Lab 9
Functional languages: compilation for an abstract machine

Objective

Understand the semantics of a toy functional programing language, and:

- discover an implementation of the compilation and execution of the abstract machine seen in the course.
- extend it.

First, download the archive.

9.1 Compiling Fun programs for an abstract machine

Discovering the toplevel In this week's archive, there is a toplevel for the compiler and virtual machine seen during the lectures. Hit `make` to compile it, it should generate a `toplevel` binary. When you run it, you get a toplevel where you can ask to compile expressions, run bytecode, or compile and run some expressions. The documentation of the options is available with `./toplevel --help`. You can find examples of valid source in the repository `tests/` of the archive, here is one [1]:

```
$ ledit ./toplevel
> c 2 + 2
Cst 2; Cst 2; Add
> cr 2 + 2
4
> cr 17 + (3 + 42)
62
```

Here is the abstract syntax of our (input) language:

```
(* Syntax of the input language of terms. *)
type expr =
  (* Arithmetic fragment *)
  | Constant of int
  | Addition of expr * expr

  (* Let fragment *)
  | Variable of variable
  | Let of variable * expr * expr (* [Let (x, a, b)] stands for [let x = a in b] *)

  (* Functional fragment *)
  | Function of variable * expr
  | Application of expr * expr
```

EXERCISE #1 ▶ **Abstract machine for Fun**

The objective is is to complete the given code in order to be able to compile and execute the whole language:

1. Read the files `main.ml` `syntax.ml` and `tP.ml` in order to become familiar with the syntax of the language and the machine code, and the structure of the code. The compilation into the abstract machine and its evaluation are done in `tP.ml`.

[1] `ledit` is a one-line editor written in OCaml. It provides line editing for the Caml toplevels, as well as other interactive Unix commands. On the ENS machines you can find it in `/usr/bin/`.

2. For local definitions, here is the expected output:

```
> c (let x = 1 in x+2)
Cst 1; Let "x"; Access "x"; Cst 2; Add; EndLet
> cr (let x = 1 in x+2)
3
```

Complete the following table that mimics the execution of the compiled program:

Code	Env.	Stack	comment
Cst 1`;Let "x";...EndLet`	∅	∅	push the constant on the stack
Let "x"`;...EndLet`	∅	`Vint(1)`	
Access "x"`;...EndLet`			
Cte 2`;Add;EndLet`			
Add`;EndLet`			
EndLet			
(end)			

Now complete the `TP.compile` and `TP.transition` functions.

3. (This is the hardest question of the lab) First, implement the compilation of user-defined functions (declaration, application):

```
> c ((fun x -> x+1) 42)
Cst 42; Closure("x", Access "x"; Cst 1; Add; Ret); App
> c ((let y=2 in fun x -> x+y)) 40
Cst 40; Cst 2; Let "y"; Closure("x", Access "x"; Access "y"; Add;
Ret); EndLet; App
```

The compilation of a given function thus only produces an element `Closure`($var, code$).

Then, the objective is to implement the execution steps, so that to obtain:

```
> cr ((fun x -> x+1) 41)
42
> cr ((let y=2 in fun x -> x+y)) 40
42
```

For the execution, you have to use the `VClosure of string * code * environment` value type for functions, that links together the argument of the function, its code and the environment *at its definition point.* A bit of the execution scheme for functions is depicted in Figure 9.1.

Code	Env.	Stack	Code	Env.	Stack	Comment
`Closure(x,c')`;c	σ	s	c	σ	$(x,c')[\sigma].s$	a new closure (x,c') on top of stack
`App`;c	σ	$(x,c')[\sigma'].v.s$	c'	$(x,v).\sigma'$	$c.\sigma.s$	push the code and initial env
`Ret`;c	σ	$v.c'.\sigma'.s'$	c'	σ'	$v.s$	stack top = returned value

Figure 9.1: Execution scheme for user-defined functions

First do the two following examples by hand:

Code	Env.	Stack
Cst 41;`Closure("x",Acces x;Cst 1; Add; Ret);App)`	$\emptyset$	$\emptyset$
Closure("x",Acces x;Cst 1; Add; Ret);`App`	$\emptyset$	`Vint(41)`

Code	Env.	Stack
Cst 40;`Cst 2; Let "y"; Closure("x",Acces x;Access y; Add; Ret);EndLet; App` (*Let code be* `Acces x;Access y; Add; Ret`)	$\emptyset$	$\emptyset$
Closure("x",code);`EndLet; App` (where $fv(code) = \{x,y\}$)	$y \mapsto 2$	`VInt40`

Now, implement:

- To implement the closure $(x, c')[\sigma]$ that is pushed on the stack you have to produce an element of type `VClosure of string * code * environment`. The environment which is pushed is the current value of all free variables of the code, that you can compute using the following Ocaml function:

  ```
  let rec fv = function
    | Access s :: rest -> s :: fv rest
    | Closure (_, code) :: rest ->
      fv code @ fv rest
    | _ :: rest -> fv rest
    | [] -> []
  ```

  You only have to filter out the current environnement values to only get track of these free variable values.

- To implement the `App` part, you have to recover the VClosure and the argument on top of stack, localy bind the argument to its value in the environment. **Use the following trick to push the remaining code c and the initial environnement σ on the stack: push a `VClosure("_",`c, σ`)` (an anonymous special closure).**

- For `Ret`, there should be a value followed by the special closure on top of stack. You only have to recover the initial environment and code, that are stored in this special closure.

4. Manually compile the expression `print 2` (where `print` is to be considered as a variable name)

5. The `VBuiltin` value type is used to represent concrete operations (like `print`). Explain how the transition for the `App` instruction when the function is not a `VClosure` but a `VBuiltin` will be implemented.

   An example of such builtins is `print` defined in `TP.builtins`, which is an initial environment the machine is loaded with when running code (See `main.ml` around line 80).

   Implement the transition for the instruction `App` when there is a `VBuiltin` on the stack (instead of a `VClosure`). Check that this works on the compiled form of `print 2` :

   ```
   > r Cst 2; Access "print"; App
   2
   0
   ```

6. Test your code against the examples in `tests/` and write your own ones:

   ```
   ./toplevel <tests/arith.test
   ```

7. Introduce a mistake in your compiler, by adopting the following wrong transition rule for the `App` machine instruction:

Code	Environment	Stack	Code	Environment	Stack
App;c	σ	$(x, c')[\sigma'].v.s$	c'	$(x, v).\sigma$	$c.\sigma.s$

   The mistake is in the Environment after the transition, right?

   Invent a Fun program that, when run on this erroneous machine, yields an unexpected result, thus showing the bug.

8. Add references to the language by introducing a new kind of (stack) value: `VRef of value ref`. Add three builtins operations to the `builtins` list, namely `ref`, `get` and `set` that mimick the ML operations `ref`, `!` and `:=`.

9.2 Recursion

EXERCISE #2 ► **Adding recursive definitions.**

We want here to extend our compiler to handle the definition of recursive functions of the following form:
`let rec f = fun x -> e1 in e2`
where `f` can be used in `e1` (and in `e2`, of course).

1. Suppose that we compile `letrec` like `let` (and hence a recursive function like a plain function).

   What will be the problematic transition in the abstract machine if we do so? (You can explain this on an example)

2. *(More difficult)* Extend the definitions in files `syntax.ml` and `tP.ml` to handle recursive definitions of the form given above.

   For this, a solution is to have a new kind of value. In absence of recursion, values are either integers or closures of the form $(x, c)[\sigma]$ (see definition of type `value` in the file `syntax.ml`). To handle recursion, we also have values of the form $Rec(f)(x, c)[\sigma]$ (a closure to represent recursive function f).

   The transition rule for instruction `Access`(x) is changed: instead of just looking up in the environment σ, and returning $\sigma(x)$:

   - we return v in case $\sigma(x) = v$ and v is either an integer or a plain closure (as before);
   - if on the other hand $\sigma(x) = Rec(f)(x, c)[\sigma]$, we return a closure $(x, c)[\sigma']$, where σ' is defined according to the behaviour we expect for a recursive definition.

   (a) Explain the run steps on the following example (that does not terminate):

   ```
   let rec pr = fun n-> print n+pr (n+1) in pr 0
   ```

   (b) Describe the behaviour of `Access`(x) in the case where $\sigma(x) = Rec(f)(x, c)[\sigma]$ (this amounts to defining σ' as introduced above).

   (c) Accordingly, define the compilation of a recursive definition, in which a value of the new form is added to the environment.

   (d) Implement, test!

   (e) (hard) Is it possible to get rid of those `Rec(f)(x, c)`$[\sigma]$ values? (Only to keep regular closure).