



~

GIROT-PERSON Claire  
[claire.girot-person@imt-atlantique.net](mailto:claire.girot-person@imt-atlantique.net)

JAUJAY Augustin  
[augustin.jaujay@imt-atlantique.net](mailto:augustin.jaujay@imt-atlantique.net)

## Compilation project

UE LaLog



**IMT Atlantique**  
Bretagne-Pays de la Loire  
École Mines-Télécom

## Contents

<b>1</b>	<b>Exercise 1</b>	<b>3</b>
<b>2</b>	<b>Exercise 2</b>	<b>3</b>
<b>3</b>	<b>Exercise 3</b>	<b>3</b>
3.1	.....	4
3.2	.....	4
3.3	.....	4
<b>4</b>	<b>Exercise 4</b>	<b>6</b>
4.1	.....	6
4.2	.....	6
<b>5</b>	<b>Exercise 5</b>	<b>6</b>
5.1	.....	6
5.2	.....	7
<b>6</b>	<b>Exercise 6</b>	<b>7</b>
<b>7</b>	<b>Exercise 7</b>	<b>7</b>
<b>8</b>	<b>Exercise 8</b>	<b>7</b>
<b>9</b>	<b>Exercise 9</b>	<b>7</b>
9.1	.....	7
9.2	.....	7
9.3	.....	8
<b>10</b>	<b>Exercise 10</b>	<b>8</b>
10.1	.....	8
10.2	.....	9
10.3	.....	9
10.4	.....	9

## Introduction

The git repository of our project is available at <https://gitlab.imt-atlantique.fr/a19jauja/compilerlalog>.

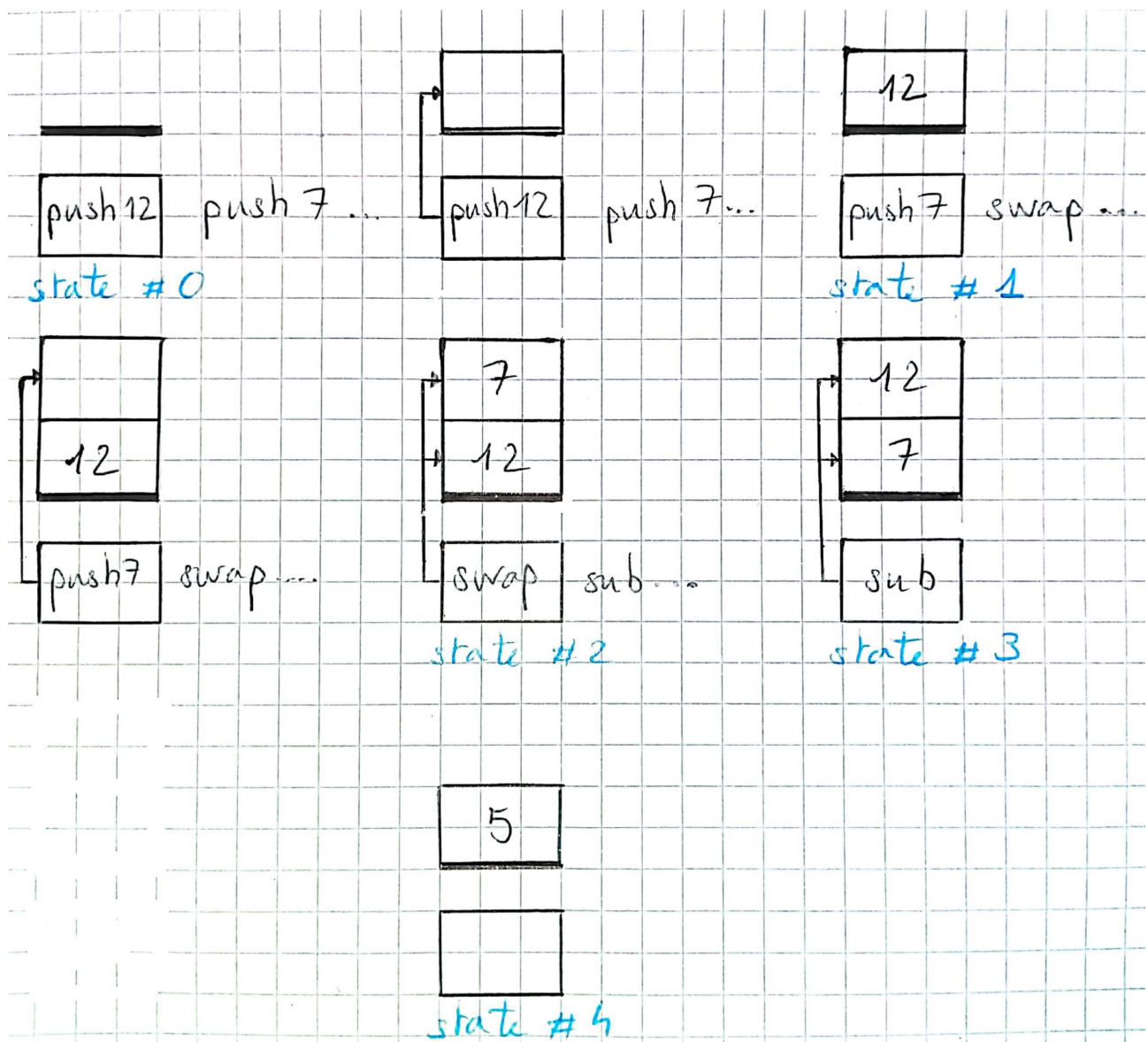
```
git clone git@gitlab.imt-atlantique.fr:a19jauja/compilerlalog.git
```

## 1 Exercise 1

A stack is an ordered set of elements that are only accessible by the top. The two main operations associated to this data structure are the following :

- push(x), which adds an element x at the top of stack,
- pop() which deletes and returns the element at the top of the list.

## 2 Exercise 2



## 3 Exercise 3

### 3.1

- (1) : if the number of arguments is different than the announced number  $i$ , then Pfx should return an error,
- (2) : if any instruction in the instruction sequence returns an error, Pfx should return an error,
- (3) : when the instruction sequence is empty, the program is done and Pfx should return the value on top of the stack.

### 3.2

$$\frac{Q, v_1 :: \dots :: v_n :: \emptyset \rightarrow^* \emptyset, \emptyset}{v_1, \dots, v_n \vdash n, Q \implies \text{None}}$$

When both the stack and the instruction sequence are empty, None is returned.

### 3.3

Push :

$$\frac{}{v_1, \dots, v_n \vdash \text{push}(x).Q, S \rightarrow Q, x :: S}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{push}(\emptyset).Q, S \rightarrow \text{Err}}$$

Pop :

$$\frac{}{v_1, \dots, v_n \vdash \text{pop}.Q, u :: S \rightarrow Q, S}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{pop}.Q, \emptyset \rightarrow \text{Err}}$$

Swap :

$$\frac{}{v_1, \dots, v_n \vdash \text{swap}.Q, u_1 :: u_2 :: S \rightarrow Q, u_2 :: u_1 :: S}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{swap}.Q, u :: \emptyset \rightarrow \text{Err}}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{swap}.Q, \emptyset \rightarrow \text{Err}}$$

Add :

$$\frac{}{v_1, \dots, v_n \vdash \text{add}.Q, u_1 :: u_2 :: S \rightarrow Q, (u_1 + u_2) :: S}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{add}.Q, u :: \emptyset \rightarrow \text{Err}}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{add}.Q, \emptyset \rightarrow \text{Err}}$$

Sub :

$$\frac{}{v_1, \dots, v_n \vdash \text{sub}.Q, u_1 :: u_2 :: S \rightarrow Q, (u_1 - u_2) :: S}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{sub}.Q, u :: \emptyset \rightarrow \text{Err}}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{sub}.Q, \emptyset \rightarrow \text{Err}}$$

Mul :

$$\frac{}{v_1, \dots, v_n \vdash \text{mul}.Q, u_1 :: u_2 :: S \rightarrow Q, (u_1 \times u_2) :: S}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{mul}.Q, u :: \emptyset \rightarrow \text{Err}}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{mul}.Q, \emptyset \rightarrow \text{Err}}$$

Div :

$$\frac{}{v_1, \dots, v_n \vdash \text{div}.Q, u_1 :: u_2 :: S \rightarrow Q, \frac{u_1}{u_2} :: S}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{div}.Q, u :: \emptyset \rightarrow \text{Err}}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{div}.Q, \emptyset \rightarrow \text{Err}}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{div}.Q, u_1 :: 0 :: S \rightarrow \text{Err}}$$

Rem :

$$\frac{}{v_1, \dots, v_n \vdash \text{rem}.Q, u_1 :: u_2 :: S \rightarrow Q, (u_1 \% u_2) :: S}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{rem}.Q, u :: \emptyset \rightarrow \text{Err}}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{rem}.Q, \emptyset \rightarrow \text{Err}}$$

$$\frac{}{v_1, \dots, v_n \vdash \text{rem}.Q, u_1 :: 0 :: S \rightarrow \text{Err}}$$

## 4 Exercise 4

### 4.1

Please refer to tag "question\_4.1" in the git repository.

```
git checkout question_4.1
```

### 4.2

Please refer to tag "question\_4.2".

```
git checkout question_4.2
```

## 5 Exercise 5

### 5.1

$$\frac{}{S \vdash \text{Const}(x) \rightsquigarrow \text{push}(x)}$$

$$\frac{S \vdash a \rightsquigarrow p_1 \quad S \vdash b \rightsquigarrow p_2}{S \vdash \text{Binop}(\text{Badd}, a, b) \rightsquigarrow p_2 p_1 \text{ add}}$$

$$\frac{S \vdash a \rightsquigarrow p_1 \quad S \vdash b \rightsquigarrow p_2}{S \vdash \text{Binop}(\text{Bsub}, a, b) \rightsquigarrow p_2 p_1 \text{ sub}}$$

$$\frac{S \vdash a \rightsquigarrow p_1 \quad S \vdash b \rightsquigarrow p_2}{S \vdash \text{Binop}(\text{Bmul}, a, b) \rightsquigarrow p_2 p_1 \text{ mul}}$$

$$\frac{S \vdash a \rightsquigarrow p_1 \quad S \vdash b \rightsquigarrow p_2}{S \vdash \text{Binop}(\text{Bdiv}, a, b) \rightsquigarrow p_2 p_1 \text{ div}}$$

$$\frac{S \vdash a \rightsquigarrow p_1 \quad S \vdash b \rightsquigarrow p_2}{S \vdash \text{Binop}(\text{Bmod}, a, b) \rightsquigarrow p_2 p_1 \text{ rem}}$$

$$\frac{S \vdash x \rightsquigarrow p}{S \vdash \text{Uminus}(x) \rightsquigarrow p \text{ push}(0) \text{ sub}}$$

### 5.2

Please refer to tag "question\_5.2".

```
git checkout question_5.2
```

## 6 Exercise 6

Please refer to tag "question\_6".

```
git checkout question_6
```

## 7 Exercise 7

Please refer to tag "question\_7".

```
git checkout question_7
```

## 8 Exercise 8

Please refer to tag "question\_8".

```
git checkout question_8
```

## 9 Exercise 9

### 9.1

No, there is no need to modify the rules defined previously : we just need to add rules to handle the "App" and "Fun" expressions. Since "App" and "Fun" only depend on previously defined expression types, we do not need to add anything else.

### 9.2

$$\frac{Q_0 \text{ is an instruction sequence}}{v_1, \dots, v_n \vdash [Q_0].Q, S \rightarrow Q, Q_0 :: S}$$

$$\begin{array}{c}
\frac{Q_0 \text{ is an instruction sequence}}{v_1, \dots, v_n \vdash \text{exec}.Q, Q_0 :: S \rightarrow Q_0 @ Q, S} \\
\\
\frac{}{v_1, \dots, v_n \vdash \text{exec}.Q, [] \rightarrow \text{Err}} \\
\\
\frac{c \in \{\text{add}, \text{sub}, \text{mul}, \text{div}, \text{rem}\}, Q_0 \text{ is an instruction sequence}}{v_1, \dots, v_n \vdash c.Q, Q_0 :: S \rightarrow \text{Err}} \\
\\
\frac{c \in \{\text{add}, \text{sub}, \text{mul}, \text{div}, \text{rem}\}, Q_0 \text{ is an instruction sequence}}{v_1, \dots, v_n \vdash c.Q, :: Q_0 :: S \rightarrow \text{Err}} \\
\\
\frac{}{v_1, \dots, v_n \vdash \text{get}.Q, i :: x_1 :: \dots :: x_n :: [] \rightarrow Q, x_i :: x_1 :: \dots :: x_n :: []} \\
\\
\frac{j < i}{v_1, \dots, v_n \vdash \text{get}.Q, i :: x_1 :: \dots :: x_j :: [] \rightarrow \text{Err}} \\
\\
\frac{j < i}{v_1, \dots, v_n \vdash \text{get}.Q, i :: x_1 :: \dots :: x_j :: [] \rightarrow \text{Err}} \\
\\
\frac{Q_0 \text{ is an instruction sequence}}{v_1, \dots, v_n \vdash \text{get}.Q, Q_0 :: S \rightarrow \text{Err}}
\end{array}$$

### 9.3

Please refer to tag "question\_9".

```
git checkout question_9
```

## 10 Exercise 10

### 10.1

$(\lambda x.x + 1)2$  is equivalent to

`[Push 2; InstructionSeq [Push 1; Push 1; Get; Add]; Exec; Swap; Pop]`

in Pfx.

First, we add the argument "2" to the stack with the command "Push 2". Then, we translate the function to Pfx as an instruction sequence (as "[Push 1; Push 1; Get; Add]"). The "[Push 1; Get]" subsequence enables us to retrieve the argument we pushed earlier. The whole sequence will be added to the stack. Finally, we call the previously defined function with the command "Exec", and remove the argument with [Swap; Pop] sequence.



## 10.2

$$\frac{\text{env}(v) = i}{S \vdash \text{Var}(v) \rightsquigarrow \text{push}(i) \text{ get}}$$
$$\frac{S \vdash t \rightsquigarrow Q \text{ an instruction sequence}}{S \vdash \text{app}(\text{fun}(v, t), a) \rightsquigarrow \text{push}(a) Q \text{ exec swap pop}}$$

env is the environment function, which associates the index in the stack for a given argument.

## 10.3

Please refer to tag "question\_10".

```
git checkout question_10
```

Unfortunately, we are not able to handle the following cases :

- when the argument of an `App(Fun(⟦,⟦),⟦)` is another `App(Fun(⟦,⟦),⟦)`,
- when the argument has the type `Binop(⟦,⟦)`.

In these cases, we need to decrement the indexes in the environment to handle the next calls of variables.

## 10.4

The compiled version of  $((\lambda x. \lambda y. (x - y))12)8$  is the following :

```
Push 8 [Push 12 [Push 0 Get Push 2 Get Sub] Exec Swap Pop]
```

```
Exec Swap Pop
```

First we push the argument of the x-lambda function. Meanwhile, we add the variable x associated to this argument in the environment. The environment stores and updates the indexes of all variable values in the stack. Then, we deal with the content of the function : it leads us to deal with the y-lambda function as an Instruction Sequence. We proceed the same way as we did with the x-lambda function, by pushing the argument and indexing the variable y. The content of the y-lambda function is also translated to Instruction Sequence and will be pushed on the stack when encountered in the execution sequence of the program. Finally, the Instruction Sequences are called by Exec, and the useless arguments are cleared by Swap and Pop.