프로그래밍 언어(나) 구현과제1

- RD Parser -

숭실대학교 컴퓨터학부 정해성

목차

1. 동기 및 목적

본 보고서는 아래와 같은 EBNF로 문법이 정의되는 언어를 해석하고 그 언어의 의미에 따라 동작하는 Recursive-Descent Parser를 구현하는 과제에 대한 보고서입니다.

```
<program> → {<statement>}
<statement> → <var> = <expr> ; | print <var> ;
<expr> → <bexpr> | <aexpr>
<bexpr> → <number> <relop> <number>
<relop> → == | != | < | > | <= | >=
<aexpr> → <term> {( + | - ) <term>}
<term> → <factor> {( * | / ) <factor>}
<factor> → <number>
<number> → <dec> {<dec>}
<dec> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<var> → x | y | z
```

2. 설계 및 구현

Java 언어로 구현하였으며, 라인을 기준으로 입력을 받고 terminate라는 명령어가 입력되면 프로그램을 종료합니다. 그렇지 않으면 입력에 대해 lexical 분석 및 및 토크나이징, 그리고 파싱을 진행하여 실행 결과를 출력합니다. 입력 문자열을 각각의 lexeme으로 구분하고 토큰 분석하는 부분은 Lexical Analyzer 클래스가 담당하고, RDParser가 Lexical Analyzer 객체를 활용하여 다음 토큰을 확인하는 방식으로 구성하였습니다.

2-1. Lexical Analyzer

Lexical Analyzer의 목적은 다음 lexeme에 대응되는 토큰을 식별하는 것입니다. 구현은 강의 자료에서 소개된 것과 비슷하게 구현하였으며, 다른 점이 있다면 lexeme을 추출할 때 배열에 문자 하나씩 추가하는 것이 아니라 lexeme의 시작 인덱스와 끝나는 인덱스를 찾아 Java String의 substring으로 추출했다는 점입니다. la.lex() 함수를 호출하면 la.nextToken, la.lexeme 값이 다음 lexeme에 해당하는 토큰으로 바뀌며, 토큰 종류는 아래와 같이 구분해주었습니다.

```
enum Token {
    IDENT,      // id (식별자)
    INT_LIT,    // integer literal
    ASSIGN_OP,  // =
    ADD_OP, SUB_OP, MULT_OP, DIV_OP, // +, -, *, /
    REL_EQ, REL_NEQ, REL_LT, REL_GT, REL_LE, REL_GE, // ==, !=, <, >, <=, >=
    SEMICOLON,  // ;
    EOF,
    UNKNOWN
}
```

2-2. Recursive-Descent Parser

EBNF의 parse tree를 재귀 함수의 호출 형태로 구성하는 RD Parsing을 수행하는 RD Parser 클래스를 정의했습니다. 이 클래스의 각 메서드들은 EBNF의 non-terminal의 이름과 일대일 대응됩니다.

syntax error 감지

syntax error는 함수의 리턴값으로 판단합니다. 0보다 작은 값이 리턴되면 syntax error가 발생했다는 것으로 판단할 수 있으며, 해당 리턴 값은 함수 호출 스택을 따라 전파됩니다.

main 함수에서 프로그램 시작하는 부분

```
public static void main(String[] args) {
    /* … */
    while (true) {
        /* … */
        System.out.print(">> ");
        if (parser.program() < 0) System.out.println("syntax error!!");
    }
}
```

RDParser.program() 정의

```
int program() {
    la.lex();
    while (la.nextToken == null || la.nextToken != Token.EOF) {
        if (statement() < 0) return -1;
    }
    System.out.println();
    return 0;
}
```

프로그램 실행

과제 명세에 실행을 어느 시점에 시켜야 하는지에 대해 명시 되어있지 않았기 때문에 인터프리터처럼 파싱을 진행하면서 <statement> 기준으로 문법에 맞다면 동시에 실행도 시키는 방식으로 구현하였습니다.

각 변수의 값은 RD Parser 클래스의 필드에 VARMAP 형태로 저장해 두고 있으며, TRUE, FALSE와 수식의 결과를 모두 저장해 두어야 하기 때문에 값을 문자열 타입으로 저장합니다. 수식의 계산 결과는 <number>의 범위 숫자 즉, 양의 정수만으로 한정된다고 가정하기 때문에 앞에서 설명한 리턴값이 음수일 때 syntax error로 처리하는 과정에 위배되지 않습니다.

‘<expr> -> <bexpr> | <aexpr>’ 문법의 이슈에 관해서

과제에서 주어진 EBNF 문법은 Pairwise disjointness test 를 통과하지 못합니다. 따라서 순수한 RD Parser 로는 구현할 수 없으므로 이것을 해결하기 위한 로직을 추가하였습니다.

이 로직의 의도는 bexpr(), aexpr() 중 어느 함수를 호출해야 하는지를 판단하기 위함입니다. bexpr 과 aexpr 은 첫 번째 토큰으로는 알 수 없지만 두 번째 토큰까지 lookup 하면 확실히 구분할 수 있습니다. 따라서 bexpr()을 먼저 호출한 다음, 두 번째 토큰에서 문제가 생긴다면 aexpr 일 가능성이 있다는 의미이므로 Lexical Analyzer 의 상태를 bexpr()을 호출하기 전으로 되돌린 후 aexpr()을 호출하여 나머지 파싱을 진행하는 구조로 구현하였습니다.

3. 실행 결과

Java

```
C:\Users\harry\.jdks\openjdk-18.0.1.1\bin\java.exe "-javaagent:C:\Pr
>> x = 1234 ;   y = 20 / 2 - 2 *  2 + 3 ; print y ; print x ;
>> 9 1234
>> x = 10 ; print x ; print a ;
>> 10 syntax error!!
>> print 10 ;
>> syntax error!!
>> y = 2 ; x = 3 + y ; print x ;
>> syntax error!!
>> x == 12 34 ; print x ;
>> syntax error!!
>> terminate

Process finished with exit code 0
```

```
C:\Users\harry\.jdks\openjdk-18.0.1.1\bin\java.e
>> a = 10 == 5 + 5 ; print a ;
>> syntax error!!
>> x = 10 <= 10 ; print x ;
>> TRUE
>> x = 10 > 10 ; print y ; print x ;
>> 0 FALSE
>> print x ; x = 1 != 2 ;       print x ;
>> 0 TRUE
>> z = 1 > 2 ; print z
>> FALSE syntax error!!
>> z = 1 > 2 print z
>> syntax error!!
>> z = 1 > 2 ; print z ;
>> FALSE
>> terminate

Process finished with exit code 0
```

Python

```
D:\PycharmProjects\network_programming\.venv\Scripts\python.exe D:\Pycha
>> z = 1 print z ;
>> syntax error!!
>> z = 1 ; print z ;
>> 1
>> y = 10 + * 1 ; print y ;
>> syntax error!!
>> y = 10 ** 2 + 1 ; print y ;
>> syntax error!!
>> print y ; y = 10 ; x = 10 + 20 * 3 - 5 / 2 ; print x ; print y ;
>> 0 68 10
>> terminate

Process finished with exit code 0
```

C++

```
~/SSU/PL  main !2 ?2  g++ Project1.cpp && ./a.out
>> x = 100 - 3 * 4 / 2 + 10  ; print x ;
>> 104
>> print y ; print x ; y = 10 * 3 ; x = 11 ;
>> 0 0
>> print y ; y = 10 * 3 ; print y ;
>> 0 30
>> terminate
```