UE – Langage et Logique

Formation Ingénieur sous Statut Étudiant

Prof encadrant : J-C Bach

Compilation Project

Étudiants:

CHAHBOUNE Nawal
TAF:DCL

BEN AMER YASMINE
TAF:DCL

Exercise 1:

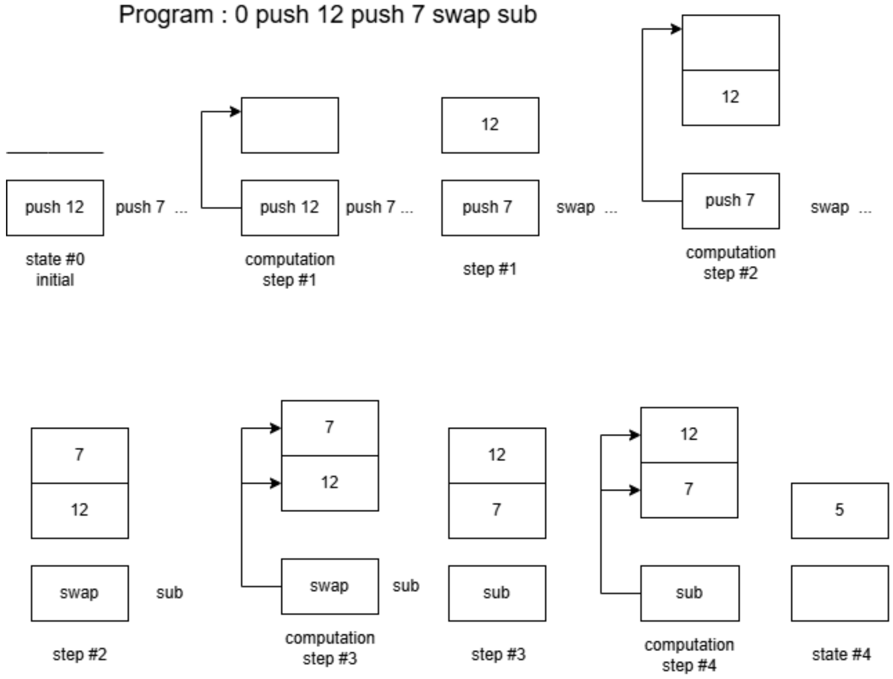
A stack is a linear data structure that operates on the principle of Last In, First Out (LIFO). This means the last element added to the stack will be the first one to be removed. It is commonly used in programming for tasks like function call management, undo mechanisms, and parsing expressions.

Basic Operations on a Stack

1. Push: Adds a new element to the top of the stack.
2. Pop: Removes and returns the top element from the stack.
3. Peek (or Top): Retrieves the topmost element without removing it.
4. isEmpty: Checks whether the stack is empty.
5. isFull: Checks if the stack has reached its maximum capacity.
6. Size: Returns the number of elements currently in the stack.

Exercise 2:

Program : 0 push 12 push 7 swap sub



Result : 5

Exercise 3:

3-1

The semantic of the program:

$$(1)\ \frac{i \neq n}{v_1, \ldots, v_n \vdash i, Q \Rightarrow \textsc{Err}}$$

A program fails if the number of arguments provided does not match the number of expected arguments.

The semantic of the program:

$$(2)\ \frac{Q, v_1 :: \ldots :: v_n :: \varnothing \rightarrow^* \textsc{Err}}{v_1, \ldots, v_n \vdash n, Q \Rightarrow \textsc{Err}}$$

If the execution of an instruction sequence Q with an initial stack containing values v*1* to v*n* leads to an error (after zero or more steps), then the complete program with n arguments and the instruction sequence Q also results in an error. In other words, if the body of the program produces an error during execution, the entire program fails.

$$(3)\ \frac{Q, v_1 :: \ldots :: v_n :: \varnothing \rightarrow^* \varnothing, v :: S}{v_1, \ldots, v_n \vdash n, Q \Rightarrow v}$$

The semantic of the program:

This rule indicates that if the execution of an instruction sequence Q with an initial stack containing values v*1* to v*n* terminates normally with a stack whose top element is v, then the result of the complete program is this value v. In other words, the result of a program that executes correctly is the value at the top of the stack at the end of execution.

3-2

$$\frac{Q, v_1 :: \ldots :: v_n \rightarrow^* \emptyset, v :: S, \quad \#S \geq 1}{v_1, \ldots, v_n \vdash n, Q \Rightarrow \mathrm{ERR}}$$

A new rule might be needed to handle a case where execution terminates, but the stack does not reduce to a single value.
For example, the stack might contain multiple values instead of a single result, which could indicate an undefined result or incorrect program behavior.

3.3

push: If the next instruction is n (a number), it is pushed onto the stack, and execution continues with Q.

$$n.Q, S \rightarrow Q, n :: S$$

add: The ADD instruction pops two values from the stack, sums them, and pushes the result.

$$ADD.Q, v_1 :: v_2 :: S \rightarrow Q, (v_1 + v_2) :: S$$

multiply: The MUL instruction pops two values, multiplies them, and pushes the result.

$$MUL.Q, v_1 :: v_2 :: S \rightarrow Q, (v_1 \times v_2) :: S$$

subtract: The SUB instruction pops two values, subtracts them, and pushes the result.

(The order matters: top value is subtracted from the second top value.)

$$SUB.Q, v_1 :: v_2 :: S \rightarrow Q, (v_2 - v_1) :: S$$

swap: The SWAP instruction swaps the top two values on the stack.

$$SWAP.Q, v_1 :: v_2 :: S \rightarrow Q, v_2 :: v_1 :: S$$

pop: The POP instruction removes the top value from the stack

$$POP.Q, v_1 :: S \rightarrow Q, S$$

div: The DIV instruction divides the second value on the stack by the top value

$$\frac{n_1 \neq 0}{DIV.Q, n_1 :: n_2 :: S \rightarrow Q, (n_2 \div n_1) :: S}$$

Runtime Error rules:

-ADD:

$$ADD.Q, n :: \emptyset \rightarrow Err$$

$$ADD.Q, \emptyset \rightarrow Err$$

-MUL

$$MUL.Q, n :: \emptyset \rightarrow Err$$

$$MUL.Q, \emptyset \rightarrow Err$$

-SUB:

$$SUB.Q, n :: \emptyset \rightarrow Err$$

$$SUB.Q, \emptyset \rightarrow Err$$

-SWAP:

$$swap.Q, n :: \emptyset \rightarrow Err$$

$$swap.Q, \emptyset \rightarrow Err$$

-POP:

$$pop.Q, \emptyset \rightarrow Err$$

-DIV:

$$div.Q, n :: \emptyset \rightarrow Err$$

$$div.Q, \emptyset \rightarrow Err$$

Exercise 4:

---

see code in files: pfx/basic/ast.ml and pfx/basic/eval.ml

Exercise 5:

5-1:

$$\text{Expr} ::= n \mid \text{Expr} + \text{Expr} \mid \text{Expr} - \text{Expr} \mid \text{Expr} \times \text{Expr} \mid \text{Expr} \div \text{Expr}$$

n is a constant integer,
+, -, ×, and ÷ are arithmetic operators,

<u>Compilation Schema:</u>

The compilation function C translates an expression into a Pfx instruction sequence.

$$C(n) = n$$

A constant is pushed in the stack

$$C(e_1 + e_2) = (C(e_1) \cdot C(e_2)) \cdot ADD$$

$$C(e_1 - e_2) = (C(e_1) \cdot C(e_2)) \cdot SUB$$

$$C(e_1 \times e_2) = (C(e_1) \cdot C(e_2)) \cdot MUL$$

$$C(e_1 \div e_2) = (C(e_1) \cdot C(e_2)) \cdot DIV$$

$$C(e_1 \mod e_2) = (C(e_1) \cdot C(e_2)) \cdot MOD$$

$$C((e)) = (C(LPAR) \cdot C(e) \cdot C(RPAR))$$

Addition:

$$\frac{e_1 \to Q_1 \quad e_2 \to Q_2}{e_1 + e_2 \to (Q_1 \cdot Q_2) \cdot ADD}$$

Soustraction :

$$\frac{e_1 \to Q_1 \quad e_2 \to Q_2}{e_1 - e_2 \to (Q_1 \cdot Q_2) \cdot SUB}$$

Multiplication :

$$\frac{e_1 \rightarrow Q_1 \quad e_2 \rightarrow Q_2}{e_1 \times e_2 \rightarrow (Q_1 \cdot Q_2) \cdot MUL}$$

Division :

$$\frac{e_1 \rightarrow Q_1 \quad e_2 \rightarrow Q_2}{e_1 \div e_2 \rightarrow (Q_1 \cdot Q_2) \cdot DIV}$$

Modulo :

$$\frac{e_1 \rightarrow Q_1 \quad e_2 \rightarrow Q_2}{e_1 \mod e_2 \rightarrow (Q_1 \cdot Q_2) \cdot MOD}$$

Parenthèses :

$$\frac{e \rightarrow Q}{(e) \rightarrow (LPAR \cdot Q \cdot RPAR)}$$

5-2
see code of generate function in file: expr/basic/toPfx.ml

Exercise 6:

---

see code

Exercise 7:

See modification in code raise Location error instead of failure.

Exercise 8:

---

see code.

Exercise 9:

Question1:

Yes, the existing rules need to be adjusted because:

1. Stacks now contain both integers and executable sequences (instead of just integers).

2. Operations must distinguish between these types to ensure correct execution.

3. Execution now includes sequences that must be handled properly when they appear on the stack.

Thus, any rule that manipulates stack elements (e.g., arithmetic operations, duplication, and swapping) needs to account for the possibility of sequences as operands.

Question2:

1- Pushes the sequence Q1 as a **first-class value** on the stack

$\langle (Q1) . Q2, S\rangle \rightarrow \langle Q2, Q1 :: S\rangle$

2- **Execution of a Sequence(exec):** Pops the executable sequence Q1 from the stack and prepends it to the current instruction sequence for execution.

$\langle$exec . Q, Q1 :: S$\rangle \rightarrow \langle$Q1 . Q, S$\rangle$ if Q1 is an instruction sequence

3- **Stack Access(get):** Copies the *i-th* element from the stack and pushes it on top

$\langle$get . Q, i :: v_0 :: v_1 :: ... :: v_i :: S'$\rangle \rightarrow \langle$Q, v_i :: v_0 :: v_1 :: ... :: v_i :: S'$\rangle$

Exercise 10:

Question 1:

(λx. x + 1) 2

Step 1**: Compilation Strategy (Expr → Pfx)**

In the Pfx model, functions are compiled into executable sequences with:

- Argument access via get (using de Bruijn indices or positional access)
- Executed with exec

So we'll:

1. Push the argument
2. Push the function body as a sequence
3. exec it

Step 2: **Compilation**

push 2

(**get** 0 ;

push 1

add ; x + 1

)

exec

Steps:

1- Push 2 → push 2 onto stack

⟨(Get . Push 1 . Add) . Exec, 2 :: ∅⟩

2- (Get . Push 1 . Add) → push the sequence onto the stack

⟨Exec, (Get . Push 1 . Add) :: 2 :: ∅⟩

3- Exec → pop executable and prepend it

⟨Get . Push 1 . Add, 2 :: ∅⟩

4- Get → get the 0-th element from the stack (2)

⟨Push 1 . Add, 2 :: 2 :: ∅⟩

5- Push 1 → push 1

⟨Add, 1 :: 2 :: 2 :: ∅⟩

6- Add → add top two integers: 1 + 2 = 3

⟨∅, 3 :: 2 :: ∅⟩

Question2:

(λx. x + 1) 2

Formal Rule:

App(Fun(x, Binop(Add, Var(x), Const(1))), Const(2))

Then the transformation gives:

push 2 . (push 0 . get . push 1 . add) . exec

Which comes from the application of these formal rules:

[APP]

- App(e1, e2) ⟶ [e2] . [e1] . exec

[FUN]

- Fun(x, e) ⟶ ([e with x at index 0])

[VAR]

- Var(x) ⟶ push 0 . get

[CONST]

- Const(n) ⟶ push n

[BINOP]

- e1 + e2 ⟶ [e1] . [e2] . add

<u>Apply rules:</u>

1. Var(x) → push 0 . get
2. Const(1) → push 1
3. x + 1 → push 0 . get . push 1 . add
4. Fun(x, x + 1) → (push 0 . get . push 1 . add)

5. Const(2) → push 2

6. Whole application →
   push 2 . (push 0 . get . push 1 . add) . exec

Question4:

App(App(Fun("x", Fun("y", Binop(Sub, Var("x"), Var("y")))),Const(12)),Const(8))

We'll use the compilation rules from before:

- Fun(x, e) ⟶ ([e with x at index 0])

- Var(x) ⟶ push i . get (*i is the index from top of stack*)

- App(e1, e2) ⟶ [e2] . [e1] . exec

steps of compilation:

we will start from inside out

1- Fun("y", x - y) ⟶ (push 1 . get . push 0 . get . sub)

2- Fun("x", Fun("y", x - y)) ⟶ ((push 1 . get . push 0 . get . sub))

3- App(Fun("x", Fun("y", x - y)), 12)

⟶ push 12 . ((push 1 . get . push 0 . get . sub)) . exec

4- App(App(Fun("x", Fun("y", x - y)), 12), 8)

⟶ push 8 . push 12 . ((push 1 . get . push 0 . get . sub)) . exec . exec

However, while evaluating the pfx code

push 8 . push 12 . ((push 1 . get . push 0 . get . sub)) . exec . exec we will figure out that the result will always be wrong because 8 will just be in the stack and never been use. In fact:

Step-by-Step Evaluation

Initial stack: ∅
Instruction sequence:

push 8 . push 12 . ((push 1 . get . push 0 . get . sub)) . exec . exec

1. push 8

Stack: 8

2. push 12

Stack: 12 :: 8

3. ((push 1 . get . push 0 . get . sub))

→ Push the **executable sequence** onto the stack
Stack: (push 1 . get . push 0 . get . sub) :: 12 :: 8

4. exec

→ Pop the executable sequence and prepend it to the instruction queue
New instruction sequence:

push 1 . get . push 0 . get . sub . exec

Stack: 12 :: 8

5. push 1

Stack: 1 :: 12 :: 8

6. get

Pop 1 → look up index 1 → result = 12
Stack: 12 :: 12 :: 8

7. push 0

Stack: 0 :: 12 :: 12 :: 8

8. get

Pop 0 → look up index 0 → result = 12
Stack: 12 :: 12 :: 12 :: 8 => Wrong!

Expected result is 4 = 12 − 8)

8 is too deep in the stack and never used!

Exercise 11:

Question 1:

let x = e1 in e2 ≡ App (Fun (x, e2), e1)

Question 2: see modifications in expr/fun in lexer.mll et parser.mly

Exercise 12:

Question 2:

App(App(Fun("x", Fun("y", Binop(Sub, Var("x"), Var("y")))), Const 12), Const 8)

<u>step1: applying to 12</u>

compile(App(Fun("x", Fun("y", Binop(Sub, Var("x"), Var("y")))), Const 12)

<u>step2: Apply inner function to 8</u>

x ↦ 12, y ↦ 8

<u>step3: Evaluate x - y</u>

x ↦ 12, y ↦ 8 → 12 − 8 = **4**

<u>final result:</u>

The evaluation of `(((λx.λy.(x - y)) 12) 8)` gives:**4**

Exercise 13:

Question1:

using the actual pfx, it is not possible. However, using a new modification will make this possible. In fact, The standard prefix language `Pfx` lacks the capability to dynamically capture and include the values of free variables at runtime — which is necessary for closures**.**

To represent closures in `Pfx`, we need to generate executable code (prefix notation) that includes the environment — the current bindings of free variables. This cannot be fully determined at compile time because the values of free variables are only known at runtime.

The `append` construct solves this:

- At runtime, we push the values of free variables onto the stack.
- Then we append these as `push` instructions to the front of the function body (the executable sequence).
- This gives a complete, ready-to-execute function (closure) that includes both:
  - The code.
  - The runtime values of the free variables.

Question2:

- n be an integer,
- Q be an executable sequence,
- Q' be the resulting sequence after appending the appropriate instruction.

Case 1: Integer value (push case)

$$n :: Q :: S$$

And `Q` is an executable sequence, then:

append . P , n :: Q :: S

→ P , (push n . Q) :: S

Case 2: Executable sequence value (code case):

If the stack is:

Q1 :: Q2 :: S

and both Q1 and Q2 are executable sequences, then:

append . P , Q1 :: Q2 :: S

→ P , (Q1 . Q2) :: S

Question4:

⟦e⟧env denote the translation of e in environment env, where each variable is mapped to a stack index

Constants:

⟦Const(n)⟧env = push n

Variables:

If env(x) = i:

⟦Var(x)⟧env = push i . get

Unary minus:

⟦Uminus(e)⟧env = ⟦e⟧env . uminus

Binary operations:

⟦Binop(op, e1, e2)⟧env = ⟦e1⟧env . ⟦e2⟧env . op

Function (Closure):

⟦Fun(x, body)⟧env =

(⟦body⟧env')   ; generate the core function

→ for each vi ∈ FV:

⟦vi⟧env . append ; build closure dynamically

⟦v1⟧env . append .

⟦v2⟧env . append .

...

(⟦body⟧env')

Application:

⟦App(e1, e2)⟧env = ⟦e2⟧env . ⟦e1⟧env . exec

Question6:

((λx.λy.(x − y)) 12) 8

((fun x ⇒ fun y ⇒ x − y) 12) 8

Final compiled Pfx code:

```
push (
   access 0 ;
   append ;
   push (
      access 1 ;
      access 0 ;
      sub
   )
) ;
push 12 ;
call ;
```

push 8 ;

call