

# Course Project: Stack Language

Functional Programming (ITI0212)

Due 2021.04.26

The course project is to implement a simple stack language. The project is divided into four tasks. In each task, you will expand the capabilities of your stack language. Your grade for the project will be determined by how many tasks you complete, and by how well you have completed them. The first and second task are each worth 30% of your overall grade, and the third and fourth task are each worth 20%. You will be graded both for correctness and style, with correctness making up 70% of your grade, and style the remaining 30%. Your git commit logs will be assessed as part of the style component, so push early, and push often. The project is worth 30% of your overall course grade.

You are expected to work on your project using the TalTech GitLab server (<https://gitlab.cs.ttu.ee>) within the repository named `iti0212-2021` that you have been using to submit your assignments. Please make a directory named `project` in this repository, and store your project files there. When the submission deadline arrives, the contents of this directory will be collected automatically. Please include a file named `README.txt` that contains your name in the project directory to assist in the grading process.

Recall that a *stack* is an abstract data type that serves as a collection of elements, and supports two principal operations:

- **Push**, which adds an element to the collection, and
- **Pop**, which removes the most recently added element that has not yet been removed.

A *stack language* is a way of specifying computations, built around a stack. A stack language consists of a sequence of *symbols*, each with an effect on the stack. For example, a symbol might have one of the following effects:

- Push one or more values onto the stack.
- Pop one or more values off of the stack, perform an operation on them, and push the result.
- Rearrange the elements at the top of the stack.

A *program* in a stack language is a sequence of symbols. The effect of a program is to perform the effects of its constituent symbols in the order they appear.

# 1 RPN Calculator (30%)

You may already be familiar with *reverse Polish notation (RPN)* – also known as *postfix* notation. When we write down arithmetic expressions like  $5 + 3$  or  $(3 + 5) * 6$  we write the operations between their arguments. That is, we use *infix* notation. This is not the only possibility. With reverse Polish notation, we write the operations after their arguments instead of between them:

Infix Notation:		Postfix Notation:
5 * 3		5 3 *
(3 + 5) * 6		3 5 + 6 *
3 + (5 * 6)		3 5 6 * +
(3 + 4) * (5 + 1)		3 4 + 5 1 + *

The reason that people often call this “reverse Polish notation” instead of simply “postfix notation” is that it was first introduced by a Polish logician named Jan Łukasiewicz. Writing expressions using postfix notation makes them simpler to deal with programmatically. For example, notice that when using postfix notation we do not need any parentheses to specify the order of operations.

**Your first task** is to write a function `eval : String -> Maybe Nat` that evaluates arithmetic expressions written in reverse Polish notation. Your calculator must support natural number addition and multiplication. If the input string is not a valid arithmetic expression, your function should return `Nothing`. Otherwise, your function should evaluate the expression and return `Just` the resulting natural number. For example:

```
REPL> eval 3
Just 3
REPL> eval "3 5 +"
Just 8
REPL> eval "+ 3 5"
Nothing
REPL> eval "3 + 5"
Nothing
REPL> eval "5 3 *"
Just 15
REPL> eval "hello"
Nothing
REPL> 3 5 + 6 *
Just 48
REPL> 3 5 6 * +
Just 33
REPL> eval ""
Nothing
REPL> 3 4 + 5 1 + *
Just 42
```

You must implement `eval` as a stack language. Break the input string (the program) into appropriate symbols, and compute the result by performing their effects to an initially empty stack in the order they appear.

## 2 IO and Program Files (30%)

**Your second task** has two parts. First, we want to read the program for our stack language from a file, instead of typing it in to the REPL. To do this you should use the functions provided by the `System.File` library. In particular the function

```
readFile : (filepath : String) -> IO (Either FileError String)
```

can be used to obtain the contents of a file as a `String`. You should now be able to run your evaluator as an executable. For example, if the program is stored in file `prog` then we might write

```
$> ./interpret prog
```

to call run our evaluator on the program.

Second, we want to add symbols to the stack language for reading user input, and for writing to the standard output. Specifically, add symbols `r` and `p` to your language. The effect of `r` on the stack is to prompt the user for input, attempt to parse that input as a symbol, and perform the effect of that symbol on the stack. If the user input cannot be parsed as a symbol then evaluation should fail, returning `Nothing`. The effect of `p` is pop the top element of the stack and print it. If there is no top element of the stack, then evaluation should fail, returning `Nothing`. For example:

```
$> echo 3 5 r > prog && ./interpret prog
$> Please Enter a Symbol: +
$> 8
$> echo 3 5 r > prog && ./interpret prog
$> Please Enter a Symbol: *
$> 15
$> echo 3 5 r > prog && ./interpret prog
$> Please Enter a Symbol: hello
$> No Result!
$> echo 3 r + > prog && ./interpret prog
$> Please Enter a Symbol: 5
$> 8
$> echo 3 r + > prog && ./interpret prog
$> Please Enter a Symbol: *
$> No Result!
$> echo r r r > prog && ./interpret prog
$> Please Enter a Symbol: 3
$> Please Enter a Symbol: 5
```

```

$> Please Enter a Symbol: *
$> 15
$> echo 15 p 5 > prog && ./interpret prog
$> Evaluation Prints: 15
$> 5
$> echo r p 3 > prog && ./interpret prog
$> Please Enter a Symbol: 4
$> Evaluation Prints: 4
$> 3
$> echo 3 5 * 4 + p 3 > prog && ./interpret prog
$> Evaluation Prints: 19
$> 3

```

### 3 Adding a Global Store (20%)

**Your third task** is to add a global store to your stack language. Symbols will have an effect both on the stack, and on the global store. This involves two new symbols, **b** and **d**. The effect of **b** is to store the second element of the stack at a position in the global store given by the first element of the stack, after which neither the first nor second element should remain on the stack. The effect of **d** on the stack is to replace the top element of the stack with the element at that position in the global store. If no such element is present in the store, evaluation should fail. For example:

```

$> echo 42 4 b 4 d > prog && ./interpret prog
$> 42
$> echo 42 4 b 5 d > prog && ./interpret prog
$> No Result!
$> echo 6 5 b 3 4 + 5 d * > prog && ./interpret prog
$> 24
$> echo 5 1 b 2 2 b 10 1 d + > prog && ./interpret prog
$> 15
$> echo 5 1 b 2 2 b 10 1 d + 2 d *
$> 30
$> 2 1 b 1 d 1 d 1 d 1 d * * * > prog && ./interpret prog
$> 16
$> 2 1 b 3 1 b 1 d 1 d * > prog && ./interpret prog
$> 9

```

It may be helpful to think of the global store as a table of positions and their contents, as in:

position	1	5	10
contents	42	5	2

which we can understand as a global store with three entries: position 1 holds 42, position 5 holds 5, and position 10 holds 2. Now, the effect of `15 10 b` would be to replace the contents of position 10 with 15, resulting in:

```
position | 1 | 5 | 10
-----+-----+-----
contents | 42 | 5 | 15
```

From here, the effect of `7 3 b` would be to store 7 in position 3:

```
position | 1 | 3 | 5 | 10
-----+-----+-----+-----
contents | 42 | 7 | 5 | 15
```

and the effect of, say, `1 d` would be to push 42 onto the stack, resulting in the same store:

```
position | 1 | 3 | 5 | 10
-----+-----+-----+-----
contents | 42 | 7 | 5 | 15
```

## 4 Something More (20%)

**Your fourth task** is to further extend the functionality of your language. You may choose what you would like to do, for example from the following list:

1. Make your language into a Turing-complete model of computation. See for example the Joy language (<http://joy-lang.org/>).
2. Make your stack language draw *turtle graphics* ([https://en.wikipedia.org/wiki/Turtle\\_graphics](https://en.wikipedia.org/wiki/Turtle_graphics)) as it executes, with special instructions to control the turtle.
3. Make your stack language into a command line utility. Think of something for it to do, come up with a reasonable set of features, and implement them.

Alternatively, you are encouraged to come up with your own ideas about what you would like your stack language to do. In this case, you must tell discuss your idea with the lecturers so that we can agree on something that is both nontrivial and possible to complete in a reasonable amount of time. Your idea could be anything.