Course Project: Stack Language

Functional Programming (ITI0212)

Due 4th June 2026

The course project is to implement a simple stack langage. 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 the 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%. The project is worth 1/3 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 `iti0212-2026` 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.

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 → Option Nat` that evaluates arithmetic expressions written in reverse Polish notation. 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. You will need to write an implementation of a stack. In the next task, we will add more functionality to this stack language.

Your function must support natural number addition and multiplication. If the input string is not a valid arithmetic expression, your function should return `none`. Otherwise, your function should evaluate the expression and return `some` resulting natural number. For example:

```
#eval eval 3
some 3
#eval eval "3 5 +"
some 8
#eval eval "+ 3 5"
none
#eval eval "3 + 5"
none
#eval eval "5 3 *"
some 15
#eval eval "hello"
none
#eval eval "3 5 + 6 *"
some 48
```

```
#eval eval "3 5 6 * +"
some 33
#eval eval ""
none
#eval eval "3 4 + 5 1 + *"
some 42
```

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. To do this you should use the functions provided by the standard library. In particular the function `IO.FS.readFile` 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

```
$> lean --run StackInterpreter.lean prog.txt
```

to call run our evaluator on the program. Note that you can capture arguments to the program (such as the name of the file, `prog.txt`) by giving your `main` function an argument `def main (args : List String) ...`.

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` 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 `none`.

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 `none`.

Finally, if there are no errors, the return result of the program should be the top element of the stack, if any.

When implementing this task, it's crucial to provide clear and helpful error messages for the user. This ensures that when something goes wrong during the execution, the user can easily identify the issue and correct it.

Some examples:

```
prog.txt = "3 5 r"
$> Please Enter a Symbol: +
$> 8

prog.txt = "3 5 r"
$> Please Enter a Symbol: *
$> 15
```

```
prog.txt = "3 5 r"
$> Please Enter a Symbol: hello
$> Error : Unknown symbol: hello

prog.txt = "3 r +"
$> Please Enter a Symbol: 5
$> 8

prog.txt = "3 r +"
$> Please Enter a Symbol: *
$> Error : Not enough values on stack for binary operation during Read.

prog.txt = "r r r"
$> Please Enter a Symbol: 3
$> Please Enter a Symbol: 5
$> Please Enter a Symbol: *
$> 15

prog.txt = "15 p 5"
$> Print: 15
$> 5

prog.txt = "r p 3"
$> Please Enter a Symbol: 4
$> Print: 4
$> 3

prog.txt = "3 5 * 4 + p 3"
$> Print: 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, `s`, `l` and `ps`. The effect of `s` 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 `l` (*Load*) 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. The effect of `ps` (short for *PrintStore*) is to print the current contents of the global store in a human-readable format. Each key-value pair should be printed on a new line.

Hint: you may use a `HashMap` to model the global store. Add `import Std.Data.HashMap` to the top of your file.

For example:

```
1 → 42
5 → 5
10 → 2
```

Or a prettier version:

```
position | 1  | 5  | 10
---------+----+----+----
contents | 42 | 5  | 2
```

If the store is empty, nothing should be printed.

For example:

```
prog.txt = "42 4 s 4 l"
$> 42

prog.txt = "42 4 s 5 l"
$> Error : Key not found in store.

$> prog.txt = "1 5 s 3 4 + 5 l *"
$> 7

$> prog.txt = "5 1 s 2 2 s 10 1 l +"
$> 15

$> prog.txt "5 1 s 2 2 s 10 1 l + 2 l *"
$> 30

$> prog.txt "2 1 s 1 l 1 l 1 l 1 l * * *"
$> 16

$> prog.txt = "2 1 s 3 1 s 1 l 1 l * ps"
$> position | 1
   ---------+----
   contents | 3
   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 s` 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 s` would be to store 7 in position 3:

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

and the effect of, say, `1 l` 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 are encouraged to come up with your own ideas about what you would like your stack language to do. You may 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. Here are a few possible ideas that you may use:

1. Add support for scoped/local variables.
2. Add support for simple control flow based on stack values (e.g. if statements).
3. Formulate and prove properties about certain stack programs.
4. Make your language into a Turing-complete model of computation. See for example the Joy language (`http://joy-lang.org/`), or Forth.
5. 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.