프로그래밍 언어(나) 구현과제2

숭실대학교 컴퓨터학부 정해성

2024-05-27

목차

1. 동기 및 목적

본 보고서는 아래와 같은 EBNF로 문법이 정의되는 언어를 해석하고 그 언어의 의미에 따라 동작하는 Parser를 구현하는 과제에 대한 보고서입니다. 저는 저번 과제와 마찬가지로 Recursive-Descent Parser 방식으로 구현하였습니다.

```
<program> → {<declaration>} {<statement>}
<declaration> → <type> <var> ;
<statement> → <var> = <aexpr> ; | print <bexpr> ; | print <aexpr> ; |
              do ' { ' {<statement>} ' } ' while ( <bexpr> ) ;
<bexpr> → <relop> <aexpr> <aexpr>
<relop> → == | != | < | > | <= | >=
<aexpr> → <term> {( + | - | * | / ) <term>}
<term> → <number> | <var> | ( <aexpr> )
<type> → int
<number> → <dec>{<dec>}
<dec> → 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
```

2. 설계 및 구현

Java 언어로 구현하였으며, 각 역할에 따라 `LexicalAnalyzer`와 `RDParser` 클래스를 설계하였습니다. `LexicalAnalyzer`는 입력 문자열의 다음 토큰을 분석하여 분류하는 클래스이고, `RDParser`는 `LexicalAnalyzer`의 `nextToken` 값을 활용하여 파싱 및 실행하는 클래스입니다.

2-1. Lexical Analyzer

지난 설계 과제1에서 구현한 것과 동일합니다. 차이점이 있다면, 반복문을 수행하기 위해 체크포인트를 복원할 수 있는 `setCheckPoint(int curIdx)` 메서드가 추가되었다는 것입니다. 여기서 체크포인트는 입력 문자열의 인덱스를 의미합니다.

토큰 설계

```
enum Token {
    IDENT,          // id (식별자)
    INT_LIT,        // integer literal

    // keyword
    TYPE_INT,       // int
    PRINT,          // print
    DO,             // do
    WHILE,          // while

    // 연산자 & 특수기호
    ASSIGN_OP,      // =
    ADD_OP,         // +
    SUB_OP,         // -
    MULT_OP,        // *
    DIV_OP,         // /
    REL_EQ,         // ==
    REL_NEQ,        // !=
    REL_LT,         // <
    REL_GT,         // >
    REL_LE,         // <=
    REL_GE,         // >=
    SEMICOLON,      // ;
    LEFT_PAREN,     // (
    RIGHT_PAREN,    // )
    LEFT_BRACE,     // {
    RIGHT_BRACE,    // }

    EOF,
    UNKNOWN
}
```

언어가 달라짐에 따라 새로운 토큰이 추가되었습니다. 또한 `print`도 하나의 토큰으로 취급하고 있습니다.

2-2. Recursive-Descent Parser

syntax error 감지

`parse()` 메서드로 먼저 문법에 맞지 않는 입력이 있는지 확인한 후, static semantics를 포함한 문법적인 문제가 없다면 `execute()` 메서드로 입력의 처음부터 다시 보면서 print문, do-while문을 실행합니다. `parse()` 메서드는 중간에 syntax error가 감지되면 Exception을 던지므로 뒤에 `execute()` 메서드가 실행되지 않습니다.

do-while문 실행

`parse()` 메서드와 `execute()` 메서드의 차이는 `exe`라는 내부 플래그 값을 `true` 또는

false로 세팅해주는 것이 전부입니다. exe 플래그는 print문을 만났을 때 콘솔에 출력할지 여부를 제어하며, do-while문을 만났을 때 또한 반복을 수행할지 파싱만 하고 넘어갈지를 판단하는 기준이 됩니다.

반복을 수행하는 과정은 단순히 do 다음에 나오는 중괄호 다음의 인덱스를 기억해둬서 반복을 수행할 것이라면 LexicalAnalyzer의 curIdx 값을 기억해둔 인덱스로 설정해줍니다. LexicalAnalyzer는 기계적으로 curIdx에서 다음 토큰을 반환하기 때문에 반복문 수행이 가능합니다.

main 함수에서 프로그램 시작하는 부분

```
public static void main(String[] args) {
   /* … */
    while (true) {
        System.out.print(">> ");
        String input = sc.nextLine();
        if (input.equals("terminate")) break;

        RDParser parser = new RDParser(input);
        try {
            // 파싱을 먼저 시도한 후 실행합니다.
            parser.parse();
            parser.execute();
        } catch (SyntaxException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

statement()에서 do-while을 수행하는 부분 정의

```
void statement() throws SyntaxException {
    switch (la.nextToken) {
        /* … */
        case DO -> {
            la.lex();
            if (la.nextToken != Token.LEFT_BRACE) throw new SyntaxException();

            // 블록의 시작이 되는 인덱스를 임시로 저장합니다.
            int checkpoint = la.getCheckPoint();
            boolean b;
            do {
                la.lex();
                while (la.nextToken != Token.RIGHT_BRACE)
                    statement();
                la.lex();
                if (la.nextToken != Token.WHILE) throw new SyntaxException();
```

```
                la.lex();
                if (la.nextToken != Token.LEFT_PAREN) throw new SyntaxException();
                la.lex();
                b = bexpr() && exe;
                if (la.nextToken != Token.RIGHT_PAREN) throw new SyntaxException();
                if (b) {
                    // bexpr의 값이 true이면 블록의 시작으로 이동합니다.
                    la.setCheckPoint(checkpoint);
                }
            } while (b);
            la.lex();
        }
    }
    /* … */
}
```

3. 실행 결과

Java

```
>> print ( 1 ) ;
1
>> int abcdabcdabcd ;
Syntax Error!!
>> int a = 1 ; print a ;
Syntax Error!!
>> int abc ; int k ; abc = 100 ; k = 1 + 4 * (1 + 2) ; print abc + k ;
115
>> int var ; print == 1 var + 1 ;
TRUE
>> int a ; int b ; int c ; a = 2 ; b = a + 3 / 2 ; c = a * b ; print a + b + c ;
8
>> int PascalCase ; PascalCase = 1 ;
Syntax Error!!
>> int a ; do { print a ; a = a + 1 ; } while ( < a 5 ) ; print == 1 1 ; print ( a ) ;
0 1 2 3 4 TRUE 5
>> int a ; int b ; int c ; do { print == 1 1 ; b = 0 ; do { print c ; c = c + 1 ; b = b + 1 ; } while ( < b 3 ) ; a = a + 1 ; } while ( < a 5 ) ;
TRUE 0 1 2 TRUE 3 4 5 TRUE 6 7 8 TRUE 9 10 11 TRUE 12 13 14
>> do { print 1 ; } while ( == 1 1 ) ; print x ;
Syntax Error!!
```