2023-2 PL Project #1

~Internal Document~

Team 파서만 팠어	
20194198	Min-sik Kim
20222663	Ki-yeong Kweon

1. Lexical Analyzer

```
46    // LexicalAnalyzer
47    void advance();
48    void getNonBlank();
49    string getConst();
50    string getIdent();
51    void lexical();
52
```

```
unordered_map<string, pair<bool, int> > SymbolTable; // name, {isInitial, value}
```

위 함수들은 어휘 분석을 하는 부분으로 각 함수의 기능은 아래와 같다. 심볼 테이블은 STL인 unordered_map으로 구성되어 식별자 이름과 초기화 여부, 값 정보를 저장한다. 각 함수의 기능은 아래와 같다.

advance(): 입력 문자열에서 현재 위치를 다음 위치로 이동시킨다.
getNonBlank(): 입력 문자열에서 현재 위치를 다음 공백문자가 아닌 문자까지 이동시킨다.
getConst(): 입력 문자열에서 상수를 나타내는 문자열을 읽어 반환한다.
getIdent(): 입력 문자열에서 식별자를 나타내는 문자열을 읽어 반환한다.
Lexical(): 입력 문자열을 분석하여 다음 토큰을 분류한다.

2. Syntex Analyzer

```
// SyntaxAnalyzer
void Statements(); // <statements> -> <statement> | <statement><semicolon><statements>
void Statement(); // <statement> -> <ident><assignment_op><expression>
void Expression(); // <expression> -> <term><term_tail>
void Term_tail(); // <term_tail> -> <add/sub><term><term_tail> | lambda
void Term(); // <term> -> <factor><factor_tail>
void Factor_tail(); // <factor_tail> -> <mult/div><factor><factor_tail> | lambda
void Factor(); // <factor> -> <left_paren><expression><right_paren> | <ident> | <const>
```

위 함수들은 실제 구문분석을 실행하는 함수들로, 서로 재귀적으로 연결되어 있어 제시된 문법에 따라 입력 값을 파싱한다. 제일 왼쪽 항부터 순서대로 재귀적 호출을 진행하다 비단말기호를 만나면 return하는 방식으로 문법에 맞게 파싱을 진행할 수 있다.

```
void Term_tail()
{
    pair<bool, int> operand1;
    pair<bool, int> operand2;
    if (next_token == ADD_OP)
    {
        opCnt++;
        lexical();
        Term();
        if(!s.empty())
        {
            operand2 = s.top(); s.pop();
            operand1 = s.top(); s.pop();
            s.push(make_pair(operand1.first & operand2.first, operand1.second + operand2.second));
        }
        Term_tail();
    }
    else if (next_token == SUB_OP)
    {
```

예를 들어 Term_tail() 함수에서는 다음 토큰이 ADD_OP인지 SUB_OP인지에 따라 케이스가 나눠지고 문법에 따라 Term()을 호출한 후 연산 결과를 스택에 삽입하고 다시 Term_tail()을 호출한다. 이렇게 각 함수들은 lexical()을 통해 다음 토큰을 읽어오고 문법에 따라 다음 함수를 호출하는 재귀적 방식으로 연결되어 있다. 각 함수가 처리하는 문법은 아래와 같다.

Statement()

<statements> -> <statement> | <statement><semicolon><statements> 처리

Statement()

<statement> -> <ident><assignment_op><expression> 처리

Expression()

<expression> -> <term><term_tail> 처리

Term_tail()

<term_tail> -> <add/sub><term><term_tail> | lambda 처리

Term()

<term> -> <factor><factor_tail>

Factor_tail()

<factor_tail> -> <mult/div><factor><factor_tail> | lambda 처리

Factor()

<factor> -> <left_paren><expression><right_paren> | <ident> | <const> 처리

3. Error handling

ERROR1 : 연산자(사칙연산)가 연속해서 나오는 경우

토큰이 연산자가 나올 경우 다음 토큰이 연산자가 아닐 때 까지 계속 검사를 하고, 중복된 연산자는 모두 삭제 후 진행한다. 연산자가 나와야 할 위치에서만 이 과정을 진행하는데, 그 이유는 연산자가 나오지 않아야 할 위치에서 연산자 중복이 나온다면 모두 지워야 하는데 하나를 남기고 지우기 때문이다.

ERROR2 : 정의되지 않은 변수(IDENT)를 사용하는 경우

SymbolTable에 없는 변수거나, UNKNOWN 값을 가지고 있는 변수를 사용하는 경우이다. 정의되지 않은 변수와 연산하는 다른 변수들도 모두 UNKNOWN 값을 가지게 된다.

ERROR3 : 괄호가 열렸는데 닫히지 않은 경우

<factor>의 안에서, LEFT_PAREN이 나오고 <expression>을 들어갔다 나왔는데, next_token이 RIGHT_PAREN가 아닌 경우에, 닫는 괄호를 추가하여 진행한다.

ERROR4 : Factor 토큰 오류

<factor>의 안에 들어가면 next_token의 값이 IDENT, CONST, LEFT_PAREN, END_OF_FILE, SEMICOLON 중 하나가 나와야 되는데, 이들 중 하나가 아닐 경우 Factor 토큰 오류로써 처리한다. 토큰을 제거하고 맞는 토큰이 나올 때 까지 다시 lexical을 하면서 <factor>에 들어간다.

ERROR5 : 피연산자(CONST, IDENT)가 연속해서 나오는 경우

첫 번째 피연산자만 남기고 모두 제거한다.

ERROR6 : 대입연산자가 나와야 할 자리에 나오지 않음.

대입연산자가 없으면, 이를 추가하고 파싱을 계속 진행한다.

ERROR7 : Statement 맨 앞에 변수가 나와야 할 자리에 나오지 않음

변수가 하나 나올 때 까지 나오는 모든 토큰을 제거하면서 계속 파싱을 진행한다. IDENT, SEMICOLON, END_OF_FILE 중 하나가 나오기 전 까지 <statement>를 계속해서 재귀적으로 들어간다.

ERROR8 : 의미 없는 문장
(아무것도 없이 ;만 있는 문장, % # 등 의미 없는 문자로만 이루어진 문장 등)

예를 들어, "% $ @;" 또는 ";"와 같이, 선언하지 않았거나 문법에 맞지 않아 모든 토큰이 제거되고, <statement>를 들어가자 마자 next_token이 END_OF_FILE 혹은 SEMICOLON이 나오게 된다면, 아무 의미 없는 문장이므로 이를 처리한다.

ERROR9 : 0으로 나누는 경우

0으로 나누는 것은 불가능하므로 0으로 나눴을 때 그 값을 unknown으로 처리한다. 연산에 필요한 피연산자들은 스택에 푸쉬 되는데, 이 때 초기화 되었는지 안되었는지를 나타내는 bool 값을 false로 바꾼다. 모든 상수들은 bool 값이 true이지만, 0으로 나누었을 경우에만 false로 바뀌게 된다.

ERROR10 : 선언하지 않은 토큰이 등장함

lexical 함수 내에서, 만약 next_token이 UNKNOWN이라면, 선언하지 않은 토큰이 등장한 것이므로 제거 후 UNKNOWN이 나오지 않을 때 까지 lexical을 새로 진행한다.

ERROR11 : 괄호가 열리지 않았는데 닫히는 경우

이 경우에는 괄호의 개수를 세는 변수 parenCnt를 선언해서, LEFT_PAREN가 나올 때는 parenCnt++, RIGHT_PAREN가 나올 때는 parenCnt--을 해 어떤 괄호가 몇 개 더 많은지를 센다. 만약 RIGHT_PAREN이 나오면, parenCnt를 확인해서 parenCnt <= 0 이라면 괄호가 열리지 않았는데 닫힌 괄호가 나온 것이므로 제거한 뒤 lexical을 새로 진행한다.