<프로그래밍 언어 :구현과제1 보고서>

20211397 강동민

1. 과제 개요

본 과제는 주어진 EBNF 문법에 기반한 간단한 언어를 해석하는 **재귀 하강 파서 (Recursive-Descent Parser)**를 C/C++, Python 언어로 구현하는 것이다. 파서는 사용자가 직접 입력한 한 줄의 문장을 분석하고, 문법에 맞게 해석하거나, 구문 오류(Syntax Error)를 탐지한다. 또한, 변수에 대한 할당과 출력 기능을 포함하며, 계산된 표현식의 값을 저장하고 출력하는 기능을 수행한다.

2. 문법 설명 (EBNF)

<program> → {<statement>}
<statement> → <var> = <expr> ; | print <var> ;
<expr> → <term> {+ <term> | * <term>}
<term> → <factor> {- <factor>}
<factor> → [-] (<number> | <var> | '('<expr>')')
<number> → <digit> {<digit>}
<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<var> → <alphabet>{<alphabet>}
<alphabet> → a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

- 산술연산자는 +, -, *가 사용되며 연산자 우선순위는 괄호 > * > +/- 순이다.
- '-' 연산자는 빼기연산 외에도 음수를 표현할 때 쓰이기도 한다.
- <var>은 알파벳 소문자만 허용되며, 알파벳과 숫자의 혼용은 허용하지 않는다.
- print 문을 통해 변수의 값을 출력할 수 있다.

3. 시스템 설계

3.1 Lexer

입력된 문자열을 공백기준으로 나누고, 각각을 토큰으로 분류하여 저장한다. C++구현에서는 구조체를, Python 구현에서는 class를 선언해 '**토큰코드**'와 '**lexeme**'정보를 저장한다.

토큰 코드는 아래와 같이 임의로 구성하였다.

토큰 분류	토큰 종류	코드
숫자 상수	INT_LIT	10
변수	IDENT	11
연산자	ASSIGN_OP	20
	ADD_OP	21
	SUB_OP	22
	MULT_OP	23

괄호	LEFT_PAREN	31
	RIGHT_PAREN	32
세미콜론	SEMICOLON	33
종료표시	END	34
예약어	PRINT	41

3.2 Parser

주어진 BNF룰에 따라 재귀함수로 파싱을 수행하며, 파싱 중 계산 및 변수 저장을 한다.
모든 입력은 줄간에 변수 공유를 하지 않고 독립적으로 동작한다.

파싱에 필요한 주요 함수들은 아래와 같다.

함수명	처리내용
program()	프로그램 시작, statement() 반복
statement()	대입연산 / print문 처리
expr()	'+', '*' 연산 처리
term()	'-' 연산 처리
factor()	숫자, 변수, 괄호, 단항음수 표현

4. 예외 처리 및 제한 사항

- 변수/숫자처리는 최대 10글자 까지 테스트
- 연산자는 공백으로 분리되어 있으며, 공백의 개수는 상관 없다.
- 연속된 부호는 에러처리 ex) ++, +=
- 대입되지 않은 변수의 초기값은 0임

5. 코드 구현 및 분석

5.1 C++ 구현

5.1.1 Lexer

모든 lexeme은 공백으로 구분되어 있으나 공백의 개수에는 제한이 없으므로 lexer의 시작부에 공백을 제거한 뒤 findTokenType()를 통해 토큰 타입을 가져온다.
토큰 타입은 LETTER, DIGIT, UNKNOWN으로 구분되며 이 값들은 한 렉심 내에서 다른 토큰타입을 가진 char을 받을 시 syntax error를 출력하기 위함이다.

```
//공백제거
while (i < len && isspace(input[i])) i++;
if (i >= len) break;

int currentType = findTokenType(input[i]);
```

예를들어 아래의 코드처럼 lexeme의 첫 글자가 a이면 currentType은 LETTER의 토큰 타입을 가지는데 이후에 입력되는 char이 1이면 토큰 타입으로 DIGIT를 반환한다. LETTER != DIGIT이므로 신텍스 에러이다.

```
while (i < len && !isspace(input[i])) {
    int thisType = findTokenType(input[i]);
    if (thisType != currentType) {
        printf("Syntax Error!\n");
        return -1;
    }
    new_token.lexeme[lexIndex++] = input[i++];
}
```

토큰 타입이 UNKNOWN일 경우 symbol을 의미하므로 lookup table을 확인해 처리한다.
이때 ++, +=과 같이 연속적인 연산 기호는 BNF에 정의되어있지 않으므로 에러처리를 한다.
lookup()은 토큰코드를 반환하므로 정상적으로 lexical분석이 이루어지면 token배열에 tokencode와 lexeme이 저장된다.

```
// UNKNOWN → symbol 처리
if (currentType == UNKNOWN) {
    int code = lookup(input[i]);

    // 정의되지 않은 기호
    if (code == -1) {
        printf("Syntax Error!\n");
        return -1;
    }

    // 다음 글자도 공백 없이 붙으면 에러 (예: ++, =+ 등)
    if (i + 1 < len && !isspace(input[i + 1])) {
        printf("Syntax Error!\n");
        return -1;
    }

    // 단독 기호로 처리
    Token new_token;
    new_token.tokenCode = code;
    new_token.lexeme[0] = input[i++];
    new_token.lexeme[1] = '\0';
    token[tokenIndex++] = new_token;
    continue;
}
```

UNKNOWN 심볼 외에도 토큰타입을 기반으로 syntax error를 판단하고 문제가 없을 시 token code와 lexeme을 저장한다.

```
// 토큰코드 매핑
if (currentType == LETTER) {
    if (strcmp(new_token.lexeme, "print") == 0)
        new_token.tokenCode = PRINT;
    else
        new_token.tokenCode = IDENT;  // 일반 식별자도 IDENT
} else if (currentType == DIGIT) {
    new_token.tokenCode = INT_LIT;
} else {
    printf("Syntax Error!\n");
    return -1;
}

token[tokenIndex++] = new_token;
```

5.1.2 Parser

1) program()

```
/* ------------------------------------------------------
 * 설명 : <program> -> {<statement>}
 * ------------------------------------------------------
 */
void program(void){
    symbolTable.clear();
    errorFlag = false;
    printed = 0;
    // 입력인 END가 아니고 에러가 없다면 statement()호출
    while(currentToken().tokenCode != END && !errorFlag){
        statement();
    }
    if(errorFlag){
        printf("Syntax Error!\n");
    }
    else if (printed) {
        if (!errorFlag) {
            for (int val : printBuffer) {
                printf("%d ", val);
            }
            putchar('\n');
        }
        printBuffer.clear(); // 다음 줄 입력 대비
    }
}
```

- 현재 토큰이 END가 아니고 에러가 나지 않았다면 statement()를 반복적으로 호출한다.
반복문을 탈출했을 때 errorFlag == true이면 “Syntax Error!”를 출력한다.
- print가 여러번 호출 되었을 경우를 생각해 printBuffer에 출력하려는 값들을 누적해서 넣어두고 program()이 종료되기 전에 모두 출력한다.
- 입력줄마다 symbolTable, errorFlag, printBuffer를 초기화한다.

2) statement()

문장의 형태는 두 가지로 구분된다:
- print <var> ; 형태인 경우:

```
//print
if(currentToken().tokenCode == PRINT){
    nextToken();
    //print 다음 <var>이 오지 않으면 에러
    if(currentToken().tokenCode != IDENT){
        errorFlag = true;
        return -1;
    }

    string varName = var();
    //print가 ';'로 끝나지 않을 시 에러
    if(expect(SEMICOLON)==-1) return -1;
    printed = 1;
    printBuffer.push_back(symbolTable[varName]);
}
```

print 키워드를 확인한 후, 변수명이 따라오는지 검사한다.
세미콜론(;)으로 문장이 끝나야 하며, printBuffer에 해당 변수 값을 추가한다.

- <var> = <expr> ; 형태인 경우:

```
else if(currentToken().tokenCode == IDENT){
    string varName = var();
    // '='기호 처리
    if(expect(ASSIGN_OP) == -1) return -1;

    //<expr> 처리
    int res_expr = expr();
    if (errorFlag) return -1;

    // ';'기호 처리
    if(expect(SEMICOLON) == -1) return -1;

    //변수에 값 할당
    symbolTable[varName] = res_expr;
}
```

변수명을 먼저 파싱하고, = 연산자가 있는지 확인한다.
이후 expr()을 호출해 계산 결과를 얻은 뒤, symbolTable에 해당 변수 값을 저장한다.

-위의 두 경우 모두 해당하지 않는 경우에는 에러로 판단하고 errorFlag를 true로 설정한다.

```
else{
    errorFlag = true;
    nextToken();
    return -1;
}
```

3) expr()

```
int expr(void){
    int res_term = term();
    if (errorFlag) return -1;

    //'+' or '*'기호 처리
    while(!errorFlag && (currentToken().tokenCode == ADD_OP || currentToken().tokenCode == MULT_OP)){
        int op = currentToken().tokenCode;
        nextToken();
        int val = term();
        if(errorFlag) return -1;
        if(op == ADD_OP) res_term += val;
        else if(op == MULT_OP) res_term *= val;
    }
    return res_term;
}
```

term()을 호출하여 첫 항의 값을 가져온 후, 이후 + 또는 * 연산자가 나오는 동안 반복적으로 term()을 호출하여 계산을 수행한다.
에러가 발생할 경우 -1을 반환하고, 이후 호출 함수에서 errorFlag를 확인하여 처리한다.

4) term()

```
int term(void){
    int res_factor = factor();
    if(errorFlag) return -1;

    // '-'기호 처리
    while(!errorFlag && (currentToken().tokenCode == SUB_OP)){
        nextToken();
        int val = factor();
        if(errorFlag) return -1;

        res_factor -= val;
    }
    return res_factor;
}
```

factor()를 호출하여 값을 얻고, 이후 - 연산자가 나오는 동안 반복적으로 factor()를 호출하여 뺄셈을 수행한다.
뺄셈만 처리하는 이유는 문법상 *과 +는 expr()에서 처리하고, -만 term()에서 다루기 때문이다.
에러 발생 시 -1 반환 후 에러 플래그를 설정한다.

5) factor()
-가 먼저 나올 경우, 단항 음수 부호로 인식하여 sign = -1을 설정하고 다음 항을 처리한다.
--3처럼 연속된 부호가 나오면 Syntax Error로 처리

```
int sign = 1;

if(currentToken().tokenCode == SUB_OP){
    sign = -1;
    nextToken();

    //연속된 - 에러처리
    if (currentToken().tokenCode == SUB_OP){
        errorFlag = true;
        nextToken();
        return -1;
    }
}
```

factor는 다음 중 하나일 수 있음:

정수 리터럴: number() 호출하여 값 반환
변수: var() 호출하여 심볼 테이블에서 값 조회
괄호 식: '(' expr ')' 형태로 파싱 후 expr() 값을 반환
sign을 곱해서 최종 값을 반환

```
// number
if(currentToken().tokenCode == INT_LIT){
    int val = number();
    return sign*val;
}
//var
else if(currentToken().tokenCode == IDENT){
    string varName = var();
    return sign*symbolTable[varName];
}
//괄호
else if(currentToken().tokenCode == LEFT_PAREN){
    nextToken();
    int val = expr();
    if(expect(RIGHT_PAREN) == -1){
        nextToken();
        return -1;
    }
    return sign * val;
}
```

6) alphabet()

알파벳은 소문자만 허용된다.

```
bool alphabet(char c){
    char al[26] = {'a','b','c','d','e','f','g','h',
                   'i','j','k','l','m','n','o','p',
                   'q','r','s','t','u','v','w','x','y','z'};
    for(int i = 0 ; i < 26 ; i++){
        if (c == al[i]){
            return true;
        }
    }
    return false;
}
```

5.2 Python 구현

5.2.1 Lexer

사용자 입력 문자열(input_line)을 받아서 개별 토큰으로 분리하고, token 리스트에 Token 객체로 저장하는 함수이다.

시작 시 공백 문자를 제거한 후, findTokenType()을 통해 현재 문자의 토큰 타입을 판단한다. 가능한 타입은 'LETTER', 'DIGIT', 'UNKNOWN' 세 가지이다.

```
#공백제거
while i < length and input_line[i].isspace():
    i += 1
if i >= length:
    break

c = input_line[i]
currentType = findTokenType(c)
```

UNKNOWN 타입 (기호 문자) 처리
lookup(c) 함수를 호출하여 해당 기호가 정의된 연산자/기호인지 확인한다.
정의되지 않은 기호는 -1을 반환하며, "Syntax Error!"를 출력하고 실패 처리한다.
또한, 기호 문자 다음에 공백 없이 또 다른 문자가 이어질 경우(예: ++, =+)는 문법에 없는 표현이므로 오류로 처리된다.

```
# UNKNOWN -> symbol 처리
if currentType == 'UNKNOWN':
    code = lookup(c)

    #정의되지 않은 기호
    if code == -1:
        print("Syntax Error!")
        return -1

    #다음글자가 공백없이 붙으면 에러(++, =+ 등)
    if i + 1 < length and not input_line[i+1].isspace():
        print("Syntax Error!")
        return -1
    token.append(Token(code, c))
    i += 1
```

LETTER 또는 DIGIT 타입 처리
동일한 타입의 문자가 연속되는 동안 하나의 lexeme으로 묶어 처리한다.
만약 LETTER로 시작한 lexeme에 DIGIT가 섞이는 경우 (예: a1)는 findTokenType()에서 타입이 바뀌므로, 문법 오류로 처리한다.

```
#토큰코드 매핑
    #LETTER일 때 print 따로 처리
    if currentType == 'LETTER':
        code = token_codes['PRINT'] if lexeme == 'print' else token_codes['IDENT']
    #DIGIT는 INT_LIT 토큰
    elif currentType == 'DIGIT':
        code = token_codes['INT_LIT']
    else:
        print("Syntax Error!")
        return -1

    token.append(Token(code, lexeme))
```

완성된 lexeme은 다음과 같이 토큰 코드로 매핑된다:
"print" 문자열은 예약어로 판단되어 PRINT 토큰 코드로 처리
일반 문자 조합은 모두 변수로 간주되어 IDENT 코드
숫자는 INT_LIT 코드로 처리됨

정상적으로 분석된 토큰은 Token 클래스의 인스턴스로 생성되어 token 리스트에 저장된다.
분석이 끝나면 입력 종료를 표시하기 위해 END 토큰이 추가된다.
분석 성공 시 0을 반환하고, 중간에 에러가 발생할 경우 -1을 반환하여 상위 로직에서 판단 가능하게 한다.

5.2.2 Parser

1) program()

```
def program():
    global errorFlag, printed, printBuffer, symbol_table
    symbol_table.clear()
    errorFlag = False
    printed = False

    #입력인 END가 아니고 에러가 없다면 statement()호출
    while currentTokenIndex < len(token) and currentToken().tokenCode != token_codes['END'] and not errorFlag:
        statement()
    if errorFlag:
        print("Syntax Error!")
    elif printed:
        if not errorFlag:
            #버퍼에 있는 값 합쳐서 출력
            print(' '.join(map(str, printBuffer)))
    printBuffer.clear()
```

현재 토큰이 END가 아니고 errorFlag == False일 때 statement()를 반복적으로 호출한다.
반복문을 탈출했을 때 errorFlag == True면 "Syntax Error!"를 출력한다.
printBuffer에 값이 저장되어 있으면 ' '.join()으로 한 줄로 출력한다.
다음 줄의 처리를 위해 printBuffer를 초기화한다.
또한 프로그램 실행 전 symbol_table, errorFlag, printed를 초기화해 독립적인 한 줄 처리를 유지한다.

2) statement()

두 가지 문장 형태를 처리한다:

print <var> ; 형태:

```
#print
if currentToken().tokenCode == token_codes['PRINT']:
    nextToken()
    #print 뒤에 <var>이 아닌경우 error
    if currentToken().tokenCode != token_codes['IDENT']:
        errorFlag = True
        return -1
    varName = var()
    #print가 ;로 끝나지 않으면 error
    if expect(token_codes['SEMICOLON']) == -1:
        return -1
    printed = True
    printBuffer.append(symbol_table.get(varName, 0))
```

print 키워드 뒤에 식별자가 오지 않으면 에러 처리
;로 끝나야 하며, 해당 변수값을 symbol_table에서 꺼내 printBuffer에 추가

<var> = <expr> ; 형태:

```
# <var> = <expr> ;
elif currentToken().tokenCode == token_codes['IDENT']:
    varName = var()
    # '='기호 처리
    if expect(token_codes['ASSIGN_OP']) == -1:
        return -1
    #<expr> 호출
    val = expr()
    if errorFlag:
        return -1

    # ; 처리
    if expect(token_codes['SEMICOLON']) == -1:
        return -1

    # 심볼테이블에 변수 값 할당
    symbol_table[varName] = val
else:
    errorFlag = True
    nextToken()
    return -1
return 0
```

변수명을 먼저 받아오고, = 다음 expr()으로 식 계산
세미콜론이 없으면 에러, 계산 결과를 변수에 저장
위 두 형태가 모두 아닐 경우 errorFlag = True 설정

3) expr()

```
def expr():
    res_term = term()
    if errorFlag:
        return -1
    while not errorFlag and currentToken().tokenCode in [token_codes['ADD_OP'], token_codes['MULT_OP']]:
        op = currentToken().tokenCode
        nextToken()
        val = term()
        if errorFlag:
            return -1
        #ADD_OP이면 더하기
        if op == token_codes['ADD_OP']:
            res_term += val
        #MULT_OP이면 곱하기
        elif op == token_codes['MULT_OP']:
            res_term *= val
    return res_term
```

term()을 먼저 호출하여 값을 계산하고, 이후 + 또는 *가 나올 때마다 term()을 재귀적으로 호출하여 연산을 진행한다.

4) term()

```
def term():
    res_factor = factor()
    if errorFlag:
        return -1

    # - 기호 처리
    while not errorFlag and currentToken().tokenCode == token_codes['SUB_OP']:
        nextToken()
        val = factor()
        if errorFlag:
            return -1
        res_factor -= val
    return res_factor
```

factor()를 호출하여 값을 받아오고, 이후 - 연산자가 연속해서 등장하는 경우마다 다음 factor() 값을 빼는 방식으로 계산한다.
단항 마이너스는 factor() 내부에서 처리하며, 이 함수에서는 이항 뺄셈만 다룬다.

5) factor()
단항 부호 -가 있을 경우 sign = -1로 설정하여 이후 값에 곱해준다.
연속된 - 기호나 괄호 짝이 맞지 않으면 Syntax Error를 발생시킴

```
global errorFlag
sign = 1
if currentToken().tokenCode == token_codes['SUB_OP']:
    sign = -1
    nextToken()

    # 연속된 - 에러 처리 (ex) - - )
    if currentToken().tokenCode == token_codes['SUB_OP']:
        errorFlag = True
        nextToken()
        return -1
```

다음 세 가지 유형 중 하나를 처리:
정수 리터럴: number() 호출
변수명: var() 호출 후 symbol_table에서 값 조회
괄호식: '(' expr ')' 형태로 처리 (재귀적으로 expr() 호출)

```python
# number처리
if currentToken().tokenCode == token_codes['INT_LIT']:
    return sign * number()

# var 처리
elif currentToken().tokenCode == token_codes['IDENT']:
    return sign * symbol_table.get(var(), 0)

# 괄호 처리
elif currentToken().tokenCode == token_codes['LEFT_PAREN']:
    nextToken()
    val = expr()
    if expect(token_codes['RIGHT_PAREN']) == -1:
        nextToken()
        return -1
    return sign * val
else:
    errorFlag = True
    nextToken()
    return -1
```

7. 결론 및 느낀 점

이번 과제를 통해 간단한 언어를 위한 재귀 하강 파서의 구조를 이해하고 직접 구현해 볼 수 있었다. 특히, 어휘 분석과 구문 분석의 명확한 역할 분담, 연산자 우선순위 구현, 심볼 테이블을 통한 변수 관리 등이 중요한 포인트였다. 향후 더 복잡한 문법 확장이나 툴을 이용한 다른 lr파서 등을 추가 구현 해보는 것도 좋은 발전 방향이 될 것이다.

8. 실행화면 스냅샷

8.1 C++

```
>> x = ( 12 + 3 ;
Syntax Error!
>> x = 10 + 5 / 2 ;
Syntax Error!
>> k = 3 ; j = 20 ; print k + j ;
Syntax Error!
>> a = 5 ; b = 3 ;
>> ab = - ( - ( -( -12 ) ) ) ) ;
Syntax Error!
>> ab = - ( - ( -( - 12 ) ) ) ) ;
Syntax Error!
>> ab = - ( - ( - ( - ( - 12 ) ) ) ) ;
>> y = - ( 12 - 3 ) ; print a ;
0
>> b = ( 12 * 2 ) + 30    * 5 ; print b ;
270
>> x = - 12 + 3 ; print x ;
-9
>> x = 10 + 5 - - 2 ; print x ;
17
>> z = 100 * 3 ; y = 42 - 7 ; abc = z - 10 * y + 50 ; print y ; print abc ;
35 10200
>> x = - 12 + 3 - ( 2 + 8 ) ; y = 10 + 5 * 3 ; print y ; print x ;
45 -19
>> y = 10 * 5 - 3 ; xyz = 5 - 2 + - 8 - 2 ; print y ; print xyz ;
20 -7
>>
 * 터미널이 작업에서 다시 사용됩니다. 닫으려면 아무 키나 누르세요.
```

8.2 Python

```
(base) D:\vsCoding>C:/Users/kdm12/anaconda3/python.exe d:/vsCoding/PL_20211397/PL_HW
#01_20211397.py
>>  x = ( 12 + 3 ;
Syntax Error!
>> x = 10 + 5 / 2 ;
Syntax Error!
>> k = 3 ; j = 20 ; print k + j ;
Syntax Error!
>> a = 5 ; b = 3 ;
>> ab = - ( - ( - ( -( - 12 ) ) ) ) ;
Syntax Error!
>> ab = - ( - ( - ( - ( - 12 ) ) ) ) ;
>> y = - ( 12 - 3 ) ; print a ;
0
>> y = - ( 12 - 3 ); print a ;
Syntax Error!
>> b = ( 12 * 2 ) + 30   * 5 ; print b ;
270
>> x = - 12 + 3 ; print x ;
-9
>> x = 10 + 5 - - 2 ; print x ;
17
>> z = 100 * 3 ; y = 42 - 7 ; abc = z - 10 * y + 50 ; print y ; print abc ;
35 10200
>> x = - 12 + 3 - ( 2 + 8 ) ; y = 10 + 5 * 3 ; print y ; print x ;
45 -19
>> y = 10 * 5 - 3 ; xyz = 5 - 2 + - 8 - 2 ; print y ; print xyz ;
20 -7
>>
```