COMPILATION PROJECT

QUESTIONS

Authors : KROUT Mohamed Amine, ROMDHANI Mohamed Mokhtar

Exercise 1 : (expl)

▷ **What is a stack? What are the operations that you usually execute on a stack?**

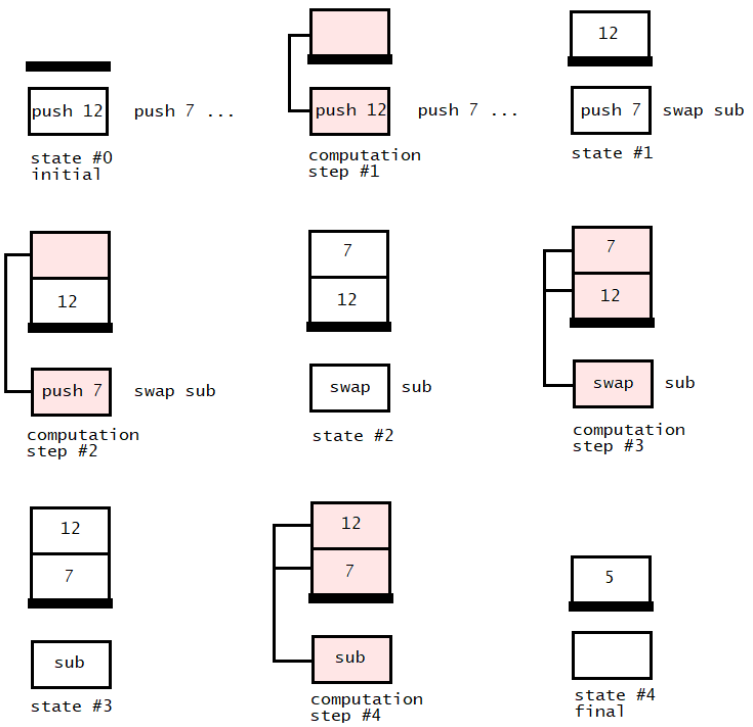
In computer science, a stack is an abstract data type that represents a collection of elements arranged in a linear order, with the ability to access and manipulate only the top element of the stack. It follows the **Last-In-First-Out (LIFO)** principle, where the last element added to the stack is the first one to be removed.

As we said, a stack is an abstract data type, so we need a way to store our data and access it. Thus, there are some basic operations we usually use to do so, such as :

- **Push :** adds an element to the top of the stack
- **Pop :** removes the top element from the stack
- **Peek :** returns the top element without removing it
- **IsEmpty :** checks if the stack is empty or not.
- **Clear :** removes all elements from the stack, making it Empty (multiple pops)
- **Size :** returns the number of elements currently in the stack

Exercice 2 : (expl)

▷ **Detail in the same way the execution of 0 push 12 push 7 swap sub.**



Exercice 3 :

▷ **Question 3.1 (expl):** ***Explain using plain words the semantics of programs.***

The semantics of programs specify how a computer program behaves and what it does when it is executed. It describes how the program instructions are executed step-by-step and how the program interacts with the input data and output results. The semantics define the rules and meanings of the program statements and expressions, as well as the behaviour of the program under different conditions, such as errors or exceptions. The program semantics also specify the values of variables and data structures used by the program and how they are updated during program execution. Overall, program semantics provide a formal and precise description of what a program does and how it behaves.

In the given examples we provided this explanations :

$$(1)\ \frac{i \neq n}{v_1, ..., v_n \vdash i, Q \Rightarrow \text{ERR}}$$

- Given a program i,Q which awaits i arguments, if we provide it with n arguments such as n is different from i (so we provided a wrong number of arguments) this should result in an error.

$$(2)\ \frac{Q, v_1 :: ... :: v_n :: \varnothing \rightarrow^* \text{ERR}}{v_1, ..., v_n \vdash n, Q \Rightarrow \text{ERR}}$$

- Here, suppose we provided the right number of arguments and a sequence of instructions. If one of the intermediate instructions of the transitive closure generates an error then the program should generate an error too.

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

- In this case, given a program provided with the right number of arguments, the correct set of instructions that generates no errors and that returns the stack, the semantics defines the function that returns the value on top of the stack. So, we're talking about the **Peek** or the **Pop** functions.

▷ **Question 3.2 (math):** ***A case is still missing, spot it out and give the corresponding rule.***

An important case that we should mention is what happens if we provide an empty stack ? should that work or we should throw an error ? . In most cases, when attempting to perform an operation on an empty stack, an error should be thrown, as it's not possible to retrieve or manipulate any data from an empty stack.

$$(4)\ \frac{Q, v_1 :: ... :: v_n :: \varnothing \rightarrow^* \varnothing, \varnothing}{v_1, ..., v_n \vdash n, Q \Rightarrow \text{ERR}}$$

▷ **Question 3.3 (math):** ***Give the rules describing the small step semantics for instruction sequences. Beware to cover all cases of runtime errors.***

- Skip rule : given an empty instruction, the program should skip to the next one

$$\frac{I=\varnothing}{I.Q,S \rightarrow Q,S}$$

- Arithmetic rules : given two numbers, the program should perform the right operation correctly.
    - Runtime error : to be triggered when one of the two operands does not exist and if the second operand is equal to zero for the DIV operation.

$$\frac{r = v1 + v2}{ADD.Q, v1 :: v2 :: S \rightarrow Q, r :: S} \qquad ADD.Q,\ v1 :: \emptyset \ \rightarrow\ ERR$$

$$\frac{r = v1 - v2}{SUB.Q, v1 :: v2 :: S \rightarrow Q, r :: S} \qquad SUB.Q,\ v1 :: \emptyset \ \rightarrow\ ERR$$

$$\frac{r = v1 \times v2}{MULT.Q, v1 :: v2 :: S \rightarrow Q, r :: S} \qquad MULT.Q,\ v1 :: \emptyset \ \rightarrow\ ERR$$

$$\frac{v2 \neq 0, r = v1\ /\ v2}{DIV.Q, v1 :: v2 :: S \rightarrow Q, r :: S} \qquad DIV.Q,\ v1 :: \emptyset \ \rightarrow\ ERR$$

- Push rule : given a number and a stack, the number should be pushed on the top of the stack
    - Runtime error : to be triggered when one of the two operands is not a number.

$$\frac{I = PUSH\ v1}{I.Q, S \rightarrow Q, v1 :: S}$$

- Pop rule : given a stack, the pop function should retrieve the top element and return the stack
    - Runtime error : to be triggered when the stack is empty

$$\frac{I = POP}{I.Q, \_ :: S \rightarrow Q, S} \qquad \frac{I = POP}{I.Q, \emptyset \rightarrow ERR}$$

- Swap rule : given a stack, it swaps the top element with the one just after it.
    - Runtime error : to be triggered if the stack has less than two elements.

$$\frac{I = SWAP}{I.Q, v1 :: v2 :: S \rightarrow Q, v2 :: v1 :: S} \qquad \frac{I = SWAP}{I.Q, v1 :: \emptyset \rightarrow ERR}$$

Exercice 5 :

▹ **Question 5.1 (math):** ***Propose a compilation schema of Expr in Pfx. Give its formal description. Notice that with the current definition of Pfx, we cannot implement variables. We defer their implementation to a later exercise.***

Given the definition of Pfx we have, we can propose a compilation scheme that covers constants, binary operations and Uminus of an expression. Let's call our function **generate** that matches each Expr with the corresponding Pfx. Then, we'll have the following descriptions :

- generate(Const n) : ça devrait faire un appel au PUSH n de Pfx → *[PUSH n]*

- generate(Binop (Badd, e1, e2)) : la fonction doit être récursive pour qu'on puisse traiter les sous-expressions → [*generate(e2) ; generate(e1) ; PLUS]*
- generate(Binop (Bsub, e1, e2)) → [*generate(e2) ; generate(e1) ; SUB]*
- generate(Binop (Bmul, e1, e2)) → [*generate(e2) ; generate(e1) ; MULT]*
- generate(Binop (Bdiv, e1, e2)) →[*generate(e2) ; generate(e1) ; DIV]*
- generate(Binop (Bmod, e1, e2)) → [*generate(e2) ; generate(e1) ; MOD]*
- generate(Uminus e) → [generate(e) ; generate(0) ; SUB]

Exercice 9 :

▷ **Question 9.1 (expl):** ***Do we need to change the rules for the already defined constructs ?***

As for now, we don't need to modify the existing constructs. All we need to do is to add the rules for interpreting an executable sequence and executing it using **exec** and also the **get** instruction to access elements in the stack.

▷ **Question 9.2 (math):** ***Give the formal semantics of these new constructions.***

Executable sequence (Q), where Q is a usual instruction sequence, when it is encountered the executable sequence is pushed on the top of the stack :

$$(Q).I.J,\ S\ \rightarrow\ I.J,\ (Q)::S$$

Exec which is an instruction that pops the top of the stack and executes it by appending it in front of the executing sequence, notice that the top of the stack must be an executable sequence :

$$\frac{I = exec}{I.J, (Q)::S \rightarrow Q.I.J, S} \qquad \frac{I = exec,}{I.J, v1::S \rightarrow ERR}$$

Get pops the integer i on top of the stack, and copies on top of the stack the i-th element of the stack, it raises an error if there is not enough element on the stack :

$$\frac{I = get, v1 = 3}{I.J, v1::v2::v3::S \rightarrow v3::v2::S} \qquad \frac{I = exec, \#S < v1}{I.J, v1::S \rightarrow ERR}$$

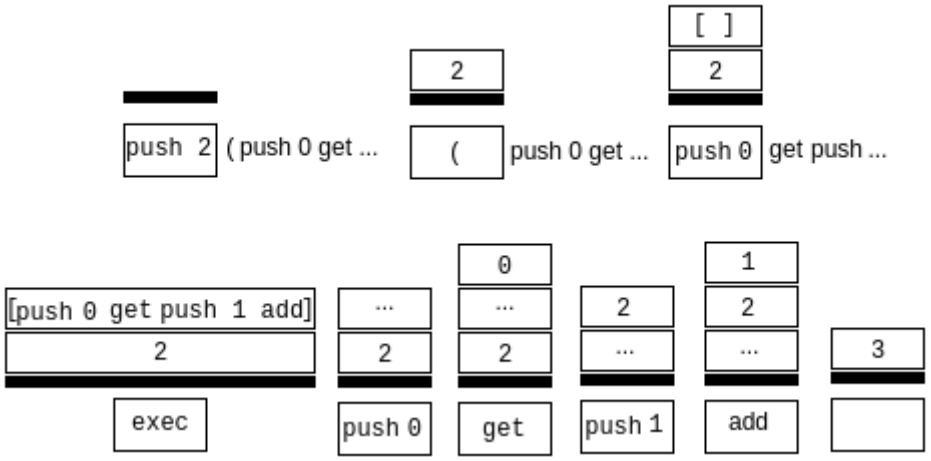
Exercice 10 :

▷ **Question 10.1 (expl):** ***Give the compiled version of the expression (λx.x + 1) 2. Then describe step by step the evaluation of its Pfx translation.***

The compiled version of the expression (λx.x + 1) 2 is the following : **push 2 seq_start push 0 get push 1 add seq_end exec.**

The first step is to push our argument into the stack, then we need to translate our lambda abstraction to an executable sequence, using the environment P = {x→0}, where x is at position 0 on the stack. Finally, all we need is to execute the executable sequence.

The evaluation of the expression presented below is as follows :



In the initial state, the stack is empty. First of all we push the parameter we'll use to execute the lambda function. So, in the second state we have **[2]** as a stack. The second step is to push the executable sequence on top of the stack. Later, we'll need to execute it as follows : we retrieve the element on the 0-th position of the stack which is the 2 we've already pushed, then we push 1 and execute the addition (add). The final state of the stack is the result of the lambda expression applied to our argument.
* In the picture above, the '(' is the seq_start and the ')' is the seq_end.

▹ **Question 10.2 (math):** ***Give the formal rule for transformation.***

Say we have a lambda expression called '**e**' with the form '**λx.t**' where **t** is an expression and **x** is a variable. The transformation is defined as follows : **e →Exec_Seq [push n; push [t]; exec]** with n is the position of the argument needed for the expression.

▹ **Question 10.4 (expl):** ***Give the compiled version of the expression ((λx.λy.(x − y)) 12) 8. Then describe step by step the evaluation of its Pfx translation. What do you think of the result? What is happening?***

The compiled version of the expression ((λx.λy.(x - y)) 12) 8 is the following : **push 12 push 8 seq_start push 0 get push 0 get swap sub seq_end exec**

The first step is to push our arguments into the stack (push 8 push 12), then we need to translate our lambda abstraction to an executable sequence. So, to do the subtraction we need to access our arguments using the get. Finally, all we need is to execute the executable sequence.

The problem is that in order to access our arguments, each time we need to push their position. For simple cases it may work, but for other complex cases, we'll need to keep track of them without the need of pushing their position.

Exercice 11 :

▹ **Question 11.1 (expl):** ***What is the translation in the syntax of Expr already defined of a new let x = e1 in e2 ?***

In the syntax of Expr, the **let x = e1 in e2** can be defined using the Lambda expressions. In fact, this expression allows us to define a new variable named **x,** bind its result to the expression **e1** and then use it in the second expression **e2**. This can be done in the lambda calculus as follows : **(λx.e2) e1**. This expression is supported by the fun/Expr already defined.

▷ **Question 11.2 (code):** ***What part of the code must be modified to get support for let? Give the modifications.***

To get support for let, we need to modify the **generate** function so it can match the let with the corresponding lambda structure and then transform it to be red by the pfx language.

Exercice 12 : (math)

▷ **Give the proof derivation computing the value of the term of question 10.4 (((λx.λy.(x−y)) 12) 8).**

To compute the value of the term of question 10.4, we can simply apply the lambda calculus properties :

(((λx.λy.(x−y)) 12) 8)
→ (λy.(12−y)) 8
→ 12 − 8
→ **4**

This is how we obtain the result 4.