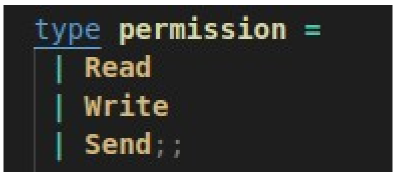
Language Based Technology For Security
AA 2021-22
Homework 1
Marc Hofmann
Matricola: 644820

1. The definition of the programming language along with some examples.

Permissions:

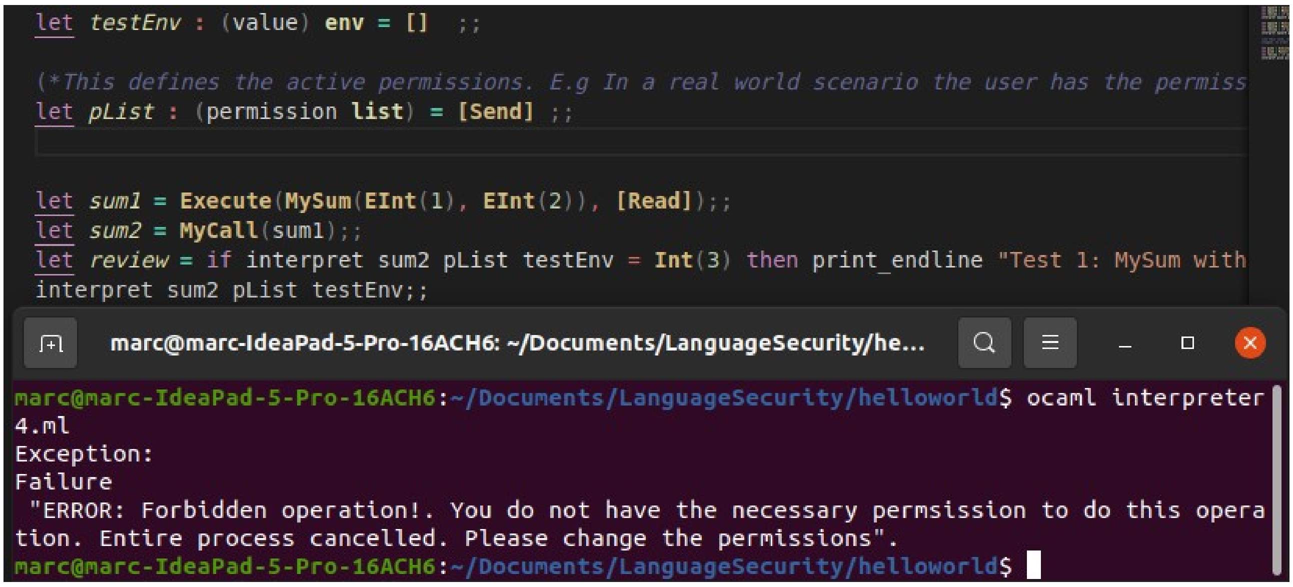


```
type permission =
  | Read
  | Write
  | Send;;
```

Figure 1: Permissions

I defined three types of permissions. *Read*, *Write* and *Send*. Due to the simplicity of the new language and interpreter, these permissions are not hierarchical, e.g. *Send* is not a higher permission than *Write*, and *Write* is not higher than *Read*. They are therefore permissions that are not "connected" to each other, but are instead completely independent.

In other words: Having the *Send* permission, does for example not mean that a expression with required permission *Read* can be used. The program will give an Error message, see figure 2:



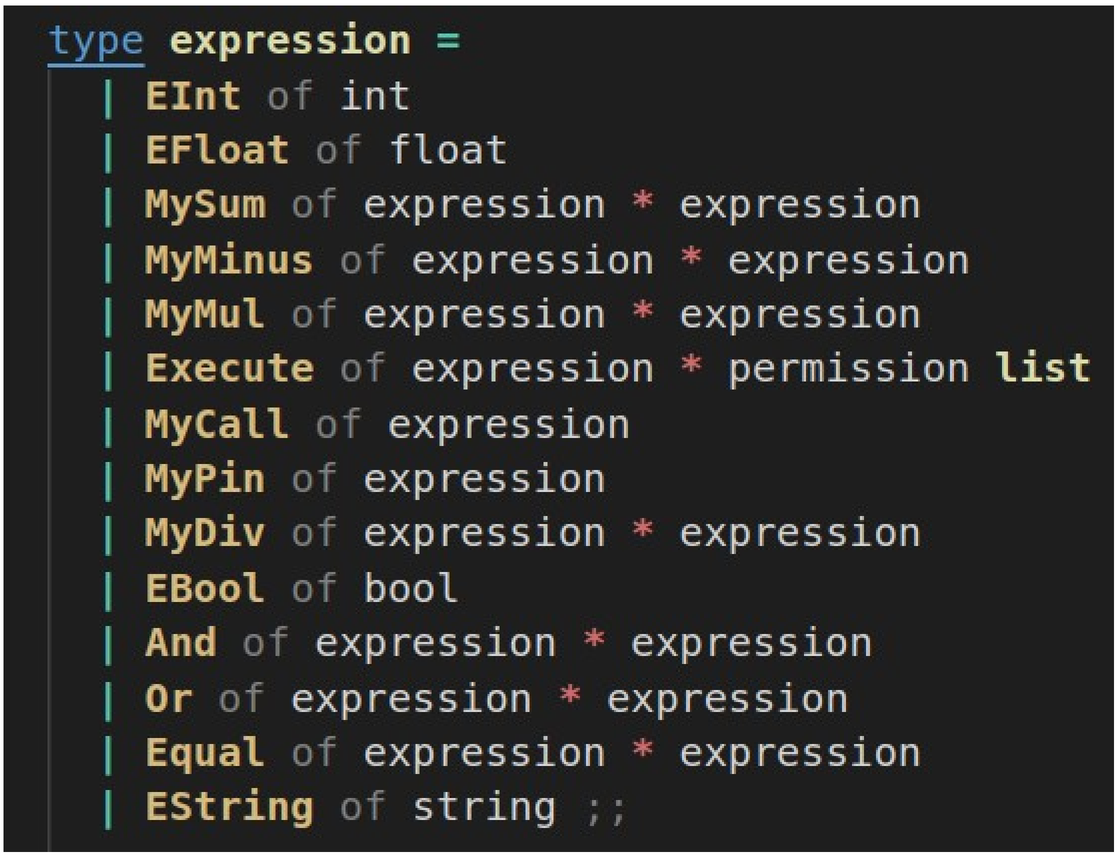
```
let testEnv : (value) env = []  ;;

(*This defines the active permissions. E.g In a real world scenario the user has the permiss
let pList : (permission list) = [Send] ;;


let sum1 = Execute(MySum(EInt(1), EInt(2)), [Read]);;
let sum2 = MyCall(sum1);;
let review = if interpret sum2 pList testEnv = Int(3) then print_endline "Test 1: MySum with
interpret sum2 pList testEnv;;
```

```
marc@marc-IdeaPad-5-Pro-16ACH6: ~/Documents/LanguageSecurity/he...
marc@marc-IdeaPad-5-Pro-16ACH6:~/Documents/LanguageSecurity/helloworld$ ocaml interpreter
4.ml
Exception:
Failure
 "ERROR: Forbidden operation!. You do not have the necessary permsission to do this opera
tion. Entire process cancelled. Please change the permissions".
marc@marc-IdeaPad-5-Pro-16ACH6:~/Documents/LanguageSecurity/helloworld$
```

Figure 2: Example on when permissions do not work

Expressions:

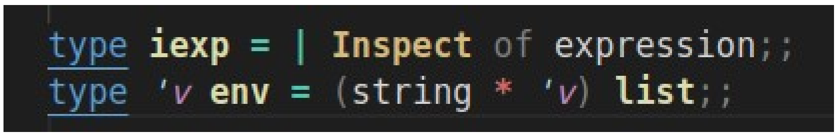
I have created several new expressions, that are necessary to fulfill the functional requirements. I will give examples in figure 3.



```
type expression =
    | EInt of int
    | EFloat of float
    | MySum of expression * expression
    | MyMinus of expression * expression
    | MyMul of expression * expression
    | Execute of expression * permission list
    | MyCall of expression
    | MyPin of expression
    | MyDiv of expression * expression
    | EBool of bool
    | And of expression * expression
    | Or of expression * expression
    | Equal of expression * expression
    | EString of string ;;
```

Figure 3: Defined expressions

In addition to the above mentioned examples, the programming language also defines supportive expressions like:

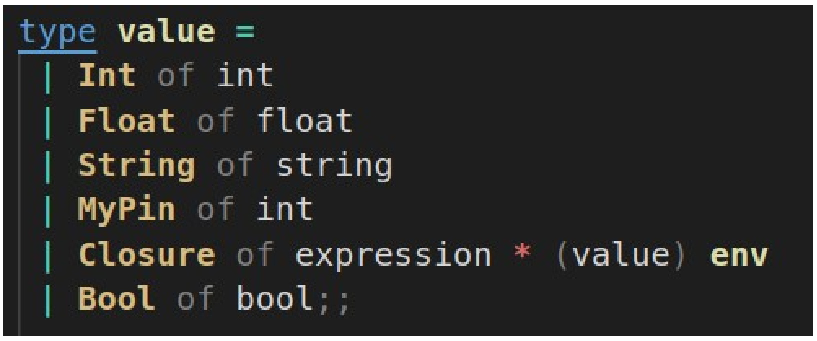


```
type iexp = | Inspect of expression;;
type 'v env = (string * 'v) list;;
```

Figure 4: Supportive Functions

iexp is for example used to perform stack inspection, it is later explained in which context.

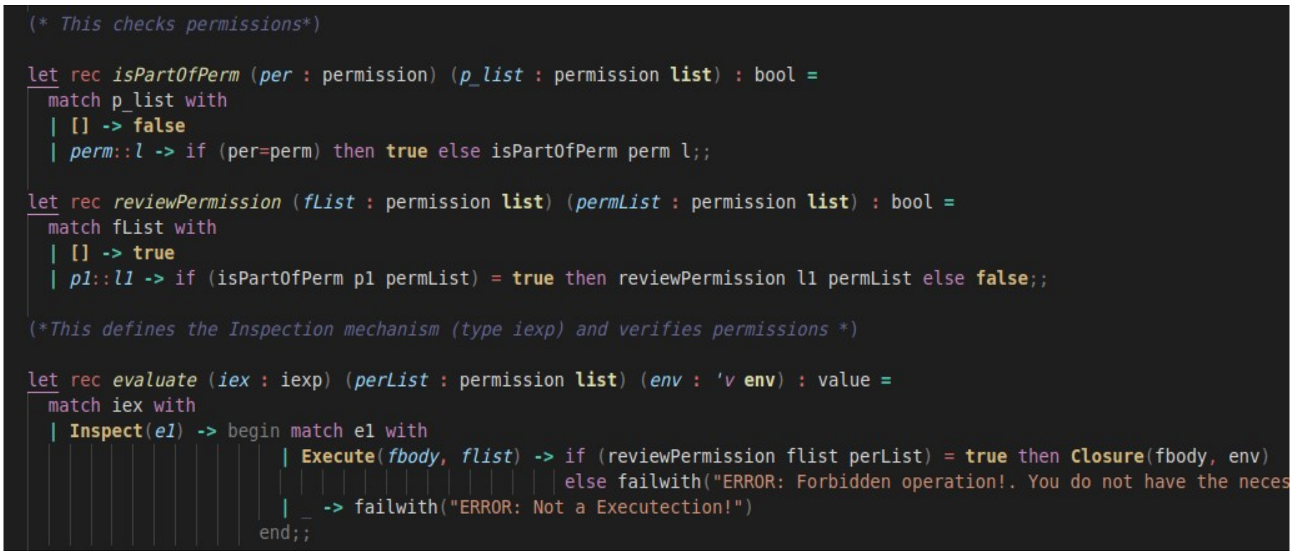
I also defined types, among them a closure.



```
type value =
  | Int of int
  | Float of float
  | String of string
  | MyPin of int
  | Closure of expression * (value) env
  | Bool of bool;;
```

Figure 5: Language Types

Permission Evaluation



```
(* This checks permissions*)

let rec isPartOfPerm (per : permission) (p_list : permission list) : bool =
  match p_list with
  | [] -> false
  | perm::l -> if (per=perm) then true else isPartOfPerm perm l;;

let rec reviewPermission (fList : permission list) (permList : permission list) : bool =
  match fList with
  | [] -> true
  | p1::l1 -> if (isPartOfPerm p1 permList) = true then reviewPermission l1 permList else false;;

(*This defines the Inspection mechanism (type iexp) and verifies permissions *)

let rec evaluate (iex : iexp) (perList : permission list) (env : 'v env) : value =
  match iex with
  | Inspect(e1) -> begin match e1 with
                         | Execute(fbody, flist) -> if (reviewPermission flist perList) = true then Closure(fbody, env)
                                                    else failwith("ERROR: Forbidden operation!. You do not have the neces
                         | _ -> failwith("ERROR: Not a Executection!")
                       end;;
```

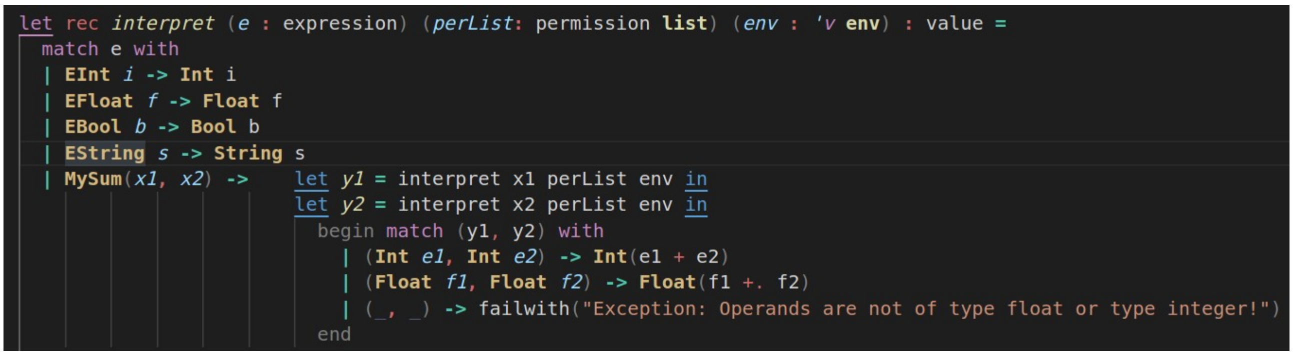
Figure 6: Permission mechanism

These 3 let expressions support in evaluating the permissions of user and operation / expression. The first one just checks if a certain permission is part of the permission list, while the second one evaluates if the permission of a expression is in line with the permission a user has. See in the tests later. The third *let rec evaluate* serves already as a partial interpreter. In this case the previously defined *Inspect* (see figure 4) gets implemented as part of Stack Inspection. This is later used as part of the *Execute(body, plist)* expression to overall determine whether the permission allows for execution of an operation.

Therefore, these 3 let expressions are closely linked together and serve as the mechanism to evaluate the permissions.

Interpreter

The last important part is the actual interpreter. The interpreter uses an expression e, a permission list and the environment. Figure 7 shows a partial, selected list:

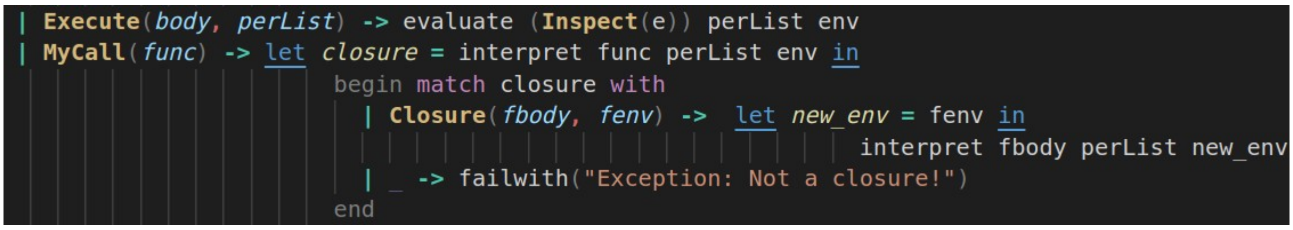


```
let rec interpret (e : expression) (perList: permission list) (env : 'v env) : value =
  match e with
  | EInt i -> Int i
  | EFloat f -> Float f
  | EBool b -> Bool b
  | EString s -> String s
  | MySum(x1, x2) ->    let y1 = interpret x1 perList env in
                        let y2 = interpret x2 perList env in
                          begin match (y1, y2) with
                            | (Int e1, Int e2) -> Int(e1 + e2)
                            | (Float f1, Float f2) -> Float(f1 +. f2)
                            | (_, _) -> failwith("Exception: Operands are not of type float or type integer!")
                          end
```

Figure 7: Partial Interpreter, with focus on MySum

The interesting part in figure 7 is the *MySum*. This expression at first stores the values *y1* and *y2* and uses them later in the matching.

Here I match patterns, in order to define how inputs should be handled. In all of the defined cases that include some type of number, float and integers can be used to allow for maximum flexibility for the users.



```
| Execute(body, perList) -> evaluate (Inspect(e)) perList env
| MyCall(func) -> let closure = interpret func perList env in
                  begin match closure with
                    | Closure(fbody, fenv) ->  let new_env = fenv in
                                                 interpret fbody perList new_env
                    | _ -> failwith("Exception: Not a closure!")
                  end
```

Figure 8: Execute and MyCall

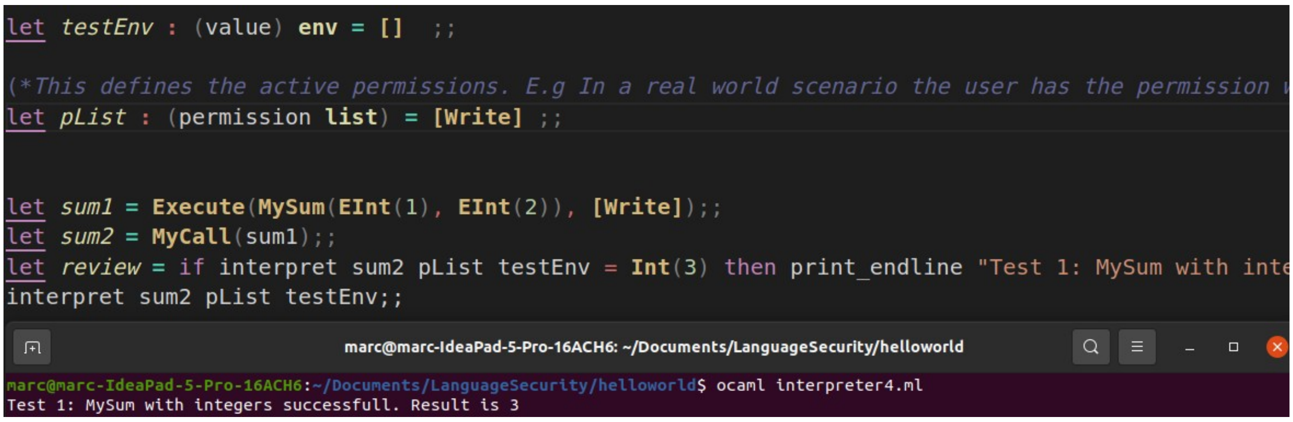
Two other relevant expressions exist. *Execute* and *MyCall*. *Execute* takes a *body* (in this case another expression) and the *perList* as arguments. This is then further used in the above described *evaluate* expression.

MyCall is simply an expression used to call the *Execute* expression and its arguments. Without *MyCall*, the program would not give any results. For this a *closure* is used, if the verification is positive.

Additional, but not worth mentioning, expressions were implemented and can be found in the source code.

<u>Testing</u>
For this program, it makes sense to evaluate tests using the *ocaml interpreter.ml* command in the terminal, as this will confirm or deny the success of the tests via command line. If one wants to look at the actual results, I currently recommend evaluating them using try.ocamlpro.com. This happens, because this is a very early version of the programming language and therefore has usability flaws like described.

I created some simple, manual tests. I put a focus on the testing in this report, as testing is the quickest and best way to evaluate the correct functioning of the project. In the following is a short description of the tests: A simple operation is conducted, and the results are compared to the expected – previously determined – results. An example is shown in figure 9.



```
let testEnv : (value) env = []  ;;

(*This defines the active permissions. E.g In a real world scenario the user has the permission
let pList : (permission list) = [Write] ;;


let sum1 = Execute(MySum(EInt(1), EInt(2)), [Write]);;
let sum2 = MyCall(sum1);;
let review = if interpret sum2 pList testEnv = Int(3) then print_endline "Test 1: MySum with inte
interpret sum2 pList testEnv;;
```

```
marc@marc-IdeaPad-5-Pro-16ACH6: ~/Documents/LanguageSecurity/helloworld
marc@marc-IdeaPad-5-Pro-16ACH6:~/Documents/LanguageSecurity/helloworld$ ocaml interpreter4.ml
Test 1: MySum with integers successfull. Result is 3
```

Figure 9: Testing example

Figure 9 shows testing of the expression *Execute(MySum.(..).*

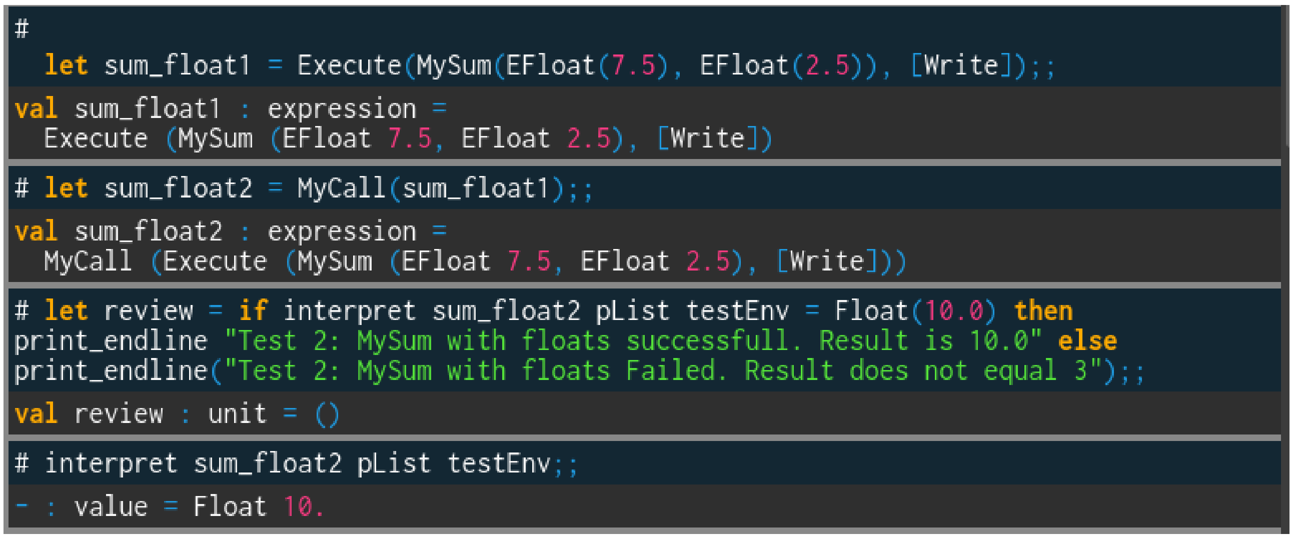
The result of 1 + 2 is 3. *Execute(MySum(...)* calculates this successful and compares it to the expected value in *let review*, therefore the test is passed.

Permissions can be tested by changing *let pList : …..* and by changing the required permissions of the expressions, e.g. *let sum1 = Execute(MySum(….)).*

The *pList* serves as a description of which permissions the user has.

Additional tests can be found for all expressions, also for float values in the source code.

try.ocamlpro.com prints values directly to the screen, which is in the current implementation not possible in the terminal. Example:



```
#
  let sum_float1 = Execute(MySum(EFloat(7.5), EFloat(2.5)), [Write]);;
val sum_float1 : expression =
  Execute (MySum (EFloat 7.5, EFloat 2.5), [Write])
# let sum_float2 = MyCall(sum_float1);;
val sum_float2 : expression =
  MyCall (Execute (MySum (EFloat 7.5, EFloat 2.5), [Write]))
# let review = if interpret sum_float2 pList testEnv = Float(10.0) then
print_endline "Test 2: MySum with floats successfull. Result is 10.0" else
print_endline("Test 2: MySum with floats Failed. Result does not equal 3");;
val review : unit = ()
# interpret sum_float2 pList testEnv;;
- : value = Float 10.
```

Figure 10: Testing in ocamlpro

Sadly, due to time constraints I did not have enough time to implement the Property based testing. If my understanding of it is correct though, it could be applied to the above manual testing examples, by using loops of the newly designed interpreter and comparing it with calculation of loops by the regular ocaml interpreter.

Some pseudo code:

```
Counter x = x ++
Counter y = y ++
if MySum(EInt(x) , EInt(y)) is equal to Ocaml Int(x + y) then test is correct.
```

2. <u>The source code of the trusted execution environment of the language.</u>

The source code is in a separate .ml file. It can be opened using try.ocamlpro.com or an ide, or even Libre Office.

3. A short description of the design choices.

I am not really sure what to write here. I think most of it was described above, unless I am misunderstanding.

In general I tried to keep everything simple and easy. Mainly due to my lack of previous Ocaml experience, this was quite hard and required lots of research in the internet, provided slides and previous examples during class and also the guides on the Ocaml website and other websites.

Obviously this is a very simple interpreter. There could be made additional implementations, like hierarchical permissions or also removing the *MyCall* expression and using only *Execute* instead.