<프로그래밍 언어 :구현과제2 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

```
<program> → {<declaration>} {<statement>}
<declaration> → <type> <var> ;
<statement> → <var> = <aexpr> ; | print <aexpr> ; |
              while ( <bexpr> ) do ' { ' {<statement>} ' } ' ; |
              if ( <bexpr> ) ' { ' {<statement>} ' } ' else ' { ' {<statement>} ' } ' ;
<bexpr> → <var> <relop> <var>
<relop> → == | != | < | >
<aexpr> → <term> {( + | - ) <term>}
<term> → <factor> { * <factor>}
<factor> → [ - ] ( <number> | <var> | '('<aexpr>')' )
<type> → integer
<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
```

<주요 특징>

- 연산자 우선순위는 * 가 + , -보다 높다.
- bexpr의 경우 변수와 변수 사이의 비교만 가능하다. 따라서 a < 5와 같은 숫자와의 비교는 금지된다.
- declaration과 statement부는 구분되어야 한다.
- while문과 if문은 중첩이 가능하다.

3. 주요 구현 함수

전체 코드는 lexer와 parser로 구분되며 lexer에서는 각 토큰별로 정보를 저장하고 parser에서 BNF룰에 따라 재귀함수로 파싱을 수행한다.

<declaration>

```
int declaration(void){
    if(type() <0){
        errorFlag = true;
        return -1;
    }

    // 선언되지 않은 변수인지 확인
    // if (symbolTable.count(currentToken().lexeme) != 0) {
    //     errorFlag = true;
    //     return -1;
    // }
    string varName = var();

    if(expect(SEMICOLON) == -1){
        errorFlag = true;
        return -1;
    }

    symbolTable[varName] = 0;
    return 0;
}
```

선언부에서는 같은 변수에 대해 두 번 이상 선언하여도 bnf상 문제가 없으므로 에러가 나면 안된다.

<while>

2pass로 평가를 하며, 1pass에서는 bexpr의 결과에 상관없이 입력이 bnf 규칙을 따르는지 확인한다. 에러가 있는 경우 Syntax error를 출력하며 그렇지 않은 경우 token index를 변화시키며 while문을 수행한다.

```
else if(currentToken().tokenCode == WHILE){
    nextToken();

    // 1. 기본 구조 파싱
    if(expect(LEFT_PAREN) == -1) return -1;
    int currentWhileConditionStart = currentTokenIndex;

    bexpr();
    if(errorFlag) return -1;

    int tempAfterConditionIndex = currentTokenIndex;
    currentTokenIndex = tempAfterConditionIndex;
    if(expect(RIGHT_PAREN) == -1) return -1;
    if(expect(DO) == -1) return -1;
    if(expect(LBRACE) == -1) return -1;

    int currentWhileBodyStart = currentTokenIndex;

    // 루프 본문을 먼저 한 번 파싱하여 문법 오류 검사
    // 심볼 테이블과 출력 상태 백업
    map<string, long long> backupSymbolTable = symbolTable;
    vector<long long> backupPrintBuffer = printBuffer;
    int backupPrinted = printed;

    // 본문 파싱 (실행은 하지만 결과는 무시)
    while(currentToken().tokenCode != RBRACE && !errorFlag && currentToken().tokenCode != END){
        if(statement() == -1) {
            return -1; // 문법 오류 발견 시 즉시 종료
        }
    }
```

pass1의 과정

```
// 2. 실제 실행 로직
while(true){
    int savedTokenIndexBeforeLoopIteration = currentTokenIndex;
    currentTokenIndex = whileStack.back().conditionStartIndex;
    bool cond = bexpr();
    currentTokenIndex = savedTokenIndexBeforeLoopIteration;

    if (errorFlag) {
        if(!whileStack.empty()) whileStack.pop_back();
        return -1;
    }

    if (!cond) {
        currentTokenIndex = whileStack.back().afterLoopIndex;
        break;
    }

    // 조건이 참일 때 본문 실행
    currentTokenIndex = whileStack.back().bodyStartIndex;
    while(currentToken().tokenCode != RBRACE && !errorFlag && currentToken().tokenCode != END){
        if(statement() == -1) {
            if(!whileStack.empty()) whileStack.pop_back();
            return -1;
        }
    }
    if(errorFlag) {
        if(!whileStack.empty()) whileStack.pop_back();
        return -1;
    }
}
```

pass2의 과정

<IF>
while과 마찬가지로 문법상 오류를 검사하고 에러가 없다면 실제 수행부를 실행한다.

```
// then 블록을 먼저 파싱하여 문법 오류 검사
// 심볼 테이블과 출력 상태 백업
map<string, long long> backupSymbolTable = symbolTable;
vector<long long> backupPrintBuffer = printBuffer;
int backupPrinted = printed;

// then 블록 파싱
while(currentToken().tokenCode != RBRACE && !errorFlag && currentToken().tokenCode != END){
    if(statement() == -1) {
        return -1; // 문법 오류 발견 시 즉시 종료
    }
}
if(errorFlag) return -1;

// 상태 복원
symbolTable = backupSymbolTable;
printBuffer = backupPrintBuffer;
printed = backupPrinted;

int currentIfThenBlockClosingBrace = currentTokenIndex;
if (expect(RBRACE) == -1) return -1;
if (expect(ELSE) == -1) return -1;
if (expect(LBRACE) == -1) return -1;
int currentIfElseBlockStart = currentTokenIndex;
```

pass1

```
// 2. 실제 실행 로직
int savedCurrentTokenIndex = currentTokenIndex;

// 조건 평가
currentTokenIndex = ifElseStack.back().conditionStartIndex;
bool cond = bexpr();
currentTokenIndex = savedCurrentTokenIndex;

if (errorFlag) {
    if(!ifElseStack.empty()) ifElseStack.pop_back();
    return -1;
}

if (cond) { // 조건이 true: then 블록 실행
    currentTokenIndex = ifElseStack.back().thenBlockStartIndex;
    while(currentToken().tokenCode != RBRACE && !errorFlag && currentToken().tokenCode != END){
        if(statement() == -1) return -1;
    }
    if (expect(RBRACE) == -1) return -1;
}
else { // 조건이 false: else 블록 실행
    currentTokenIndex = ifElseStack.back().elseBlockStartIndex;
    while(currentToken().tokenCode != RBRACE && !errorFlag && currentToken().tokenCode != END){
        if(statement() == -1) return -1;
    }
    if (expect(RBRACE) == -1) return -1;
}
```

pass2

while과 if문은 중복될 수 있으므로 stack구조로 body의 시작부 종료부의 token상 index를 저장해둔다. 추가로 조건문의 참 거짓 여부와 관련없이 항상 BRACE내부의 문법을 테스트할 경우 무한루프가도는경우가 생겨 문법체크용 함수와 실제 구현용 함수를 분리하여 구현하였다.

<실행결과 스냅샷>
C++

```
>>  a = 10 ;
Syntax Error!
>> float b ;
Syntax Error!
>> integer k = 2;
Syntax Error!
>> integer k = 2 ;
Syntax Error!
>> integer k ; integer j ; k = 3 ; j = 20 ; integer a ; print k + j ;
Syntax Error!
>> integer k ; integer j ; k = 3 ; j = 20 ; print k > j ;
Syntax Error!
>> integer k ; integer j ; k = 3 ; j = 20 ; while ( k < 5 ) do { print k ; k = k + 1 ; } ;
Syntax Error!
>>  integer k ; integer j ; k = 3 ; j = 20 ; print k + j ;
23
>> integer k ; print k + 100 * 3 - 1 ;
299
>> integer x ; x = 10 + 5 - ( 2 + 3 - 5 * 10 ) ; print x ;
60
>> integer x ; integer y ; x = 10 + 5 * 2 ; y = 20 - 5 ; if ( x < y ) { print x - 5 ; } else { x = 7 ; print x + 5 ; } ;
12
>> integer k ; integer j ; k = 30 ; j = 25 ; while ( k > j ) do { print ( k - j ) * 10 ; k= k – 1 ; } ; print k ;
Syntax Error!
>> integer k ; integer j ; k = 30 ; j = 25 ; while ( k > j ) do { print ( k - j ) * 10 ; k = k – 1 ; } ; print k ;
Syntax Error!
>> integer k ; integer j ; k = 30 ; j = 25 ; while ( k > j ) do { print ( k - j ) * 10 ; k = k - 1 ; } ; print k ;
50 40 30 20 10 25
>> integer i ; integer j ; integer k ; i = 0 ; j = 5 ; k = 3 ; while ( i < j ) do { if ( i < k ) { i = i + 1 ; print i ; } else { i = i + 1 ; } ; } ; print i ; print j * k ;
1 2 3 5 15
```

주어진 pdf 상에 -기호 대신 하이픈(-)이 들어간 케이스가 있던데 bnf 상으로는 -기호만 있어 구현에 포함시키지 않음.

Python

```
(base) D:\vsCoding>C:/Users/kdm12/anaconda3/python.exe d:/vsCoding/PL_20211397/PL_HW#02_20211397.py
>> integer variable ; variable = 365 ;
>>  a = 10 ;
Syntax Error!
>> float b ;
Syntax Error!
>> integer k = 2 ;
Syntax Error!
>> integer k ; integer j ; k = 3 ; j = 20 ; integer a ; print k + j ;
Syntax Error!
>> integer k ; integer j ; k = 3 ; j = 20 ; print k > j ;
Syntax Error!
>> integer k ; integer j ; k = 3 ; j = 20 ; while ( k < 5 ) do { print k ; k = k + 1 ; } ;
Syntax Error!
>> integer k ; integer j ; k = 3 ; j = 20 ; print k + j ;
23
>>  integer k ; print k + 100 * 3 - 1 ;
299
>> integer x ; x = 10 + 5 - ( 2 + 3 - 5 * 10 ) ; print x ;
60
>> integer x ; integer y ; x = 10 + 5 * 2 ; y = 20 - 5 ; if ( x < y ) { print x - 5 ; } else { x = 7 ; print x + 5 ; } ;
12
>> integer k ; integer j ; k = 30 ; j = 25 ; while ( k > j ) do { print ( k - j ) * 10 ; k = k - 1 ; } ; print k ;
50 40 30 20 10 25
>> integer i ; integer j ; integer k ; i = 0 ; j = 5 ; k = 3 ; while ( i < j ) do { if ( i < k ) { i = i + 1 ; print i ; } else { i = i + 1 ; } ; } ; print i ; print j * k ;
1 2 3 5 15
```