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

UE – Langage et Logique

Formation Ingénieur sous Statut Étudiant, Année 2

Prof encadrant : J-C Bach

Compilation Project

Dahbani Mohammed
TAF DCL

Ezzakri Anas
TAF DCL

Année académique 2023-2024

Mars - Avril 2024

Table des matières

1 A simple stack language

1.1 Exercice 1

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

A stack is an abstract data type in computer science that serves as a collection of elements.

The order in which an element is added to or removed from a stack is described as last in, first out (LIFO). This means the most recently added element is the first one to be removed.

The main operations that you usually execute on a stack are (these operations occur at the top of the stack) :

1. **Push** : adds an element to the collection.

2. **Pop :** This operation removes the most recently added element (LIFO).

3. **Peek or Top** : This operation returns the value of the last element added without modifying the stack[1].

1.2 Exercice 2

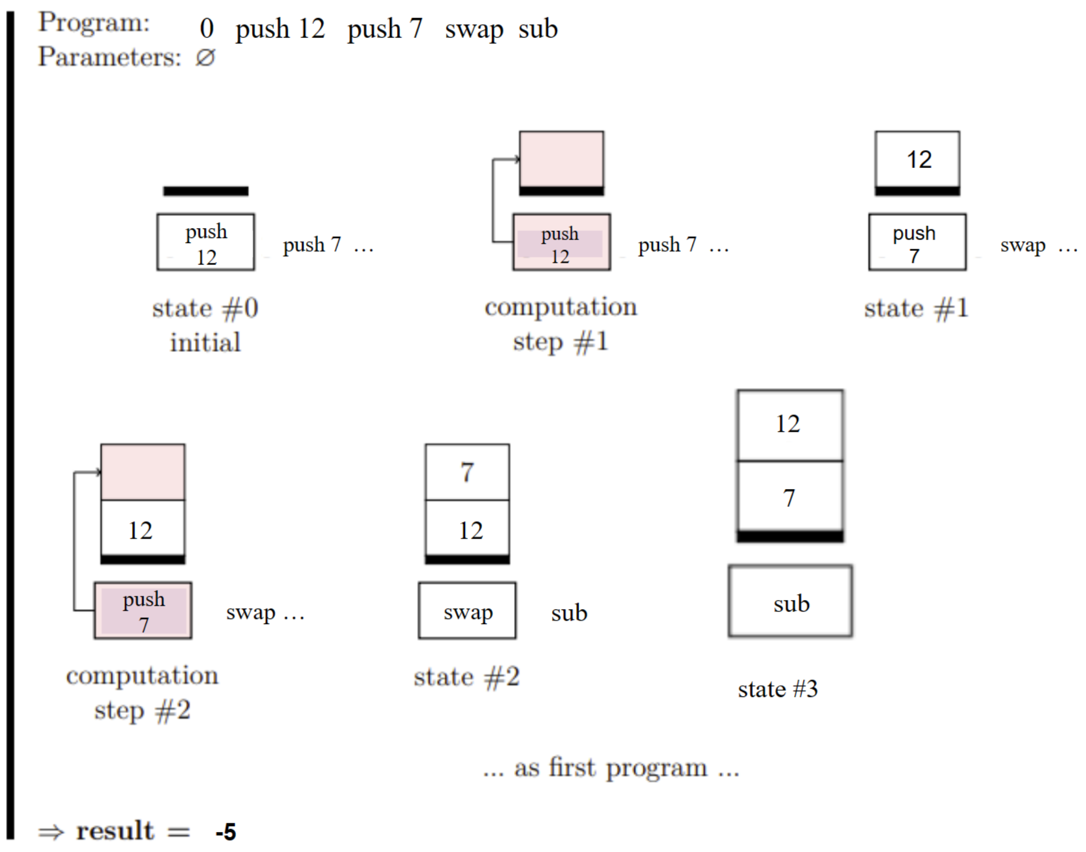


FIGURE 1 – SWOT analysis of Firefox project

1.3 Exercice 3

Question 3.1

1. If a program expects a certain number of arguments (i), but it's given a different number (n), the program throws an error (Err). This makes sense because the program wouldn't have the information it needs to run properly.

2. If the program reaches a point where the current state (Q) and the values on the stack can only lead to an error state (Err) no matter how many steps it executes (→), then the entire program results in an error. This captures situations where the program gets stuck in an infinite loop or encounters invalid operations.

3. This rule defines what happens when the program executes successfully. If the current state (Q) and the values on the stack all match up, and the program can reach a final state where there's nothing left on the stack, then the output of the program is the value on top of the stack (v).

Question 3.2

If the instruction sequence Q is not empty, represented as $Q = l_1.l_2 \ldots l_k$, and the stack contains the elements $v_1, \ldots, v_n$ followed by a 0 (indicating the bottom of the stack), then if the next instruction l_i is not a push operation, the system transitions to an error state because it cannot execute a pop or other stack operation without any elements on the stack. This rule ensures that the computation does not attempt to perform operations on an empty stack, which would be undefined behavior.

$$\frac{(Q = l_1.l_2....l_k, v_1 :: ... :: v_n :: 0 \to l_i....l_k, \emptyset)\hat{}(l_i! = \text{push})}{(v_1, ..., v_n, Q \Rightarrow \text{Err})}$$

Question 3.3

1. Push rule

$$\frac{}{n :: Q, S \to Q, n :: S}$$

2. Pop rule

$$\frac{v :: S! = \emptyset}{pop :: Q, v :: S \to Q, S}$$

3. Addition, Substraction, Multiplication, Swap Rules

$$Op \in \{Add, Sub, Mul, Swap\}$$

$$\frac{}{Op :: Q, v_1 :: v_2 :: S \to Q, (v_1 Op v_2) :: S}$$

4. Division rule

$$\frac{\begin{array}{c} Op \in \{Div, Rem\} \\ v_2! = 0 \end{array}}{Op :: Q, v_1 :: v_2 :: S \to Q, (v_1 Op v_2) :: S}$$

5. Empty sequence rule

$$\frac{}{\emptyset, S \to *\emptyset, S}$$

6. Error rule - EmptyStack

$$\frac{I \in \{Add, Sub, Mul, Swap, Div, Rem, Pop\}}{I :: Q, \emptyset \rightarrow^* Err}$$

7. Error rule - DivisionByZero

$$\frac{}{/ :: Q, v :: 0 :: S \rightarrow^* Err}$$

8. Invalid operation

$$\frac{}{Invalid :: Q, S \rightarrow^* Err}$$

1.4 Exercice 4

Question 4.1

Please take a look at the file "pfx/basic/ast.ml".

Question 4.2

Please take a look at the file "pfx/basic/eval.ml".

2 A simple arithmetic expression language

2.1 Exercice 5

Question 5.1

1. `Const n` compiles to `Push (Int n)`

   `Const n` → `Push (Int n )`

2. `Var x` is not supported in the current version of Pfx.

   `Var x` → Not Supported

3. `Binop (op, e1, e2)` compiles to `e1' ; e2' ; Operate op'` where `e1'` and `e2'` are the compilations of `e1` and `e2` respectively, and `op'` is the Pfx equivalent of the binary operator `op`.

   `Binop (op, e1, e2)` → `e1' ; e2' ; Operate op'`

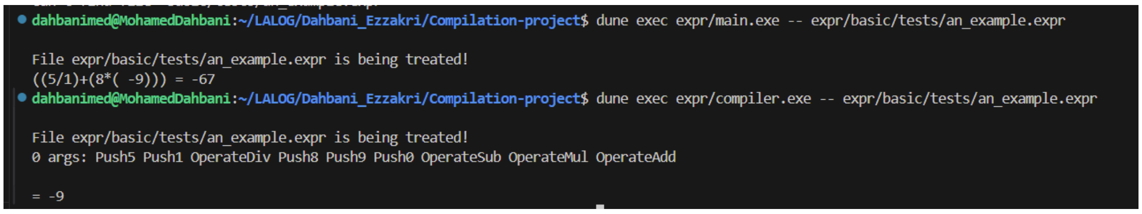
4. `Uminus e` compiles to `e' ; Push (Int 0) ; Operate Sub` where `e'` is the compilation of `e`.

   `Uminus e` → `e' ; Push (Int 0) ; Operate Sub`

Question 5.2

Please take a look at the file "expr/basic/toPfx.ml".

Successfully treated :



```
dahbanimed@MohamedDahbani:~/LALOG/Dahbani_Ezzakri/Compilation-project$ dune exec expr/main.exe -- expr/basic/tests/an_example.expr

File expr/basic/tests/an_example.expr is being treated!
((5/1)+(8*( -9))) = -67
dahbanimed@MohamedDahbani:~/LALOG/Dahbani_Ezzakri/Compilation-project$ dune exec expr/compiler.exe -- expr/basic/tests/an_example.expr

File expr/basic/tests/an_example.expr is being treated!
0 args: Push5 Push1 OperateDiv Push8 Push9 Push0 OperateSub OperateMul OperateAdd

= -9
```

Figure 2 – Test

3 Parsing

3.1 Exercice 6

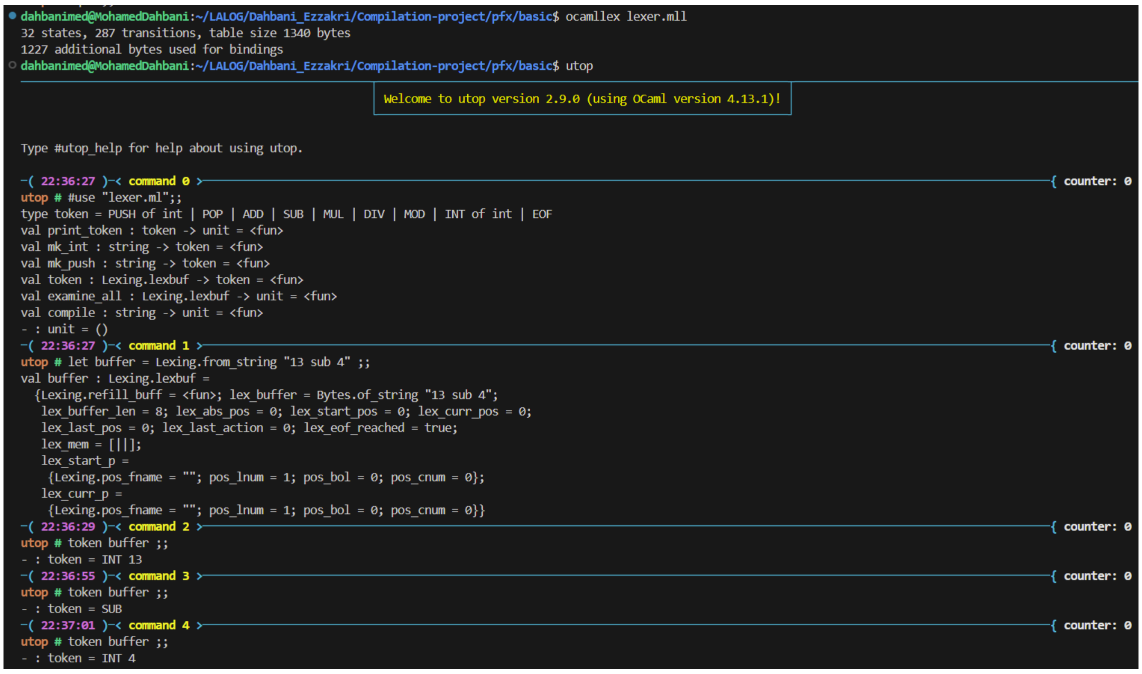
Question 6.1

Please take a look at the file "pfx/basic/lexer.mll".

Question 6.2

Please take a look at the file "pfx/basic/lexer.mll"

Successfully treated :



```
dahbanimed@MohamedDahbani:~/LALOG/Dahbani_Ezzakri/Compilation-project/pfx/basic$ ocamllex lexer.mll
32 states, 287 transitions, table size 1340 bytes
1227 additional bytes used for bindings
dahbanimed@MohamedDahbani:~/LALOG/Dahbani_Ezzakri/Compilation-project/pfx/basic$ utop

                              Welcome to utop version 2.9.0 (using OCaml version 4.13.1)!

Type #utop_help for help about using utop.

-( 22:36:27 )-< command 0 >----------------------------------------------{ counter: 0
utop # #use "lexer.ml";;
type token = PUSH of int | POP | ADD | SUB | MUL | DIV | MOD | INT of int | EOF
val print_token : token -> unit = <fun>
val mk_int : string -> token = <fun>
val mk_push : string -> token = <fun>
val token : Lexing.lexbuf -> token = <fun>
val examine_all : Lexing.lexbuf -> unit = <fun>
val compile : string -> unit = <fun>
- : unit = ()
-( 22:36:27 )-< command 1 >----------------------------------------------{ counter: 0
utop # let buffer = Lexing.from_string "13 sub 4" ;;
val buffer : Lexing.lexbuf =
  {Lexing.refill_buff = <fun>; lex_buffer = Bytes.of_string "13 sub 4";
   lex_buffer_len = 8; lex_abs_pos = 0; lex_start_pos = 0; lex_curr_pos = 0;
   lex_last_pos = 0; lex_last_action = 0; lex_eof_reached = true;
   lex_mem = [||];
   lex_start_p =
    {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 0};
   lex_curr_p =
    {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 0}}
-( 22:36:29 )-< command 2 >----------------------------------------------{ counter: 0
utop # token buffer ;;
- : token = INT 13
-( 22:36:55 )-< command 3 >----------------------------------------------{ counter: 0
utop # token buffer ;;
- : token = SUB
-( 22:37:01 )-< command 4 >----------------------------------------------{ counter: 0
utop # token buffer ;;
- : token = INT 4
```

Figure 3 – Test

3.2 Exercice 7

Please take a look at the file "pfx/basic/lexer.mll". **Successfully compiled !**

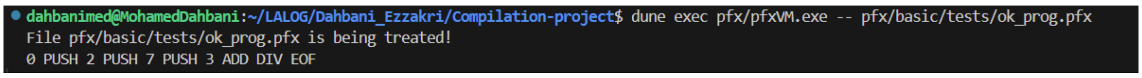
3.3 Exercice 8

Question 8.1

Please take a look at the file "pfx/basic/parser.mly"

Question 8.2

Successfully treated :



```
● dahbanimed@MohamedDahbani:~/LALOG/Dahbani_Ezzakri/Compilation-project$ dune exec pfx/pfxVM.exe -- pfx/basic/tests/ok_prog.pfx
  File pfx/basic/tests/ok_prog.pfx is being treated!
  0 PUSH 2 PUSH 7 PUSH 3 ADD DIV EOF
```

FIGURE 4 – Test

4 Simple functions

4.1 Exercice 9

Question 9.1

No, we do not need to change the rules for the already defined constructs. The new instructions Q, exec, and get are additions to the language and do not modify the behavior of existing constructs.

Question 9.2

The formal semantics of these new constructions could be defined as follows :

— Qexec : This is an executable sequence. When encountered, the sequence Q is pushed onto the top of the stack :

$$(Qexec)Q, S \rightarrow Q, Qexec :: S$$

— exec : This instruction pops the top of the stack and executes it by appending it in front of the executing sequence. If the top of the stack is an executable sequence Q and the remaining stack is S :

$$execQ, Seqs :: S \rightarrow s :: Q, S$$

— get : This instruction pops the integer i on top of the stack, and copies onto the top of the stack the i-th element of the stack. If the stack before the operation is

$$i :: S$$

and the i-th element of S is x, then the stack after the operation is

$$x :: i :: S$$

To study all cases :

— if the index cannot be obtained :

$$getQ, Seqv :: \emptyset \rightarrow Err$$

$$getQ, Intn :: \emptyset \rightarrow Err$$
$$getQ, \emptyset \rightarrow Err$$

— if the stack index cannot be reached :

$$getQ, Inti :: S, S.length < i \rightarrow Q, v2 :: v1 :: S$$

— if the stack index can be reached :

$$getQ, Inti :: S, S.length \geq i \rightarrow Q, S[i-1] :: S$$

Question 9.3

Yes, the lexer and parser of Pfx need to be extended to include these changes. For the lexer, we would add patterns to match the new instructions.

Please take a look at the file "expr/basic/fun/parser.mly" and "expr/basic/fun/lexer.mll"

4.2 Exercice 10

Question 10.1

The expression $(\lambda x.x + 1)2$ is a lambda calculus expression. When we compile this to Pfx, we first substitute 2 for x in the body of the function, and then compile the resulting expression. Here's how it works :

1. The argument command : `push 2`
2. The function command : `(push 1 push 2 get add)`
3. The application commands : `exec swap pop`, used to clean the variable after execution.

So, the compiled version of $(\lambda x.x + 1)2$ in Pfx is : `App(Fun(Var(x), Binop(Badd, Var(x), 1)), 2)`

Now, let's evaluate this Pfx expression step by step :

1. Push 2 onto the stack. The stack is now [2].
2. Executing the function command : `[2; (push 1 push 2 get add)]` where 2 represents the depth of the variable in the environment
3. The stack is now $[2, 1, 2]$.
4. Apply the + operation, which pops two values from the stack, adds them, and pushes the result back onto the stack. The stack is now $[3, 2]$.
5. After swapping : The stack is now $[2, 3]$.
6. After popping : The stack is now [3].

So, the result of evaluating the Pfx expression is 3, which is the same as the result of evaluating the original lambda calculus expression $(\lambda x.x + 1)2$.

Question 10.2

The formal rule for transformation in this context is as follows :

— For a lambda abstraction $(\lambda x.E)$, where E is the body of the function, the transformation rule is to compile E into an executable sequence that assumes its parameter is on the top of the stack when it starts to execute. Whenever it wants to use its parameter, it gets it from the stack as we did in the previous question using an environment P associating a variable to its position in the stack (its depth).

Question 10.3

Please take a look at the file "expr/basic/fun/toPfx.ml"

Question 10.4

The expression $((\lambda x.\lambda y.(x - y))\ 12)\ 8$ represents a function that takes two arguments x and y, subtracts y from x, and then applies this function to the arguments 12 and 8.

When we compile this to Pfx, we first substitute 12 for x and 8 for y in the body of the function, and then compile the resulting expression. So :

1. The compiled version translates to `App(Fun(Var(y), App(Fun(Var(x), Var(x) - Var(y)), 12)), 8)`.
2. The resulting Pfx translation is as follows : `push 8 push 12 ((push 1 get push 3 get sub)) exec swap pop exec swap pop`.

4.3 Exercice 11

Question 11.1

The expression `let x = e1 in e2` can be interpretedas $(\lambda x.e2)e1$, where $\lambda x.e2$ creates a function that takes x as an argument and computes $e2$. This function is then immediately applied to $e1$. In the `Expr` syntax, this transformation can be respresented as we saw in the previous section :
`App(Fun(Var(x), e2), e1)`.

Question 11.2

The term 'let' is a syntactic sugar (which mean according to our research a syntax within a programming language that is designed to make things easier to read or to express), so to resolve the problem, we just need to alter the token identification and connect it to the existing logic in the parser. This requires changes to be made in two files : fun/parser.mly and fun/lexer.mll.

Please take a look at the two files.

5 Closure

5.1 Exercice 12

The proof derivation computing the value of the term of question 10.4 $(((\lambda x.\lambda y.(x-y))12)8)$ is as follows :

1. Apply the function $\lambda x.\lambda y.(x-y)$ to 12, we get $\lambda y.(12-y)$: `App(Fun(Var(x), Var(x) - Var(y)), 12))`.
2. Then apply $\lambda y.(12-y)$ to 8, we get $12-8=4$. : App(Fun(Var(y), 12 - Var(y)), 8))

So, the value of the term $(((\lambda x.\lambda y.(x-y))12)8)$ is 4.

5.2 Exercice 13

(not completed)

Question 13.1

Question 13.2