):
            # 세미콜론이 나왔는데 파일의 끝이면 - warning
            warning = "(Warning) There is semicolon at the end of the program ==> ignoring semicolon"
            self.now_stmt = self.now_stmt[:-1]
            self.list_message.append(warning)
            self.is_warning = True
            self.index += 1
    def after_invalid_char(self, error):
        while self.index < len(self.source) and self.source[self.index] not in "+-*/();:= " :
            error += self.source[self.index]
            self.token_string += self.source[self.index]
            if self.verbose: print(self.token_string)
            self.now_stmt += self.source[self.index]
            self.index += 1

        if self.index >= len(self.source):
            return error

        if self.source[self.index] not in "+-*/();:= " :
            error += self.source[self.index]
            self.token_string += self.source[self.index]
            if self.verbose: print(self.token_string)
            self.now_stmt += self.source[self.index]
            self.index += 1

        return error
    def end_of_stmt(self):# statement마다 실행되는 함수

        if self.is_error and self.id_of_now_stmt in self.symbol_table and self.symbol_table[self.id_of_now_stmt] != "invalid identifier":
            # 에러가 발생한 경우 - 해당 statement의 id를 Unknown으로 설정
            self.symbol_table[self.id_of_now_stmt] = "Unknown"

        if not self.verbose:#-v 옵션 없을 때
            print(self.now_stmt)#현재 파싱한 statement 출력
            # -v 옵션 없을 때
            # ex) ID: 2; CONST: 1; OP: 1;
            print(f"ID: {self.id_cnt}; CONST: {self.const_cnt}; OP: {self.op_cnt};")

        for i in self.list_message:#에러, 경고 메시지 출력
            print(i)

        if self.is_error == True or self.is_warning == True:
            print()

        if self.is_warning == False and self.is_error == False: # 에러, 경고가 없을 때
            if not self.verbose: print("(OK)\n") #OK 출력

        # 다음 statement로 넘어가기 전에 now_stme, id_cnt, const_cnt, op_cnt, is_error, is_warning, list_message 초기화
        self.now_stmt = ""
        self.id_cnt, self.const_cnt, self.op_cnt = 0, 0, 0
        self.is_error, self.is_warning, self.before_token = False, False, None
        self.list_message = []

```

Parser.py - Parser Class

Lexer Class를 상속받음

파싱과정에서 파싱 트리를 생성한다.(-t 옵션으로 트리를 출력할 수는 있으나 해당 트리가 파싱트리 그 자체는 아니다. 파싱과정에서 필요에 따라 일부분을 수정한다.)

Member Variable list - 나머지 변수들은 부모 클래스로 부터 상속받음

```
self.test = test # 파싱이 정상적으로 되었는지 확인하기 위한 트리 출력, 변수에 대입할 값이 제대로 계산되었는지 확인
```

Member Method list -나머지 함수들은 부모 클래스로 부터 상속받음

`__init__(self, input_source, verbose=False, test=False)` - Lexer클래스의 생성자를 호출하고 입력받은 소스코드가 공백일경우 문법이 공백을 생성할 수 없으므로 오류를 출력한다.

`syntax_error(self)` - syntax에러가 있을 때 호출되며 오류메시지를 출력하고 해당 statement의 파싱을 종료한다.

`run(self)` - Parser객체가 파싱을 시작하게 하는 함수이다. `lexical()` 함수를 호출하여 첫번째 토큰을 읽은 이후 `program()`을 호출한다.

`program()`이 종료된 이후에는 -t 옵션에 따라 필요한 경우 테스트용 트리를 출력한다. 또한, -v 옵션이 없을때는 식별자 별로 할당된 값을 출력한다.

아래 함수들은 각 non-terminal들에 대응하는 함수들이다. `program()`을 제외한 모든 함수는 파라미터로 받은 parent가 부모노드이고 타 입은 함수명을 대문자로 바꾼 것인 노드를 생성한다. 이때 만든 노드는 해당 함수의 반환값이 된다.

또한 문법에서 RHS부분에 주어진 소스 코드가 부합하는지 확인하는 작업을 하며 파싱을 진행한다.

`factor(self, parent=None)`

아래는 문법 중 <factor>가 LHS인 부분이다.

```
<factor> → <left_paren><expression><right_paren> | <ident> | <const>
```

왼쪽 괄호가 나올경우 `expression()`을 호출한 이후 오른쪽 괄호가 있는지 확인한다. 이때 오른쪽 괄호가 없을 경우 경고메시지를 저장한 이후 `now_stmt`에 오른쪽괄호를 추가한다.

문법에서 <left_paren><expression><right_paren> 경우가 아닐 경우 상수나 식별자가 나오는지 확인한다. 상수나 식별자가 나올경우 타 입은 IDENT또는 CONST이고 value는 해당 토큰의 문자열, 부모는 factor노드인 노드를 생성한다.

문법에서 3경우 모두 해당되지 않을 경우 `syntax_error`를 호출하여 오류를 처리한다.

`factor_tail(self, parent=None)`

아래는 문법 중 <factor_tail>가 LHS인 부분이다.

```
<factor_tail> → <mult_op><factor><factor_tail> | ε
```

`next_token`이 `mult_operator`인 동안 해당 연산자에 대한 노드를 생성하고 부모는 `factor_tail`노드로 하며 `factor()`를 호출한다.

`term(self, parent=None)`

아래는 문법 중 <term>가 LHS인 부분이다.

```
<term> → <factor> <factor_tail>
```

`factor()`와 `factor_tail()`을 차례로 호출한다.

`term_tail(self, parent=None)`

아래는 문법 중 <term_tail>가 LHS인 부분이다.

```
<term_tail> → <add_op><term><term_tail> | ε
```

`next_token`이 `add_operator`인 동안 해당 연산자에 대한 노드를 생성하고 부모는 `term_tail`노드로 하며 `term()`를 호출한다.

expression(self, parent=None)

아래는 문법 중 <expression>가 LHS인 부분이다.

```
<expression> → <term><term_tail>
```

term()과 term_tail()을 호출한다. 이후 expression노드를 루트노드로 하는 파싱트리의 서브트리를 전위순회하며 각 노드들의 value를 리스트에 담는다.

또한 term 변수에 빈 문자열을 assign한다.

이 리스트를 순회하며 다음 작업을 수행한다.

숫자나 사칙연산자, 괄호, 정상적으로 선언된 식별자면 term에 더하고 "invalid identifier name" 이 값으로 저장된 식별자면 이미 다른 함수에서 오류메시지를 저장했으므로 넘어가며 정의되지 않은 식별자일 경우 오류메시지를 출력한다. 이때 syntax에러가 아닌 semantic에러이므로 해당 statement의 파싱을 종료하지는 않는다.

이후 term에 문자열로 저장된 식을 계산을 시도하고 계산이 가능하면 expression노드와 계산결과를, 가능하지 않으면 expression노드와 "Unknown"을 반환한다.

이때 계산이후에 expression 노드의 value는 계산 결과가 된다.

statement(self, parent=None)

아래는 문법 중 <statement>가 LHS인 부분이다.

```
<statement> → <ident><assignment_op><expression>
```

next_token이 식별자인지 확인하고 식별자면 id_of_now_stmt를 token_string으로 설정한다. 현재 statement의 LHS에 있는 식별자가 symbol_table에 추가되지 않았으면 추가하고 값은 "Unknown"으로 한다. (정상적인 구문일 경우 추후 expression()이 반환한 값으로 바뀔 것이다.)

이후 :=가 등장한지 확인하고 등장했을 경우에는 op_after_assign_op()함수를 호출해 :=뒤에 연산자가 있는지 확인하고 있을 경우 오류 출력후 해당 statement의 파싱을 종료한다. :=가 등장하지 않았을 경우 assignment 연산자가 없다는 오류를 출력하고 해당 statement의 파싱을 종료한다.

오류가 없었을 경우 assign_op 노드를 만들고 expression()을 호출한 뒤 ident에 대해 symbol_table에 저장된 값을 수정한다.

statements(self, parent=None)

아래는 문법 중 <statements>가 LHS인 부분이다.

```
<statements> → <statement> | <statement><semi_colon><statements>
```

next_token이 TokenType.END이 아닐 때 까지 아래의 작업들을 반복한다.

statement()를 호출하고 next_token이 세미콜론이면 소스코드의 끝에 세미콜론이 나온것인지 확인하고 소스코드 끝에 세미콜론이 나왔을 경우 경고 메시지를 출력후 세미콜론을 무시한다.(위 문법에서 알 수 있듯 statements의 끝은 세미콜론이 될 수 없다.) 이후 가장 최근에 파싱된 statement에 대한 정보들을 출력하기 위해 end_of_stmt()를 호출한다.

next_token이 TokenType.END이면 end_of_stmt()를 호출한다.이후 statements노드를 리턴한다.

왼쪽괄호가 없이 오른쪽 괄호가 나왔을 경우에는 statements()에서 statement를 호출한 이후 token_string이)가 된다. 이경우에는 오류 메시지를 출력하고 (없이)가 나온 statement에 남은 부분들을 go_to_next_statement()를 호출하여 처리한 이후 end_of_stmt()를 호출한다. 이후 statements노드를 리턴한다.

위 경우들에 해당하지 않을 경우 오류가 있다는 것이다. 앞에서 오류 처리가 되었으면 그냥 넘어가고 그렇지 않으면 syntax에러가 있으며 아직 처리되지 않았다는 뜻이므로 syntax_error()를 호출한 이후 end_of_stmt()를 호출하고 statements노드를 리턴한다.

program(self)

아래는 문법 중 <program>가 LHS인 부분이다.

```
<program> -> <statements>
```

이 함수는 인자로 parent를 받지 않으며 statements()를 호출한다.

아래는 Parser.py의 코드이다.

```
from Node import Node
from TokenType import TokenType
from Lexer import Lexer
from anytree import RenderTree
import re

class Parser(Lexer):#파서 클래스
    def __init__(self, input_source, verbose=False, test=False):#파서 생성자
        super().__init__(input_source, verbose=verbose)

        if input_source.replace(" ", "") == "":#입력받은 소스코드가 공백만 있을 때 - error
            error = "(Error) Grammar of this LL(1) parser cannot generate empty source code"
            self.list_message.append(error)
            self.is_error = True
            self.end_of_stmt()
        self.test = test # 파싱이 정상적으로 되었는지 확인하기 위한 트리 출력, 변수에 대입할 값이 제대로 계산되었는지 확인

    def syntax_error(self):
        error = "(Error) Syntax error - invalid token or invalid token sequence or missing token"
        self.list_message.append(error)
        self.is_error = True
        self.go_to_next_statement()
        if self.next_token != TokenType.SEMI_COLON:
            self.lexical()

    def factor(self, parent=None):
        node = Node("FACTOR", parent=parent)
        if self.next_token == TokenType.LEFT_PAREN:
            self.lexical()
            expr_node = self.expression(node)
            if self.next_token != TokenType.RIGHT_PAREN:
                #오른쪽 괄호가 없을 때 - warning
                self.is_warning = True
                warning = "(Warning) Missing right parenthesis ==> assuming right parenthesis at the end of statement"
                self.list_message.append(warning)
                #해당 statement의 맨 오른쪽에 오른쪽 괄호가 있다고 가정하고 계속 진행함
                #LL(1) 파서이므로 오른쪽 괄호가 없다는 것은 이미 파싱한 부분이 아닌 앞으로 파싱할 부분(오른쪽)에 오류가 있다는 것임 - 예러출력 + 계속파싱
                #오른쪽 괄호가 있는 곳은 맨 오른쪽으로 가정, 맨오른쪽==해당 statement의 끝

            if self.next_token == TokenType.SEMI_COLON:
                self.now_stmt = self.now_stmt[:-1] + ";"
                self.token_string = ")"
                if self.verbose: print(self.token_string)
                self.token_string = ";"
                if self.verbose: print(self.token_string)
            else:
                self.now_stmt = self.now_stmt + ")"
                self.token_string = ")"
                if self.verbose: print(self.token_string)
                self.token_string = "EOF"
                if self.verbose: print(self.token_string)
            self.go_to_next_statement()
            return node
        self.lexical()
        elif self.next_token == TokenType.IDENT or self.next_token == TokenType.CONST:
            Node(TokenType.get_name(self.next_token), value=self.token_string, parent=node)
            self.lexical()
        else:
            self.syntax_error()
        return node

    def factor_tail(self, parent=None):
        node = Node("FACTOR_TAIL", parent=parent)
        while self.next_token == TokenType.MULT_OP:
            Node(TokenType.get_name(self.next_token), value=self.token_string, parent=node)
            self.lexical()
            self.factor(node)
        return node

    def term(self, parent=None):
        node = Node("TERM", parent=parent)
        self.factor(node)
        self.factor_tail(node)
        return node
```

```

def term_tail(self, parent=None):
    node = Node("TERM_TAIL", parent=parent)
    while self.next_token == TokenType.ADD_OP:
        Node(TokenType.get_name(self.next_token), value=self.token_string, parent=node)
        self.lexical()
        self.term(node)
    return node

def expression(self, parent=None):
    node = Node("EXPRESSION", parent=parent)
    self.term(node)
    self.term_tail(node)

    RHS = node.preorder()
    term = ""
    for i in RHS:
        if re.fullmatch(r'^\d+$', i):
            term += i
        elif re.fullmatch(r'^\d+\.\d+$', i):
            term += i
        elif re.fullmatch(r'^-\d+$', i):
            term += i
        elif re.fullmatch(r'^-\d+\.\d+$', i):
            term += i
        elif i in "+-*/()":
            term += i
        elif i in self.symbol_table and self.symbol_table[i] != "Unknown":
            term += str(self.symbol_table[i])
        elif i in self.symbol_table and self.symbol_table[i] == "invalid identifier name":
            #잘못된 식별자 이름이 사용된 경우 - error
            #이미 앞에서 에러 처리후 go_to_next_statement()호출되었으므로 다음 statement로 넘어감
            return node
        elif not i in self.symbol_table or self.symbol_table[i] == "Unknown":
            #정의되지 않은 변수 참조 - error - 에러이긴 하지만 syntax error가 아니라 semantic error이므로 파싱은 계속 진행
            error = "(Error) Undefined variable is referenced(" + i + ")"
            self.list_message.append(error)
            self.is_error = True
        else:
            #여기에 걸리는 경우는 없음
            error = "Error: Invalid expression"
            self.list_message.append(error)
            return node, "Unknown"

    try:
        result = eval(term)
        node.value = str(result)
        if(self.test):print(f"Result: {result}")
        return node, result
    except:
        return node, "Unknown"

def statement(self, parent=None):
    node = Node("STATEMENT", parent=parent)
    if self.next_token == TokenType.IDENT:
        self.id_of_now_stmt = self.token_string
        if (self.token_string not in self.symbol_table) or (self.token_string in self.symbol_table and self.symbol_table[self.token_string] != self.token_string):
            Node("IDENT", value=self.token_string, parent=node)
            lhs_id = self.token_string
            self.lexical()
            if self.is_error == True:
                #앞에서 error가 발생한 이후 go_to_next_statment호출 됨. 해당 statment는 파싱이 끝난 상태
                return node
            if self.next_token == TokenType.ASSIGN_OP:
                if self.op_after_assign_op():
                    #오류 메시지는 self.op_after_assign_op()에서 출력
                    self.go_to_next_statement()
                    return node
            else:
                #<statement> -> <ident><assignment_op><expression> 형식이 아닐 때 - error
                error = "(Error) Missing assignment operator"
                self.list_message.append(error)
                self.symbol_table[lhs_id] = "Unknown"
                self.is_error = True
                self.go_to_next_statement()
                if self.next_token != TokenType.SEMI_COLON:self.lexical()
                return node
            Node("ASSIGN_OP", value=self.token_string, parent=node)
            self.lexical()
            if self.is_error == False:
                tmp_node, result = self.expression(node)
                self.symbol_table[lhs_id] = result
            else:
                self.symbol_table[lhs_id] = "Unknown"
    return node

```

```

def statements(self, parent=None):
    node = Node("STATEMENTS", parent=parent)
    while self.next_token != TokenType.END:
        self.statement(node)
        if self.next_token == TokenType.SEMI_COLON: #세미콜론이 나왔을 때
            semi_colon_node = Node("SEMI_COLON", value=self.token_string, parent=node)
            if self.index == len(self.source): # 마지막 statement일 때
                warning = "(Warning) There is semicolon at the end of the program ==> ignoring semicolon"
                self.list_message.append(warning)
                self.is_warning = True
                self.now_stmt = self.now_stmt[:-1]
            if not self.verbose : self.end_of_stmt()
            self.lexical()
        elif self.next_token == TokenType.END:
            if not self.verbose : self.end_of_stmt()
            break
        else:
            if self.token_string == ")": # 왼쪽 괄호가 없을 때 - error
                error = "(Error) Missing left parenthesis"
                self.list_message.append(error)
                self.is_error = True
                self.symbol_table[self.id_of_now_stmt] = "Unknown"
                self.go_to_next_statement()
                if not self.verbose : self.end_of_stmt()
                self.lexical()
                continue
            elif self.is_error == True: #아래의 예러가 이미 앞쪽에서 처리된 경우 - 앞에서 is_error가 True로 바뀌고 go_to_next_statement()를 호출
                continue
            self.syntax_error()
            if not self.verbose : self.end_of_stmt()
            self.lexical()
            continue
        if self.now_stmt != "":
            if not self.verbose: self.end_of_stmt()
    return node

def program(self):
    root = Node("PROGRAM")
    self.statements(root)
    return root

def run(self):
    self.lexical()
    tree = self.program()
    if self.test: #테스트용 트리 출력
        for pre, _, node in RenderTree(tree):
            print(f"{pre}{node}")
    if not self.verbose: # -v 옵션 없을 때 식별자별로 값 출력
        print("Result ==>", end="")
        if len(self.symbol_table) == 0 or (len(self.symbol_table) == 1 and None in self.symbol_table):
            print("There is no identifier")
        for i in self.symbol_table:
            if i != None:
                print(f" {i}: {self.symbol_table[i]}", end=";")
        print()

```

Node.py - Node Class

anytree모듈로 부터 AnyNode클래스를 상속받음

Member Variable List

나머지 Memver Variable들은 부모 클래스로 부터 상속받음

```

self.type = type
self.value = value

```

Member Method List

나머지 함수들은 부모클래스로 부터 상속된다.

`__init__(self, type, value=None, parent=None)` - 노드 클래스의 생성자이며 AnyNode클래스의 생성자를 호출하며 인자로 받은 value 와 parent를 Member Variable에 대입한다.

`__repr__(self)` - 노드들로 이루어진 트리를 출력할때 사용되는 함수이며 `f"{self.value} (Type: {self.type})"`형태의 문자열을 반환한다.

preorder(self):

파싱과정에서 필요한 경우 트리를 전위순회하기 위해 사용되는 함수이며 해당 노드를 루트로 하는 서브트리를 전위순회하며 방문한 노드 순서대로 리스트에 노드들의 value를 담아 반환한다.

이 함수는 expression()에서 식의 값을 계산할 때 사용되며 이미 식의 계산에 사용된 부분은 None으로 바꾼다. expression()에서 계산된 결과는 expression노드의 value에 저장하므로 None으로 바꾸더라도 이후 계산과정이나 파싱과정에서 문제는 없다.

아래는 Node.py의 코드이다.

```
from anytree import AnyNode
class Node(AnyNode): #노드 클래스
    def __init__(self, type, value=None, parent=None): #노드 생성자
        super().__init__(parent=parent)
        self.type = type
        self.value = value

    def __repr__(self): #노드 출력용
        return f"{self.value} (Type: {self.type})"

#서브트리 전위순회
def preorder(self):
    ret = []
    if self.value is not None:
        ret.append(self.value)
        self.value = None
    for child in self.children:
        ret += child.preorder()
    return ret
```

TokenType.py - TokenType Class

각 토큰 타입들과 상수값을 대응시켜둔 클래스이며 상수값으로 부터 해당 토큰이 무슨 타입인지 문자열로 반환해주는 get_name함수도 포함하고 있다.

아래는 TokenType.py의 코드이다.

```
class TokenType: #토큰 타입들을 나타내는 클래스
    UNKNOWN = 0
    IDENT = 1
    CONST = 2
    ASSIGN_OP = 3
    SEMI_COLON = 4
    ADD_OP = 5
    MULT_OP = 6
    LEFT_PAREN = 7
    RIGHT_PAREN = 8
    END = 9
    UNKNOW = 10

    @classmethod
    def get_name(cls, token_type): #토큰 타입을 문자열로 변환 숫자->문자열
        for name, value in cls.__dict__.items():
            if value == token_type:
                return name
        return "UNKNOWN"
```