L.EIC: Functional and Logic Programming
First Project 2025/26

Pedro Vasconcelos, DCC/FCUP

October 30, 2025

1 Overview

- This first project for 2025/2026 edition of the Functional and Logic Programming course consists of extending the basic calculator presented in Lecture 11 (Parsing).
- You should re-use the `Parsing.hs` module unchanged and extend the `Calculator.hs` module as described below. These files are available in the course's Moodle page.
- The proposed extensions are divided into two separate parts; you can obtain partial points for completing any of the parts.
- The project can be discussed with other students and professors. You can ask for help, or examine other students code as a means to understand the problem and its solution. However, remember that the focus must on the *understanding*, not just memorizing the code (see the following point).
- Students will be required to re-construct their solution individually under exam conditions during the mid-term test (scheduled for 14th November). You will have acess to this document and the parsing library and basic calculator files. You are expected to be able to test and debug your solution before submitting it.

2 Part 1

Extend the calculator with operations for subtraction (`-`), integer division (`/`) and remainder (i.e. modulus) (`%`). The revised grammar for the expression

language is as follows:

expr	::=	term exprCont
exprCont	::=	`'+'` term exprCont | `'-'` term exprCont | ε
term	::=	factor termCont
termCont	::=	`'*'` factor termCont | `'/'` factor termCont
	|	`'%'` factor termCont | ε
factor	::=	(as before)

Because the extended calculator will perform divisions, there is a possibility of dividing by zero. It is acceptable if your program crashes with a runtime error in those situations (these will not be tested in the grading examples).

Your program should still read multiple expressions from the standard input and output each result in a separate line.

Some sugestions: you will need to extend the `Expr` data type, the `expr` and `term` parsers and the `eval` evaluation function.

Sample input 1	**Sample output 1**
`10-5+12`	`17`
`(10+5)/(2+7)`	`1`
`(10+5)%(2+7)`	`6`
`1234/6/2`	`102`
`1234/(6/2)`	`411`

3 Part 2

Extend the calculator with variables and commands. A *variable* is a sequence of one or more letters. A *command* is either an assignment of the form *variable*`=`*expression* or just an expression.

expr	::=	(as before)
factor	::=	variable | natural | `'('` expr `')'`
command	::=	variable `'='` expr | expr

The extended calculator will now read commands and execute them:

- if a command is an assignment, then you should evaluate the expression, assign the result value to the variable and print it;
- if a command is an expression, then you should evaluate the expression and print the result;
- variables defined in one assignment can be used in subsequent expressions.

It is an error to use a variable before it has been assigned. Your program may crash with a runtime error in such conditions (the examples used for grading will never include such cases).

Some sugestions: Define a new `Command` type for commands and a suitable command parser. You also will need to extend the `Expr` type, the parser and evaluation function. To keep track of the values of variables you can use an *environment* as in the evaluator for propositions of Lecture 10, e.g.

```
type Env = [(Name, Integer)]
```

Note that the execution of a command requires the current environment as input and can result in modifying the value of a variable; to implement this we need to change the evaluation function from

```
evaluate :: String → String
```

to

```
execute :: Env → String → (String, Env)
```

The result is now a pair of the output string and the new environment (which can be same as the given environment when there is no assignment). Examples:

```
>>> execute [("x",1), ("y",2)] "x+y"
("3", [("x",1), ("y",2)])
>>> execute [("x",1), ("y",2)] "x=x+1"
("2", [("x",2), ("y",2)])
```

Your program should read multiple commands from the standard input and output each result in a separate line.

Sample input 2

```
x=1
x=x*x+1
xx=x+3
(x+xx+1)*xx
y=xx*xx+123
yy=y*y*x*xx
```

Sample output 2

```
1
2
5
40
148
219040
```

Sample input 3

```
a=1234
b=(a*a/100)%10000
a=(b*b/100)%10000
ab=a*b
(ab-a-b)/(2*a-b)
```

Sample output 3

```
1234
5227
3215
16804805
13962
```