Assignment 2019
(20 % of CS259 assessment)

[**Due by 12 PM, April 24, 2019**]

1 Language Description

PLM (Programming Language of the Moment) is a language that allows users to write code that computes non-negative integers. A PLM program consists of several lines of code, each of which defines a single function. For instance the function that returns its argument incremented by 4 can be defined using the following syntax.

```
DEF ADDFOUR x { x+4 } ;
```

`ADDFOUR` will be called the *function name*, `x` will be called the *parameter name* and `x+4` is the *function body.*

A *function definition* must contain the seven elements specified below. They must occur in a single line, exactly in the order listed below, and must be separated from each other by exactly one space character.

1. keyword `DEF`
2. function name
3. parameter name
4. left brace
5. function body
6. right brace
7. semicolon

The character `D` from `DEF` must be the first character on the line. The final semicolon must be followed immediately by the end-of-line character. **The semicolon in the last function definition must be followed immediately by the end-of-line character and then the end-of-file character.**

Additionally, PLM code must satisfy all conditions described below.

1. Function names are non-empty strings of upper-case letters with one exception. The word `DEF` is a keyword *and* a reserved word in the language. Consequently, `DEF` cannot be used as the name of a function.
2. Parameter names are non-empty strings of lower-case letters.

3. The only exception to the rules is the function name `MAIN`. No parameter name is allowed after `MAIN`, i.e. `MAIN` must be followed by one space and then the left brace.

4. There can be no whitespace inside the function body, but remember that the body must be separated from the enclosing braces by one space on each side.

5. The function body is an arithmetic expression built from non-negative integers, the associated parameter name as well as function calls. The only arithmetic operations allowed are addition and multiplication. Parentheses are not allowed, except in function calls (see next point). The function body must be non-empty.

6. Function calls have to refer to functions that have been defined as part of the same program. Function definitions are listed in no particular order. It is possible for a function body to make calls to functions defined later in the program. Functions can also call themselves.

7. Any function different from `MAIN` can be called. Function calls are made by mentioning the relevant function name along with an argument enclosed in a pair of parentheses, e.g. `ADDFOUR(3+5*6)`. No Whitespace can occur between the parentheses. The argument must satisfy the same constraints as function bodies. That is, it must be a non-empty arithmetic expression built from addition, multiplication, non-negative integers, the parameter name of the function which is making the call, as well as calls to other functions.

8. Every program must define the `MAIN` function. No function can be defined twice.

9. No characters other than the specific ASCII characters referenced so far, are allowed.

Here are some examples of PLM programs.

1. **Example 1**

   ```
   DEF MAIN { 1+ADDFOUR(2+ADDFOUR(3)) } ;
   DEF ADDFOUR x { x+4 } ;
   ```

2. **Example 2**

   ```
   DEF ABCD xyz { BCD(xyz) } ;
   DEF BCD xy { 2*CD(xy) } ;
   DEF CD x { D(x)+EF(x) } ;
   DEF D x { 10 } ;
   DEF EF x { 10*x } ;
   DEF MAIN { ABCD(1) } ;
   ```

3. **Example 3**

   ```
   DEF QQ yy { 2*PP(yy)+3*QQ(yy) } ;
   DEF PP xx { QQ(xx)+3 } ;
   DEF MAIN { PP(0)+3 } ;
   ```

Here are some examples of pieces of code which *do not* constitute a legitimate PLM program.

1. **Non-example 1**

   ```
   DIF MAIN { 1+ADDFOUR(2+ADDFOUR(3)) } ;
   ```

   This is not a PLM program because `DIF` is used instead of `DEF`.

2. **Non-example 2**

   `DEF P2P xXx { 3*Q()+R(6,Q(5))  };`

   This is not a PLM program due to any one of the following reasons.

   (a) The function name `P2P` contains a number but function names are only allowed to contain upper-case letters.

   (b) The parameter name `xXx` contains an upper-case letter, which is not allowed.

   (c) The function body contains calls to undefined functions `Q` and `R`. Also there are two spaces before }.

   (d) The `MAIN` function is missing.

   (e) There is no space between } and ;.

   (f) The argument in the call to `Q` is empty.

   (g) The argument in the call to `R` contains a ',' which is not allowed.

2 Tasks

PLM programs can be executed by running the function body of `MAIN`. This may result in calls to other functions, which may in turn call further functions and so on. When a function call is made, we assume that the argument is always evaluated first and the value is then substituted for all occurrences of the corresponding parameter in the function body. Sometimes this will produce a result: the first two examples of PLM programs we saw earlier return 14 and 40 respectively. In cases when there are circular dependencies between functions (eg. Example 3), the program will not terminate. For the purposes of evaluation, we assume that multiplication takes precedence over addition.

2.1 Task one

Implement a parser (along with a lexer) that recognizes PLM programs.

2.2 Task Two

Extend the parser to an evaluator so that it can determine whether the input program returns a result or not. If the former is the case, the result (a non-negative integer) should be printed out.

3 I/O Specifications

Your parser should read the input from `System.in`. Parsing errors should be reported on `System.err` and error messages should give one reason why the input cannot be regarded as a PLM program. `System.out` should be used for output.

- If the input is a valid PLM program, two lines must printed out: the first line only contains the word `PASS` and the second line must contain information about the results, as explained in the next sentence. If the program evaluates to a number, then the number should be printed out on the second line. Otherwise, the line should read `DIVERGENCE`.

- If the input is *not* a PLM program, only a single line with `FAIL` should be printed out to the standard output stream. The error message containing a reason why the input is not a PLM program should be reported on System.err. **You are not allowed to simply use default `javacc` messages regarding uncaught exceptions as a reason for the input not being a PLM program.** Your error message must explicitly point out at least one element from the list of specifications which is violated by the input. Consequently, you are expected to decode the `javacc` exceptions to provide your own messages.

Here is the expected output for the examples given earlier.

Example 1

```
PASS
14
```

Example 2

```
PASS
40
```

Example 3

```
PASS
DIVERGENCE
```

Nonexample 1

```
FAIL
```

Nonexample 2

```
FAIL
```

For Nonexample 1, a possible error message would be"Missing keyword DEF" and for Nonexample 2, a possible error message would be"MAIN function missing".

4 Further Information

4.1 Submission instructions

Submit a single file named `Assignment.jj` containing your specification via Tabula. The file Assignment.jj must cause `javacc` to generate your parser `Assignment.java`. Compilation should work on DCS machines using the commands:

```
javacc Assignment.jj
javac *.java
```

For reading an input file called `test.txt`, the command

```
java Assignment < test.txt
```

should produce the required output. The output of the parser should just appear on screen, it should **not** be sent to another file. To test your solution with the input file `test.txt`, invoke

```
java Assignment < test.txt > output.txt 2>err.txt
```

and check if the contents of output.txt and err.txt conform to the specifications regarding the standard output and error messages respectively.

The testing of your solutions will be **automated** and will use the compilation commands described above. So please make sure that you stick to *all* of the specifications (they are case sensitive) exactly. Moreover, please make sure that you send each type of content to the correct stream.

4.2 Evaluation policy and constraints

Task One is worth 50% and Task Two is worth 30%. The remaining 20% is for readable and maintainable code. Comments and indentation improve readability. Maintainable source code can easily have new features added to it.

- During evalutation, any change that must be made to the file name or parser name to rectify compilation/execution issues will incur a penalty of -20%. No other changes to your code will be permitted once the deadline has passed, so please make sure that you have submitted the correct version and there are no problems with compilation.
- You may not use JJTree for Task Two.
- You are not allowed to use StackOverFlowError to detect loops in the input program. Loops must be explicitly detected by your parser/evaluator.

5 Suggestions

Since partial credit will be given for partially working solutions, you may find it helpful to build your solution incrementally. You may discuss with fellow students the general workings of `javacc` or parsing, but you are **not allowed to collaborate** on the solution. The University of Warwick takes plagiarism seriously, and penalties will be incurred if any form of plagiarism is detected. Copying, or basing your work on, solutions written by people who have not taken this course is also considered plagiarism. This includes material that has been downloaded from the internet.

BACKUP: Please keep a copy of everything you submit!

STUDY: Inspect the MyParserTokenManager, MyParserConstants, Token, ParseException, TokenMgrError classes carefully.

WINDOWS USERS: Please pay attention to the fact that Windows uses "$\backslash r \backslash n$" for line breaks, while the files that will be used to test your code will use "$\backslash n$".