Homework 1 - LL(1) Calculator Parser - Translator to Java

Part 1

For the first part of this homework you should implement a simple calculator. The calculator should accept expressions with the bitwise AND(&) and XOR(^) operators, as well as parentheses. The grammar (for single-digit numbers) is summarized in:

exp -> num | exp op exp | (exp)

op -> ^ | &

num -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

You need to change this grammar to support priority between the two operators, to remove the left recursion for LL parsing, etc.

This part of the homework is divided in two parts:

1. For practice, you can write the FIRST+ & FOLLOW sets for the LL(1) version of the above grammar. In the end you will summarize them in a single lookahead table (include a row for every derivation in your final grammar). This part will not be graded.

2. You have to write a recursive descent parser in Java that reads expressions and computes the values or prints "parse error" if there is a syntax error. You don't need to identify blank space or multi-digit numbers. You can read the symbols one-by-one (as in the C getchar() function). The expression must end with a newline or EOF.

Part 2

In the second part of this homework you will implement a parser and translator for a language supporting string operations. The language supports the concatenation operator over strings, function definitions and calls, conditionals (if-else i.e, every "if" must be followed by an "else"), and the following logical expressions:

- is-prefix-of (string1 prefix string2): Whether string1 is a prefix of string2.
- is-suffix-of (string1 suffix string2): Whether string1 is a suffix of string2.

All values in the language are strings.

Your parser, based on a context-free grammar, will translate the input language into Java. You will use JavaCUP for the generation of the parser combined either with a hand-written lexer or a generated-one (e.g., using JFlex, which is encouraged).

You will infer the desired syntax of the input and output languages from the examples below. The output language is a subset of Java so it can be compiled using the "javac" command and executed using the "java" command or online Java compilers like this, if you want to test your output.

There is no need to perform type checking for the argument types or a check for the number of function arguments. You can assume that the program input will always be semantically correct.

Note that each file of Java source code you produce must have the same name as the public Java class in it. For your own convenience you can name the public class "Main" and the generated files "Main.java". In order to compile a file named Main.java you need to execute the command: javac Main.java. In order to execute the produced Main.class file you need to execute: java Main.

To execute the program successfully, the "Main" class of your Java program must have a method with the following signature: public static void main(String[] args), which will be the main method of your program, containing all the translated statements of the input program. Moreover, for each function declaration of the input program, the translated Java program must contain an equivalent static method of the same name. Finaly, keep in mind that in the input language the function declations must precede all statements.

Example #1

Input:

```
    name()  {
"John"
}

surname() {
"Doe"
}

fullname(first_name, sep, last_name) {
first_name + sep + last_name
}

name()
```

```
surname()
fullname(name(), " ", surname())
```

Output (Java):

```
public class Main {
public static void main(String[] args) {
System.out.println(name());
System.out.println(surname());
System.out.println(fullname(name(), " ", surname()));
}

public static String name() {
return "John";
}

public static String surname() {
return "Doe";
}

public static String fullname(String firstname, String sep, String last_name) {
return first_name + sep + last_name;
}
}
```

Example #2

Input:

```
    name() {
"John"
}

repeat(x) {
x + x
}

cond_repeat(c, x) {
if (c prefix "yes")
    if("yes" prefix c)
        repeat(x)
    else
        x
else
    x
}

cond_repeat("yes", name())
cond_repeat("no", "Jane")
```

Example #3

Input:

```
    findLangType(langName) {
if ("Java" prefix langName)
    if(langName prefix "Java")
        "Static"
    else
        if("script" suffix langName)
            "Dynamic"
        else
            "Unknown"
else
    if ("script" suffix langName)
        "Probably Dynamic"
    else
        "Unknown"
}

findLangType("Java")
findLangType("Javascript")
findLangType("Typescript")
```