Lab 10
Rage Against the Abstract Machine

Objective

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

- play with the virtual machine presented during the lecture, and extend it;
- implement the compilation scheme from the toy language to this machine.

First, download the archive.

10.1 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 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 `syntax.ml` and `tP.ml` in order to become familiar with the syntax of the language and the machine code. The compilation into the abstract machine and its evaluation are done in `tP.ml`.

2. Compile manually the expression `print 2` (where `print` is to be considered as a variable name)

[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 here : `/home/scastell/.opam/system/bin/ledit`

3. 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).

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

5. Extend files `tP.ml` in order to be able to compile the whole Fun language, as seen at the lecture.

   While doing so, make sure that you understand why every transition rule of the machine is as it is. For each element of the language (one after the other) you have to: (a) write a compilation rule; (b) write an execution rule; (c) write a new example to test it.

6. 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.

7. 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 `:=`.

10.2 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, you might want to interact with the TA).* 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) 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).

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

   (c) Implement, test!

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