|  |  |
| --- | --- |
| 이름 | 신동욱 |
| 학번 | 20204101 |
| 학년 | 3 |
| 과목 | 프로그래밍언어론 |



프로그래밍언어론

보고서#4

**-목차-**

1. 재귀 하강 파서 소개
2. 프로그램의 BNF
3. 코드 소개
   1. The Parser Class
   2. lex()
   3. evaluate()
   4. expr()
   5. term()
   6. factor()
   7. number()
   8. The CalGUI Class
4. 실행 결과 및 테스트
5. **재귀 하강 파서 소개**

재귀 하강 파서(Recursive Descent)란, 주어진 문법을 기반으로 입력 수식을 분석하고 구문 트리를 생성하는 알고리즘, 즉 하향식 구문 분석이다. 이 알고리즘은 재귀적으로 작동하며 수식이 더 이상 처리될 수 없는 기본 단위까지 재귀적인 절차를 가지며 분석된다.

일반적으로 BNF를 기술하는 방식과 이 알고리즘의 진행 방식은 매우 유사하며, 따라서 이러한 알고리즘을 통해 계산기를 구현하는 데에 유용하다.

1. **프로그램의 BNF**

<expr> -> <term> + <term> | <term> - <term> | <term>

<term> -> <factor> \* <factor> | <factor> / <factor> | <factor>

<factor> -> <number> | ( <expr> )

<number> -> 1, 2, … 9

예시 : 2 \* ( 3 + 4 )

<expr> -> <term>

-> <factor> \* <factor>

-> <number> \* <factor>

-> 2 \* <factor>

-> 2 \* ( <expr> )

-> 2 \* ( <term> + <term> )

-> 2 \* ( <factor> + <term> )

-> 2 \* ( <number> + <term> )

-> 2 \* ( 3 + <term> )

-> 2 \* ( 3 + <factor> )

-> 2 \* ( 3 + <number> )

-> 2 \* ( 3 + 4 )

따라서 더하기와 곱하기, 괄호에 대해 해당 BNF가 적법함을 확인할 수 있다

1. **코드 소개**

이 프로그램에서 사용한 언어는 java이다. java는 객체지향 프로그래밍을 지원하고, 예외 처리 등 다양한 기능을 손쉽게 제공한다. 또한 C++과 비교하여 그래픽을 좀 더 유연하게, 쉽게 사용할 수 있는 장점이 있다.

1. The Parser Class

class Parser {  
   
 final int *NONE* = 0;  
 final int *OPERATOR* = 1;  
 final int *NUMBER* = 2;  
   
 final String *EOE* = "\0";  
   
 private String exp;  
 private int expIdx;  
 private String token;  
 private int tokType;

public double evaluate(String exp)  
private double expr()   
private double term()  
private double factor()  
private double number()  
private void handleError()  
private void lex()

Parser 클래스의 필드와 메서드이다. NONE, OPERATOR, NUMBER는 토큰에 대한 타입을 나타내는 상수이다. EOE는 문자열의 끝을 나타내는 상수이다.

다음으로, private 멤버 변수들에 대한 설명이다. exp는 입력된 문자열을 뜻하는 String 타입 변수이다. expIdx는 exp 문자열에 대한 인덱스 변수이다. token은 분석된 어휘가 저장될 변수이다. tokType은 현재 토큰에 대한 타입을 나타낸다.

메서드는 총 7개가 존재한다. evaluate 메서드는 입력된 수식에 대하여 재귀 하강 파싱을 호출할 메서드이다. evaluate 메서드에서 expr()을 호출하면 파싱을 시작한다. expr()은 더하기와 빼기, term()은 곱하기와 나누기, factor()는 괄호, number()는 숫자에 대해 처리하는 메서드이다. lex()는 수식에 대하여 토큰으로 분할한다. handleError()는 발생한 예외에 대해 처리한다. 이 클래스의 메서드들에 대해서는 이하에서 자세히 설명한다.

1. lex()

이 메서드는 주어진 수식 문자열에 대해 토큰으로 분할하는 역할을 한다. 현재 위치에서 다음 토큰을 결정하며, 각 토큰은 NUMBER, OPERATOR 로 구분된다.

private void lex() {  
 tokType = *NONE*;  
 token = "";  
 // 기존에는 수식의 공백을 지나쳐야 하지만, 계산기 GUI를 통해  
 // 숫자 입력이 제공되므로 공백이 입력되는 경우는 없다  
   
 if (expIdx == exp.length()) {  
 token = *EOE*;  
 return;  
 }

lex() 메서드가 호출되면 현재 토큰과 토큰 타입의 값을 초기화 한다. 공백에 대해 연산을 하면 안 되기에 교재의 코드에서는 수식의 인덱스를 조절하여 공백을 지나치는 로직이 있다. 하지만 이 프로그램에서는 GUI를 통해 사용자의 입력에 대해 처리하기 때문에, 공백이 입력될 일은 없다. 따라서 해당 로직은 필요 없는 로직이기에 과감히 삭제했다.

if (isOperator(exp.charAt(expIdx))) {  
 token += exp.charAt(expIdx);  
 expIdx++;  
 tokType = *OPERATOR*;

} else if (Character.*isDigit*(exp.charAt(expIdx))) {  
 while (!isOperator(exp.charAt(expIdx))) {  
 token += exp.charAt(expIdx);  
 expIdx++;  
 if (expIdx >= exp.length())   
 break;  
 }  
 tokType = *NUMBER*;

} else {  
 token = *EOE*;   
 }  
}

조건문을 통해 현재 수식 문자열의 인덱스 값이 isOperator 인지 검사한다. 만약 참이라면 token 필드에 해당 값을 더한다. 토큰은 OPERATOR 타입이 되고 인덱스를 조절한다.

만약 조건문에서 그렇지 않으면, 만약 현재 수식 문자열 인덱스 값이 숫자라면(Character.isDigit) 연산자가 나올 때 까지 이하를 계속 반복한다. 이러한 과정을 거치는 이유는 숫자, NUMBER는 하나의 요소만이 토큰이 되는 것이 아니라 연속된 숫자, 즉 123이라는 숫자에 대해 하나의 토큰으로 취급하기 위해 이러한 과정을 거친다.

private boolean isOperator(char c) {  
 return "+-\*%^=()÷".indexOf(c) != -1;  
}

위의 코드에서 사용한 isOperator 메서드이다. 매개변수로 들어온 문자에 대해 사용자가 지정한 문자열에 해당 문자가 포함되어 있는지 검사한다. 지정한 문자열에는 파서에서 사용할 여러 연산자들이 포함되어 있다.

1. evaluate()

evaluate() 메서드는 입력된 문자열에 대해 재귀 하강 파싱을 호출하여 파싱을 시작하는 메서드이다.

public double evaluate(String expstr) throws ParserException {  
 double result;  
 exp = expstr;  
 expIdx = 0;  
   
 lex();  
 if (token.equals(*EOE*))  
 handleError();

result = expr();  
   
 if (!token.equals(*EOE*))  
 handleError();  
 return result;  
}  
   
 result = expr();  
   
 if (!token.equals(*EOE*))  
 handleError();  
   
 return result;  
}

클래스의 필드인 exp에 값을 지정하고 expIdx를 초기화한다. 이후 lex() 메서드를 호출하여 어휘 분석을 실시한다. 조건문을 사용하여 입력된 문자열에 대해 빈 문자열인지 검사한다. 만약 입력 문자열이 공백이라면 예외를 처리한다.

그 후 expr() 메서드를 호출하여 재귀 하강 파싱을 시작한다. expr()의 반환 값을 result에 저장한다. 이 값은 계산한 결과값이 저장된다.

다시한번 조건문을 사용하여 토큰을 검사한다. 파싱이 끝났는데 토큰의 값이 \0이 아니라면, 파싱을 완료했는데 토큰이 남아 있다면, 이는 수식 오류이기 때문에 예외를 처리한다.

1. expr()

private double expr() throws ParserException {  
 char op;  
 double result;  
 double nextResult;  
   
 result = term();  
   
 while ((op = token.charAt(0)) == '+' || op == '-') {  
 lex();  
 nextResult = term();  
 switch (op) {  
 case '+' -> result += nextResult;  
 case '-' -> result -= nextResult;  
 }  
 }  
 return result;  
}

expr() 메서드는 재귀 하강 파싱이 시작됐을 때 가장 먼저 호출되는 메서드이다. 이 메서드는 표현식을 평가하는 역할을 한다. 표현식은 용어(term)의 합 또는 차로 이루어져 있다. 각 항을 term() 메서드를 통해 용어를 두개 받아와 이들의 연산을 처리한다.

1. term()

private double term() throws ParserException {  
 char op;  
 double result;  
 double nextResult;  
   
 result = factor();  
 while ((op = token.charAt(0)) == '\*' || op == '÷') {  
 lex();  
 nextResult = factor();  
 switch (op) {  
 case '\*' -> result \*= nextResult;  
 case '÷' -> {  
 if (nextResult == 0.0) handleError();  
 result /= nextResult;  
 }  
 }  
 }  
 return result;  
}

term() 메서드는 expr()에서 호출되는 메서드이다. 이 메서드는 용어를 평가하는 역할을 한다. 각 용어는 요소의 곱 또는 나눗셈으로 이루어져 있다. 각 용어를 factor() 메서드를 통해 요소를 두개 받아와 이들의 연산을 처리한다.

1. factor()

private double factor() throws ParserException {  
 double result;  
   
 if (token.equals("(")) {  
 lex();  
 result = expr();  
 if (!token.equals(")"))  
 handleError();  
 lex();  
 } else result = number();  
   
 return result;  
}

factor() 메서드는 term()에서 호출되는 메서드이다. 이 메서드는 요소를 평가하는 역할을 한다. 요소는 괄호로 감싼 표현식 또는 숫자를 의미한다. 토큰이 괄호인지 검사를 해서 괄호라면, 다시 expr() 메서드를 호출한다. 만약 괄호가 아니라면 number() 메서드를 호출하여 숫자를 받아온다.

1. number()

private double number() throws ParserException {  
 double result = 0.0;  
   
 if (tokType == *NUMBER*) {  
 result = Double.*parseDouble*(token);  
 lex();  
 } else {  
 handleError();  
 }  
 return result;  
}

number() 메서드는 factor() 에서 호출되는 메서드이다. 이 메서드는 재귀 하강 파서의 제일 최소 단위가 되는 메서드이다. 현재 토큰을 읽어서 NUMBER 타입인지 검사하고, NUMBER 라면 토큰을 double 타입으로 형 변환하여 result에 담아 반환한다.

1. The CalGUI Class

public class CalGUI extends JFrame implements ActionListener {  
   
 Parser p = new Parser();  
 private final JTextField *ExpTextField*; //수식이 입력될 텍스트 필드  
 private final JTextField *ResultTextField*; //결과를 보여줄 텍스트 필드  
   
 public CalGUI() {

//생략

case "=" -> {  
 *ResultTextField*.setFont(new Font("Arial", Font.***BOLD***, 30));  
 String expression = *ExpTextField*.getText();  
   
 try {  
 double result = p.evaluate(expression);  
 *ResultTextField*.setText(Double.*toString*(result));  
 } catch (ParserException ex) {  
 *ResultTextField*.setText("" + ex);  
 }

//생략  
}

}

CalGUI 라는 클래스는 그래픽을 담당하는 클래스이다. 이 전에 작성한 재귀 하강 파서의 객체를 생성하여 가지고 있다.

\* 이 보고서에 GUI와 관련된 코드를 전부 담기에는 GUI 코드가 너무 길어서 보고서의 목적이 저해될 것 같기에 담지 않았다.

사용자가 누른 버튼에 대하여 텍스트 필드의 값이 조절된다. 예를 들어 숫자를 입력하면 ExpTextField 값이 변경된다. “=” 버튼을 누르면 ExpTextField 값을 가져와서 파서의 evaluate()를 호출하여 수식을 평가한다. 평가한 수식을 ResultTextField에 담아 사용자에게 표시한다.

텍스트, 스크린샷, 번호, 달력이(가) 표시된 사진

자동 생성된 설명

ResultTextField

ExpTextField

-계산기 프로그램의 최초 실행 모습-

1. **실행 결과 및 테스트**

다양한 수식 데이터에 대해 평가하고자 한다. 그 전에 앞서, 이 프로그램은 윈도우의 기본 계산기 프로그램을 모방하여 만들어졌다. 따라서, 제곱과 루트에 대해서는 재귀 하강 파서에 한 연산으로 처리하는 것이 아닌, 계산기 자체에서 계산하는 방식이다. 이는 윈도우 기본 계산기 프로그램의 동작 방식과 동일한 방법이다.

텍스트, 스크린샷, 번호, 달력이(가) 표시된 사진

자동 생성된 설명더하기 연산

순서대로 3, +, 5, = 클릭

( 보고서의 사진으로는 담기   
어렵지만 Del 버튼을 클릭하면   
1개가 지워진다 )

스크린샷, 텍스트, 라인, 번호이(가) 표시된 사진

자동 생성된 설명CE 클릭 (Clear)

텍스트 필드의 값을 비운다.

**\* 보고서의 가독성을 위해 상단   
텍스트 필드만 첨부**

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

빼기 연산

순서대로 6, -, 1, = 클릭

텍스트, 스크린샷, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

곱하기 연산

순서대로 2, \*, 3, = 클릭

텍스트, 스크린샷, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

나누기 연산

순서대로 9, ÷, 5, = 클릭

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명여러 연산을 한번에 입력.

여러 괄호에 대해서 처리

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

0으로 나누었을 때 에러 발생

수식에 대해 에러 발생.

괄호의 짝이 맞지 않음