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

Compte rendu du projet compilation

FAURE Augustin
GILLARD Antonino

UE F - LALOG
Année 2025

1 Exercice 1

A stack is a data structure that follows the LIFO (Last In, First Out) principle. There are two main operations on a stack :
Push : Adds an element to the top of the stack.
Pop : Removes the most recently added element from the top of the stack.

2 Exercice 2

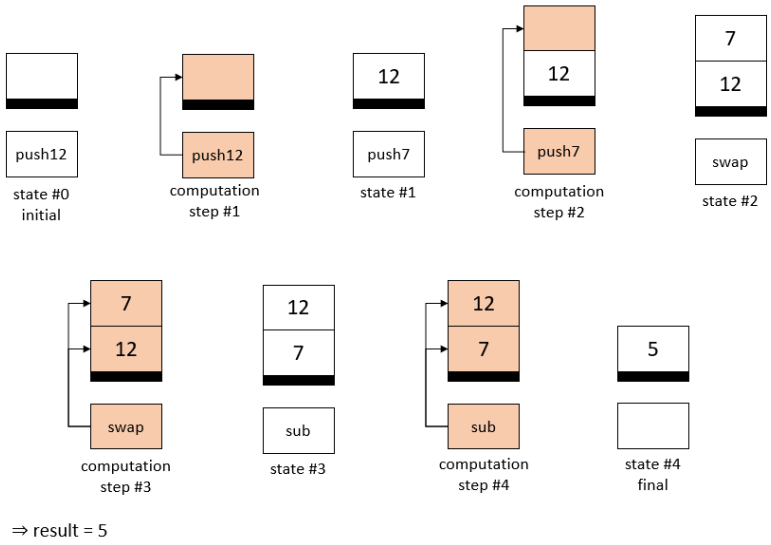


FIGURE 1 – execution of 0 push 12 push 7 swap sub

3 Exercice 3

3.1 Exercice 3.1

(1) In the context of n arguments entered, with a program that awaits i arguments, if n and i are not equal then the machine returns an error.

(2) In the context of n arguments $v_1, \ldots, v_n$, with a program that awaits n arguments and with the instruction sequence Q, if we know that during the execution of instruction sequence Q with the stack $v_1 :: \ldots :: v_n :: \emptyset$ an error is raised then the machine returns an error.

(3) In the context of n arguments $v_1, \ldots, v_n$, with a program that awaits n arguments and with the instruction sequence Q, if we know that after the execution of instruction sequence Q, the stack $v_1 :: \ldots :: v_n :: \emptyset$ becomes a non-empty stack with the first element being v then the machine returns v.

3.2 Exercice 3.2

(4) In the context of n arguments $v_1, \ldots, v_n$, with a program that awaits n arguments and with the instruction sequence Q, if we know that after the execution of instruction sequence Q, the stack $v_1 :: \ldots :: v_n :: \emptyset$ becomes an empty stack then the machine returns an error.

$$\frac{Q, v_1 :: \ldots :: v_n :: \emptyset \to \emptyset, \emptyset}{v_1, \ldots, v_n \vdash n, Q \implies ERR}$$

3.3 Exercice 3.3

Here are the rules describing the small step semantics for instruction sequences.

$$\begin{array}{rr}
push\ v.Q, S \to Q, v :: S & pop.Q, \emptyset \to ERR \\
pop.Q, v :: S \to Q, S & swap.Q, v :: \emptyset \to ERR \\
swap.Q, v_1 :: v_2 :: S \to Q, v_2 :: v_1 :: S & swap.Q, \emptyset \to ERR
\end{array}$$

$$\begin{array}{rr}
add.Q, v_1 :: v_2 :: S \to Q, (v_1 + v_2) :: S & sub.Q, v_1 :: v_2 :: S \to Q, (v_1 - v_2) :: S \\
mul.Q, v_1 :: v_2 :: S \to Q, (v_1 * v_2) :: S & div.Q, v_1 :: v_2 :: S \to Q, (v_1/v_2) :: S \\
rem.Q, v_1 :: v_2 :: S \to Q, (v_1\ mod\ v_2) :: S &
\end{array}$$

$$\begin{array}{rr}
add.Q, v :: \emptyset \to ERR & add.Q, \emptyset \to ERR \\
sub.Q, v :: \emptyset \to ERR & sub.Q, \emptyset \to ERR \\
mul.Q, v :: \emptyset \to ERR & mul.Q, \emptyset \to ERR \\
div.Q, v :: \emptyset \to ERR & div.Q, \emptyset \to ERR \\
rem.Q, v :: \emptyset \to ERR & rem.Q, \emptyset \to ERR
\end{array}$$

4 Exercice 5

4.1 Exercice 5.1

$$Const\,(v) \rightsquigarrow push\ v$$

$$\frac{exp_1 \rightsquigarrow q_1 \qquad exp_2 \rightsquigarrow q_2}{Binop\,(Badd, exp_1, exp_2) \rightsquigarrow q2.q1.add} \qquad \frac{exp_1 \rightsquigarrow q_1 \qquad exp_2 \rightsquigarrow q_2}{Binop\,(Bsub, exp_1, exp_2) \rightsquigarrow q2.q1.sub}$$

$$\frac{exp_1 \rightsquigarrow q_1 \qquad exp_2 \rightsquigarrow q_2}{Binop\,(Bmul, exp_1, exp_2) \rightsquigarrow q2.q1.mul} \qquad \frac{exp_1 \rightsquigarrow q_1 \qquad exp_2 \rightsquigarrow q_2}{Binop\,(Bdiv, exp_1, exp_2) \rightsquigarrow q2.q1.div}$$

$$\frac{exp_1 \rightsquigarrow q_1 \qquad exp_2 \rightsquigarrow q_2}{Binop\,(Bmod, exp_1, exp_2) \rightsquigarrow q2.q1.rem} \qquad \frac{exp_1 \rightsquigarrow q_1}{Uminus\,(exp_1,) \rightsquigarrow q1.push\ 0.div}$$

5 Exercice 9

5.1 Exercice 9.1

We do need to redefine the rules for the already established constructs. Previously, the stack operated solely on integers, making constructs like ADD fully compatible with the current system. With the introduction of executable sequences, the stack now holds two types of objects : integers and executable sequences. Consequently, operations such as ADD must incorporate checks to ensure that they are applied only when the top elements of the stack are integers, as these operations are incompatible with executable sequences.

5.2 Exercice 9.2

Here are the new formal semantics in Pfx for function handling (exec, get, execSeq).

$$execSeq\,(Q_1)\,.Q_2, S \to Q_2, execSeq\,(Q_1) :: S \qquad exec.Q, \emptyset \to ERR$$
$$get.Q, \emptyset \to ERR \qquad exec.Q_2, execSeq\,(Q_1) :: S_1 \to Q_2, S_2$$

$$\frac{Q_1, S_1 \to \emptyset, S_2}{exec.Q_2, execSeq\,(Q_1) :: S_1 \to Q_2, S_2} \qquad \frac{Q_1, S_1 \to ERR}{exec.Q_2, execSeq\,(Q_1) :: S_1 \to ERR}$$

$$\frac{v \in \mathbb{Z}}{exec.Q, v :: S \to ERR} \qquad \frac{i \in \mathbb{N}}{get.Q, i :: v_0 :: \ldots :: v_i :: S \to Q, v_i :: v_0 :: \ldots :: v_i :: S}$$

$$\frac{i \notin [\![0, \ldots, |S| - 1]\!]}{get.Q, i :: S \to ERR} \qquad \frac{i \in \mathbb{Z}}{push\ v.Q, S \to Q, v :: S}$$

$$\frac{i \notin \mathbb{Z}}{push\ v.Q, S \to ERR} \qquad \frac{v_1, v_2 \in \mathbb{Z}}{add.Q, v_1 :: v_2 :: S \to Q, (v_1 + v_2) :: S}$$

$$\frac{v_1, v_2 \in \mathbb{Z}}{sub.Q, v_1 :: v_2 :: S \to Q, (v_1 - v_2) :: S} \qquad \frac{v_1, v_2 \in \mathbb{Z}}{mul.Q, v_1 :: v_2 :: S \to Q, (v_1 * v_2) :: S}$$

$$\frac{v_1, v_2 \in \mathbb{Z}}{div.Q, v_1 :: v_2 :: S \to Q, (v_1/v_2) :: S} \qquad \frac{v_1, v_2 \in \mathbb{Z}}{rem.Q, v_1 :: v_2 :: S \to Q, (v_1\ mod\ v_2) :: S}$$

$$\frac{(v_1, v_2) \notin \mathbb{Z} * \mathbb{Z}}{add.Q, v_1 :: v_2 :: S \to ERR} \qquad \frac{(v_1, v_2) \notin \mathbb{Z} * \mathbb{Z}}{sub.Q, v_1 :: v_2 :: S \to ERR}$$

$$\frac{(v_1, v_2) \notin \mathbb{Z} * \mathbb{Z}}{mul.Q, v_1 :: v_2 :: S \to ERR} \qquad \frac{(v_1, v_2) \notin \mathbb{Z} * \mathbb{Z}}{div.Q, v_1 :: v_2 :: S \to ERR}$$

$$\frac{(v_1, v_2) \notin \mathbb{Z} * \mathbb{Z}}{rem.Q, v_1 :: v_2 :: S \to ERR}$$

6 Exercice 10

6.1 Exercice 10.1

The compiled version of $(\lambda x.x + 1)\, 2$ is

push2 exec_seq(push1 push1 get add) exec swap pop.

The evaluation of its Pfx translation proceeds as follows :

push2 : Push the argument 2 onto the stack $\to [2]$.
exec_seq : Push the executable sequence that defines the execution sequence of the lambda function $\to$ [execSeq(push1 push1 get add), 2].
push1 push1 : Push 1 onto the stack twice $\to [1, 1, x]$. The first is for the 1 used in the addition, and the second is popped to access the argument via **get**.
get : Retrieve the value of x from the stack $\to [x, 1, x]$.
add : Perform the addition as defined in the function $\to [(x + 1), x]$.
exec : Pop and execute the executable sequence that was just pushed $\to [2] \to [1, 2] \to [1, 1, 2] \to [2, 1, 2] \to [3, 2]$.
swap : Swap the two elements in the stack $\to [2, 3]$.
pop : Remove the top element of the stack, which is now the argument 2 after the swap $\to [3]$.
The remaining element in the stack is 3, confirming it as the result.

6.2 Exercice 10.2

Here are the formal rules of Expr to Pfx translation for function handling.

$$\frac{P \vdash exp \rightsquigarrow q}{Fun(var, exp) \rightsquigarrow execSeq(q)}$$

$$\frac{P(var) = n}{P \vdash Var(var) \rightsquigarrow push\, n; get}$$

$$\frac{P(var) = \bot}{P \vdash Var(var) \rightsquigarrow ERR}$$

$$\frac{P \vdash exp1 \rightsquigarrow q1 \qquad P \vdash exp2 \rightsquigarrow q2}{P \vdash App(exp1, exp2) \rightsquigarrow q2.q1.exec.swap.pop} \; exp1 \in Function$$

$$\frac{exp1 \notin Function}{P \vdash App(exp1, exp2) \rightsquigarrow ERR}$$

6.3 Exercice 10.4

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

push12 exec_seq(push8 exec_seq(push0 get push2 get sub) exec swap pop) exec swap pop.

The evaluation of its Pfx translation proceeds as follows :

push12 : Push the first argument 12 onto the stack $\rightarrow [12]$.
exec_seq : Push the executable sequence that defines the execution sequence of the lambda function on the first argument $x \rightarrow$ [exec_seq(push8 exec_seq(push0 get push2 get sub) exec swap pop), 12].
push8 : Push the second argument 8 onto the stack $\rightarrow [8, x]$.
exec_seq : Push the executable sequence that defines the execution sequence of the lambda function on the second argument $y \rightarrow$ [exec_seq(push0 get push2 get sub), $8, x$].
push0 : Push a 0 that will be popped to access the second argument via get $\rightarrow [0, y, x]$.
get : Retrieve the value of y from the stack $\rightarrow [y, y, x]$.
push2 : Push a 2 that will be popped to access the first argument via get $\rightarrow [2, y, y, x]$.
get : Retrieve the value of x from the stack $\rightarrow [x, y, y, x]$.
sub : Pop the retrieved arguments and perform the subtraction as defined in the function $\rightarrow [(x - y), y, x]$.
exec : Pop and execute the executable sequence on the second argument :

$$\rightarrow [0, 8, x] \rightarrow [8, 8, x] \rightarrow [2, 8, 8, x] \rightarrow [(x - 8), 8, x].$$

swap pop : Swap the top elements and pop to remove the argument 8, allowing the result to be placed on top of the stack $\rightarrow [8, (x - 8), x] \rightarrow [(x - 8), x]$.

exec : Pop and execute the executable sequence on the first argument :

$$\begin{aligned}&\to [8, 12] \to [\text{exec\_seq}(\text{push0 get push2 get sub}), 8, 12]\\ &\quad\to [0, 8, 12] \to [8, 8, 12] \to [2, 8, 8, 12] \to [4, 8, 12] \to [8, 4, 12] \to [4, 12].\end{aligned}$$

swap pop : Swap and pop to remove the argument 12, placing the result on top of the stack $\to [12, 4] \to [4]$.

The remaining element in the stack is 4, confirming it as the result.

7 Exercice 11

7.1 Exercice 11.1

In the syntax of **Expr** already defined, the construct

$$\text{let } x = e_1 \text{ in } e_2$$

can be translated as :

$$\text{App(Fun(“x”, } e_2\text{), } e_1\text{).}$$

7.2 Exercice 11.2

To support the **let ... in ...** construct, the lexer and parser of **Expr** must be modified. The lexer must be updated to recognize the **let** and **in** keywords, and the parser must be adjusted to translate **let ... in ...** into its functional equivalent :

$$\text{App(Fun("x", } e_2\text{), } e_1\text{).}$$

8 Exercice 12

To derive the term $((\lambda x.\lambda y.(x - y))\, 12)\, 8$ and compute its value, we use the formal semantics of **Expr** for function applications. The next step involves performing the same computation for the application $(\lambda x.\lambda y.(x - y))\, 12$. As we will see, this application will produce an environment $\{\varepsilon_y, y, x - y\}$, and by computing the constant 8, we apply the environment $\varepsilon_y, y \mapsto 8$ to evaluate $x - y$, which gives the result 4. This confirms that the final result is indeed 4.

Root and complete deduction

$$\frac{\{\} \vdash ((\lambda x.\lambda y.(x - y))\ 12) \Rightarrow \{\varepsilon_y, y, x - y\} \qquad \{\} \vdash 8 \Rightarrow 8 \qquad \varepsilon_y, y \to 8 \vdash x - y \Rightarrow 4}{\{\} \vdash (((\lambda x.\lambda y.(x - y))\ 12)\ 8) \Rightarrow 4}$$

By deriving $(\lambda x.\lambda y.(x - y))\, 12$, we obtain the environment $\{\varepsilon_x, x, \lambda y.(x - y)\}$ by evaluating the function $\lambda x.\lambda y.(x - y)$ with $\text{dom}(\varepsilon_x) = \{x\}$. After computing the constant 12, we get the environment $\varepsilon_x, x \mapsto 12$, and apply it to $\lambda y.(x - y)$, resulting in a new environment $\{\varepsilon_y, y, x - y\}$ with $\text{dom}(\varepsilon_y) = \{x, y\}$ and $\varepsilon_y(x) = 12$.

Deduction of the environment

$$\dfrac{\dfrac{\mathrm{dom}(\varepsilon_x) = \{x\}}{\{\} \vdash (\lambda x.\lambda y.(x-y)) \Rightarrow \{\varepsilon_x, x, \lambda y.(x-y)\}} \qquad \{\} \vdash 12 \Rightarrow 12 \qquad \dfrac{\mathrm{dom}(\varepsilon_y) = \{x,y\},\ \varepsilon_y(x) = 12}{\varepsilon_x, x \to 12 \vdash \lambda y.(x-y) \Rightarrow \{\varepsilon_y, y, x-y\}}}{\{\} \vdash ((\lambda x.\lambda y.(x-y))\ 12) \Rightarrow \{\varepsilon_y, y, x-y\}}$$

Given the environment ε_y, we now know that $\varepsilon_x, x \mapsto 12$ is equivalent to $\{x \mapsto 12; y \mapsto 8\}$. As a result, the computation of $x - y$ becomes straightforward by applying the environment to the variables, yielding the result $12 - 8 = 4$.

Operation of substraction using the environment

$$\dfrac{\{\ x \to 12; y \to 8\} \vdash x \Rightarrow 12 \qquad \{\ x \to 12; y \to 8\} \vdash y \Rightarrow 8}{\{\ x \to 12; y \to 8\} \vdash x - y \Rightarrow 12 - 8 = 4}$$

9 Exercice 13

9.1 Exercice 13.1

Translation from Expr to Pfx is possible but would require to modify a few things to the way we handle executable sequence in the stack. In order to capture its free variables, we would first need to analyse the function body to check for any free variable. Then, instead of having the executable sequence at the top of the stack then its arguments, we would need to use PUSH / GET commands to copy the values of the free variables near the executable sequence. For example, we could first push the argument of the function, then the free variables, then the executable sequence. As a result of this, when we translate the function body we must change the mapping of each free variable in the stack to the position of the copied values we put just beneath the function body.

9.2 Exercice 13.2

Here are the formal semantics of append :

$$\begin{gathered} \mathit{append}.Q, \emptyset \to \mathit{ERR} \\ \mathit{append}.Q, v :: \emptyset \to \mathit{ERR} \\ \mathit{append}.Q_1, \mathit{execSeq}\,(Q_2) :: \mathit{execSeq}\,(Q_3) :: S \to Q_1, \mathit{execSeq}\,(Q_2.Q_3) :: S \end{gathered}$$

$$\dfrac{v \in \mathbb{Z}}{\mathit{append}.Q_1, v :: \mathit{execSeq}\,(Q_2) :: S \to Q_1, \mathit{execSeq}(\mathit{push}\ v.Q_2) :: S}$$

$$\frac{v \in \mathbb{Z}}{append.Q, v_0 :: v :: S \to ERR}$$

9.3 Exercice 13.4

Here are the new formal rules of Expr to Pfx translation :

$$\frac{\mathcal{E} \vdash exp_1 \rightsquigarrow q_1 \qquad \mathcal{E} \vdash exp_2 \rightsquigarrow q_2 \qquad exp_1 \in Function \quad FV(exp_1) = \{y_1, \ldots, y_n\}}{\mathcal{E} \vdash App(exp_1, exp_2) \rightsquigarrow q_2.q_1.exec.\,(swap.pop.)^n\, swap.pop}$$

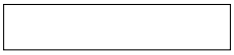
$$\frac{P \vdash exp \rightsquigarrow q \qquad FV(exp) = \{y_1, \ldots, y_n\} \qquad \forall i \in [1, n],\ P(y_i) = v_i}{P \vdash Fun(var, exp) \rightsquigarrow execSeq(q).push(v_1).get.append \ldots push(v_n).get.append}$$

9.4 Exercice 13.6

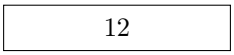
The compiled version of $((\lambda x.\lambda y.(x - y))\, 12)\, 8$ is :

```
push12
exec_seq(
  push8
  exec_seq(
    push1 get push1 get sub
  )
  push2 get append
  exec swap pop swap pop
)
exec swap pop
```

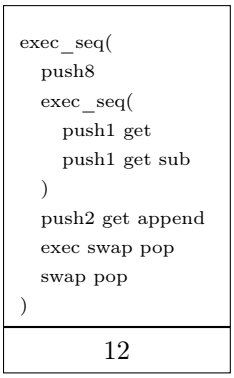
The evaluation of its Pfx translation proceeds as follows :



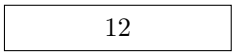
Step #0 : Empty stack



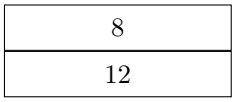
Step #1 : push(12)



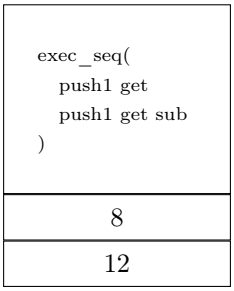
Step #2 : Sequence 1



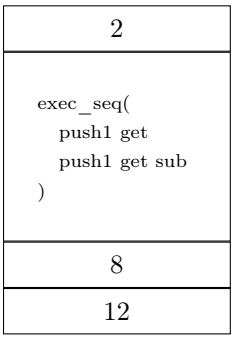
Step #3 : exec



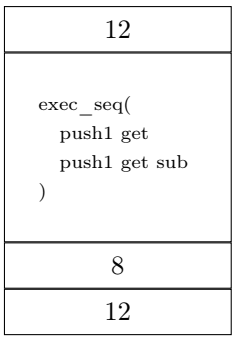
Step #4 : push(8)



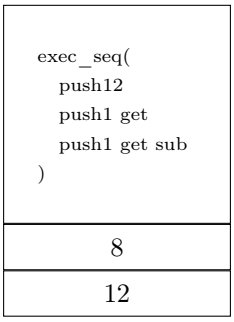
Step #5 : Sequence 2



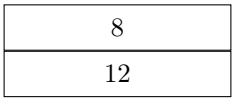
Step #6 : push(2)



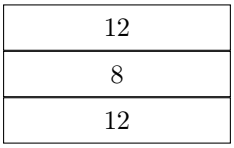
Step #7 : get



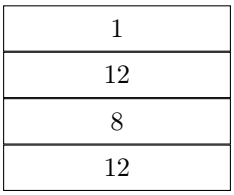
Step #8 : append



Step #9 : exec

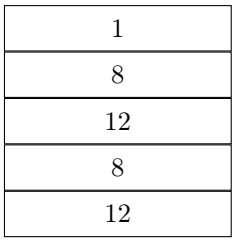


Step #10 : push(12)

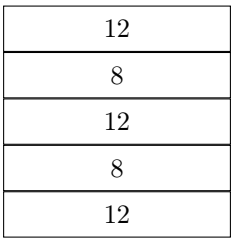


Step #11 : push(1)

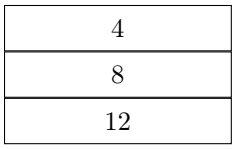
Step #12 : get



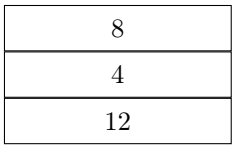
Step #13 : push(1)



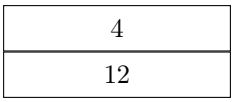
Step #14 : get



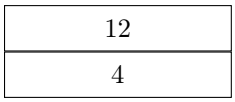
Step #15 : sub



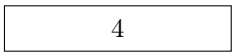
Step #16 : swap



Step #17 : pop



Step #18 : swap



Step #19 : pop

Both versions work so we will have to address the differences of each translation. The second translation requires more memory space (more commands and more room in the stack) compared to the first translation. However, the second one stores and copies all the values of the free variable at runtime. As such, if one were to modify the original value of the free variable after the exec command of an executable sequence using that value, it will have no impact on the program.

Thus, the last translation is better.