Advanced Calculator

Using JavaCC

Jonathan Verbeek

The project

- Simple calculator from class only supported
  + - * / ( )
- While that‘s already cool, it could be extended
- Goal: make it useful like a real calculator

Features

- Standard operations (adding, subtracting, multiplying, dividing)
- Intelligent use of braces
- * / operations have higher precedence than + -

Features

- Standard operations (adding, subtracting, multiplying, dividing)
- Intelligent use of braces
- * / operations have higher precedence than + -
- No termination after parsed expression => endless calculator
- Modulo % operator
- Functions: sin(), cos(), tan(), sqrt(), pow(), printMemory()
- Variables (assignment and read)

JavaCC Grammar

- Skipped tokens include:

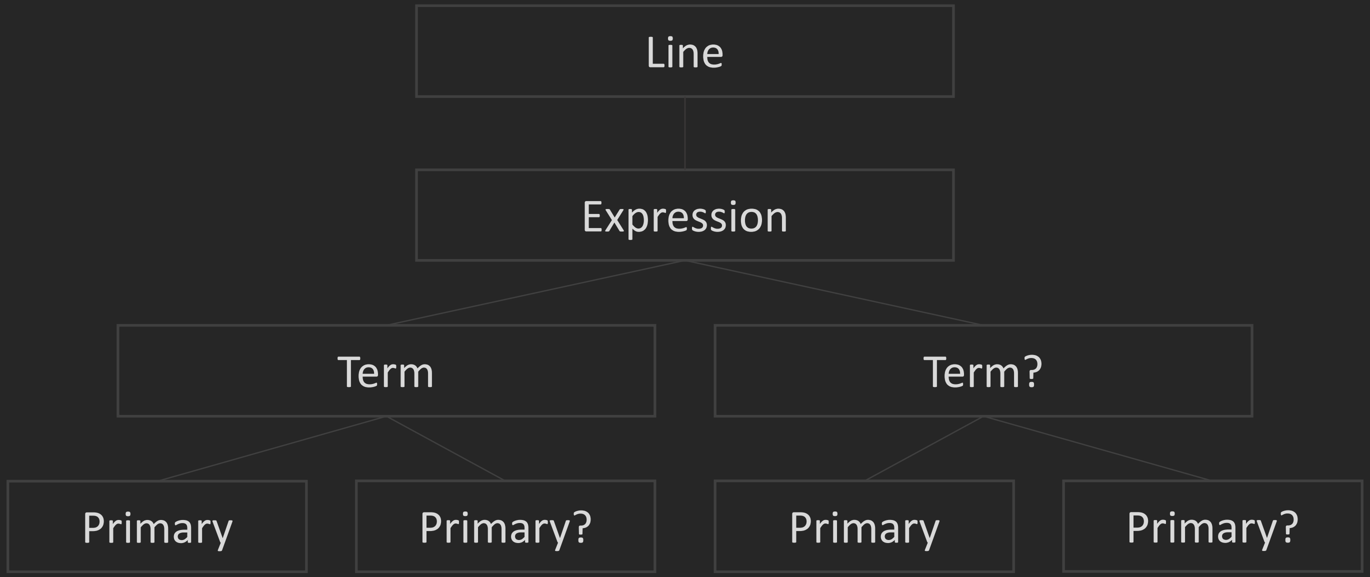
```
// These characters / regular expressions will be skipped while parsing
SKIP :
{
   // Whitespace characters and single-line comments
   " " | "\t" | "\r" | "\n" | < "//" (~["\n"])* "\n" >
}
```

JavaCC Grammar

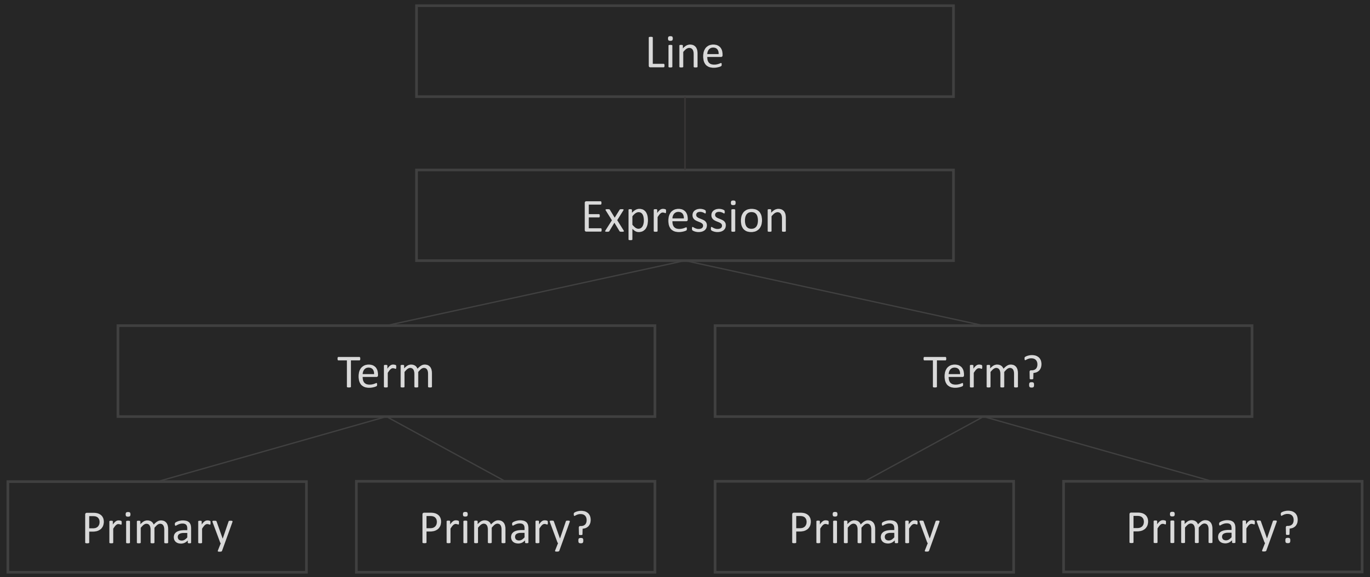
- Parsed tokens include:

```
// All tokens the program can parse, these are integers, decimal numbers and strings
for variable names
TOKEN :
{
    < Number : (([“0”-“9”])+ (“.” ([“0”-“9”])*)?) | (([“0”-“9”])* “.” ([“0”-“9”])+) >
    |
    < Variable : ([“a”-“z”,“A”-“Z”,“0”-“9”,“_”])* >
}
```

JavaCC Parsing Tree



JavaCC Parsing Tree



Line

Expression

Term

Term?

Primary

Primary?

Primary

Primary?

5 + 4 * 2;

5 + 4 * 2

5 +

4 * 2

JavaCC Parsing Tree

- **Line**: strips the semicolon, creates an Expression
- **Expression**: has a left and a right Term, creates an Add- or SubtractExpression
- **Term:** has a left and a right Primary, creates an Multiplication-, Division- or ModuloExpression
- **Primary**: parses out braces, unary minus, simple number statements, variable assignments and/or reads and functions

ModuloExpression

```java
//Expression for the % operation
public class ModuloExpression extends Expression
{
    // Left and right expressions of the operation
    private Expression fLeft;
    private Expression fRight;

    // Getters for the expressions
    public Expression getLeft() { return fLeft; }
    public Expression getRight() { return fRight; }

    // Constructor taking in the two expressions
    public ModuloExpression(Expression aLeft, Expression aRight)
    {
        fLeft = aLeft;
        fRight = aRight;
    }

    // Overwritten evaluate
    public BigDecimal evaluate()
    {
        return fLeft.evaluate().divideAndRemainder(fRight.evaluate())[1];
    }
}
```

Expressions

Getters

Constructor

Actual calculation

Variables

- Get parsed inside Primary()

```
// Now it gets tricky: since JavaCC won't know whether we want to READ or WRITE a variable, we need to specify a lookahead,
// so it can look ahead in the stream to see if there's a equal sign coming or not
LOOKAHEAD(2)
// Variable assignment, using the variable name stored in the Token t, and the expression stored in e after the "=" sign
t = < Variable > "=" e = Expression() { return new VariableAssignExpression(t.image, e, memory); }
|
// Variable read, under the hood it just returns a NumberExpression getting the value from the memory
t = < Variable > { return new NumberExpression(memory.get(t.image)); }
|
LOOKAHEAD(1)
```

Variables

- VariableAssignExpression saves a variable
  - named `t.image`
  - with the value „result of the Expression e“
  - in the memory

```
t = < Variable > "=" e = Expression() { return new VariableAssignExpression(t.image,
e, memory); }
```

Variables

- The memory is a simple Hashtable, associating a
  - String = Name of a variable
  - BigDecimal = Evaluated value of a variable
- Values are stored using `memory.put(key, value)`
- Can also be used to add constants, such as Pi or the Euler constant

```java
public static Hashtable<String, BigDecimal> memory = new Hashtable<String,
BigDecimal>();
```

Variables

- Reading a variable is simple
- Get the value for the variable name stored in the Tokens image `t.image`
- Create a new NumberExpression with that value
- Variables are handled as numbers and can be used in other expressions

```
t = < Variable > { return new NumberExpression(memory.get(t.image)); }
```

Functions

- Functions work the same way the mathematical operations worked
- Examples are SineFunctionExpression, SqrtFunctionExpression, …
- Parsed like this:

```
Expression Function() :
{
    // Gets assigned with the expression for the function evaluated below
    Expression e1, e2;
}
{
    // sin(n)
    "sin(" e1 = Expression() ")" { return new SineFunctionExpression(e1); }
    |
    // cos(n)
    "cos(" e1 = Expression() ")" { return new CosFunctionExpression(e1); }
    |
    // tan(n)
    "tan(" e1 = Expression() ")" { return new TanFunctionExpression(e1); }
    |
    // sqrt(n)
    "sqrt(" e1 = Expression() ")" { return new SqrtFunctionExpression(e1); }
    |
    // pow(b, e)
    "pow(" e1 = Expression() "," e2 = Expression() ")" { return new PowFunctionExpression(e1, e2); }
}
```

Demo

- Full source code available under https://github.com/iUltimateLP/JavaCC-Calculator