yz709-compiler-sup3

Question 2 (CPS and defunctionalisation exercise sheet)
- Problem 1
- Problem 2
- Problem 3
- Problem 4

Question 3
Question 4
Question 5(y2017p3q4)
Question 6
Question 1
- Notes for the paper

Question 2 (CPS and defunctionalisation exercise sheet)

Problem 1

Again by hand, eliminate tail recursion from fold_left. Does your source-to-source transformation change the type of the function? If so, can you rewrite your code so that the type does not change?

```
(* tail-recursive version: no deferred operation *)
(* fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a *)
let rec fold_left f accu l =
  match l with
      [] -> accu
  | a::l -> fold_left f (f accu a) l

(* sum up a list *)
let sum1 = fold_left (+) 0 [1;2;3;4;5;6;7;8;9;10] (* 55 *)
let product1 = fold_left (fun x y -> x * y) 1 [1;2;3;4] (* 24 *)
let concate1 = fold_left (fun x y -> y :: x) [9;8;7;6] [1;2;3;4]
(* [4; 3; 2; 1; 9; 8; 7; 6] *)
```

- No it has the same type `val fold_left_iterative : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>`

```
(* eliminate tail recursion *)
let fold_left_iterative f accu l =
```

```
    let accu_ref = ref accu in
    let r = ref l in
    let _ = while (!r) != [] do
        match !r with
        |[] -> r := [] (* this case will never been seen *)
        |x::xs -> r := xs; accu_ref := (f (!accu_ref) x);
    done
  in !accu_ref

let sum2 = fold_left_iterative (+) 0 [1;2;3;4;5;6;7;8;9;10] (* 55 *)
let product2 = fold_left_iterative (fun x y -> x * y) 1 [1;2;3;4] (* 24 *)
let concate2 = fold_left_iterative (fun x y -> y :: x) [9;8;7;6] [1;2;3;4]
(* [4; 3; 2; 1; 9; 8; 7; 6] *)
```

💡 Comments:

Problem 2

Apply by hand the CPS transformation to the gcd code. Explain your results.

```
let rec gcd(m, n) =
    if m = n
    then m
    else if m < n
         then gcd(m, n - m)
         else gcd(m - n, n)

let gcd_test_1 = List.map gcd [(24, 638); (17, 289); (31, 1889)]
(* int list = [2; 17; 1] *)
```

- We can eliminate tail recursion in the `gcd` function easily, because at each stage, we only recursively call the function `gcd` once

  ```
  (* eliminate tail recursion *)
  let gcd_iter(m, n) =
      let m_ref = ref m in
      let n_ref = ref n in
      let _ = while (!m_ref) != (!n_ref) do
          if (!m_ref) < (!n_ref) then n_ref := (!n_ref) - (!m_ref)
          else m_ref := (!m_ref) - (!n_ref)
          done
      in !m_ref

  let gcd_test_3 = List.map gcd_iter [(24, 638); (17, 289); (31, 1889)]
  (* val gcd_test_3 : int list = [2; 17; 1] *)
  ```

- For a cps transformation, `cnt` is the continuation/function, in `gcd_cps_util(m, n, cnt)`, it expects the result of `gcd(m, n)` as its argument, similarly, `fun a -> cnt a` is the continuation waiting for `gcd(m, n - m)`, and `fun b -> cnt b` is the continuation waiting for `gcd(m - n, n)`.
- Since we output `m` when arrived at the base case `m = n`, so the continuation `cnt` is the identity function `fun x -> x`.

```
let rec gcd_cps_util(m, n, cnt) =
  if m = n then cnt m
  else if m < n
    then gcd_cps_util(m, n - m, fun a -> cnt a)
    else gcd_cps_util(m - n, n, fun b -> cnt b)
let rec gcd_cps(m, n) = gcd_cps_util(m, n, fun x -> x)

let gcd_test_2 = List.map gcd_cps [(24, 638); (17, 289); (31, 1889)]
(* int list = [2; 17; 1] *)
```

💡 Comments:

Problem 3

Environments are treated as functions in `interp_0.ml`. Can you transform these definitions, starting with defunctionalisation, and arrive at a list-based implementation of environments?

```
(* update : ('a -> 'b) * ('a * 'b) -> 'a -> 'b *)
let update(env, (x, v)) = fun y -> if x = y then v else env y

(* mupdate : ('a -> 'b) * ('a * 'b) list -> 'a -> 'b *)
let rec mupdate(env, bl) =
    match bl with
    | [] -> env
    | (x, v) :: rest -> mupdate(update(env, (x, v)), rest)

(* env_empty : string -> 'a *)
let env_empty = fun y -> failwith (y ^ " is not defined!\n")

(* env_init : (string * 'a) list -> string -> 'a *)
let env_init bl = mupdate(env_empty, bl)
```

- The function `mupdate` is a recursive function, so we have to use CPS and DFC to turn it into a function carrying an explicit stack.

```
(* CPS of recursive function `mupdate` *)
(* val mupdate_cps_util :
  ('a -> 'b) * ('a * 'b) list * (('a -> 'b) -> 'c) -> 'c = <fun>
val mupdate_cps : ('a -> 'b) * ('a * 'b) list -> 'a -> 'b = <fun> *)
let rec mupdate_cps_util(env, bl, cnt) =
    match bl with
    | [] -> cnt env
    | (x, v) :: rest -> mupdate_cps_util(update(env, (x, v)), [],
            fun a -> mupdate_cps_util(a, rest, fun b -> cnt b))
let rec mupdate_cps(env, bl) = mupdate_cps_util(env, bl, fun x -> x)

(* Defunctionalisation:
   1. a list of continuations fun x -> e
    fun a -> mupdate_cps_util(a, rest, fun b -> cnt b)
    fun b -> cnt b
    fun x -> x
   2. get a set of free variables fv(e) - {x}
   3. define a new data structure for the continuations
      with types composed of free variables
   4. an apply function, each fun type maps to a dfc(e)
*)

type ('a,'b) funs =
|CNT_MUPDATEA of ('a * 'b) list * ('a,'b) funs
|CNT_MUPDATEB of ('a,'b) funs
|CNT_ID

(* val apply_funs : (string, 'a) funs * string -> 'a = <fun> *)
let rec apply_funs = function
|(CNT_MUPDATEA(rest, cnt), a) ->  mupdate_cps_dfc(a, rest, CNT_MUPDATEB(cnt))
|(CNT_MUPDATEB(cnt), b) -> apply_funs(cnt, b)
|(CNT_ID, x) -> x

and mupdate_cps_dfc(env, bl, cnt) =
    match bl with
    | [] -> apply_funs(cnt, env)
    | (x, v) :: rest -> mupdate_cps_dfc(
      update(env, (x, v)), [], CNT_MUPDATEA(rest, cnt))

(* Turn to a list of continuations *)
type ('a, 'b) tag =
    |MUPDATEA of ('a * 'b) list
    |MUPDATEB
type ('a, 'b) tag_list = ('a, 'b) tag list

(*  val apply_tag_list_cnt :
    ('a, 'b) tag list * ('a -> 'b) -> 'a -> 'b = <fun>
    val mupdate_cps_dfc_tags :
     ('a -> 'b) * ('a * 'b) list * ('a, 'b) tag list -> 'a -> 'b = <fun> *)
let rec apply_tag_list_cnt = function
    |([], x) -> x
    |(MUPDATEA(rest)::cnt, a) -> mupdate_cps_dfc_tags(a, rest, MUPDATEB::cnt)
    |(MUPDATEB::cnt, b) -> apply_tag_list_cnt(cnt, b)

and mupdate_cps_dfc_tags(env, bl, cnt) =
     match bl with
    | [] -> apply_tag_list_cnt(cnt, env)
```

```
    | (x, v) :: rest -> mupdate_cps_dfc_tags(
      update(env, (x, v)), [], MUPDATEA(rest)::cnt)
```

💡 Comments:

Problem 4

Below is the code for (uncurried) map, with a test using `fib`, can you apply the CPS transformation to map to produce `map_cps`? Will this `map_cps` work with `fib`? If not, what to do?

```
(* map : ('a -> 'b) * 'a list -> 'b list *)
let rec map(f, l) =
    match l with
    | [] -> []
    | a :: rest -> (f a) :: (map(f, rest))

(* fib : int -> int *)
let rec fib m =
    if m = 0
    then 1
    else if m = 1
         then 1
         else fib(m - 1) + fib (m - 2)

let map_test_1 = map(fib, [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10])
(* val map_test_1 : int list = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89] *)
```

- The function `map_cps` does not work with `fib` because the required function type is `'a -> 'a list` rather than `'a -> 'a`, but we can transform the function argument `f` to make it work.

```
(* val map_cps : ('a -> 'a list) * 'a list * ('a list -> 'b) -> 'b = <fun> *)
let rec map_cps(f, l, cnt) =
    match l with
    |[] -> cnt l
    |[a] -> cnt (f a)
    |a :: rest -> map_cps(f,[a], fun x ->
                  map_cps(f, rest, fun y -> cnt (x @ y)))

(* val cnt : 'a -> 'a = <fun> *)
let cnt(x) = x

(* val map_cps_2 : ('a -> 'a) * 'a list * ('a list -> 'b) -> 'b = <fun> *)
let map_cps_2(f, l, cnt) = map_cps((fun x -> [f x]), l, cnt)
```

```
let map_test_2 = map_cps_2(fib, [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10], cnt)
(* val map_test_2 : int list = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89] *)
```

💡 Comments:

Question 3

How is the state data type in interpreter 1 used? What is the difference between EXAMINE and COMPUTE constructors? Discuss in detail.

- In interpreter 1, a state instance is either an `EXAMINE` instance or a `COMPUTE` instance. `EXAMINE` instance is a syntactic analysis stage instance, `COMPUTE` instance is a computation stage instance.
- We will evaluate a list of expression tokens into a list of continuations, and do computation on continuations whenever we are allowed to. Eventually, evaluate the list of expression tokens into a value.
- We can transfer from `EXAMINE` to a different `EXAMINE` state, or from `COMPUTE` to a different `COMPUTE` state, or transit between `EXAMINE` and `COMPUTE` states. So the `state` data type indicates which stage we are in and what is the next stage we can transfer to via a deterministic `step` function.
- An `EXAMINE` instance encapsulates the current expression token that needs to be evaluated in the current environment, the current environment and a list of continuations that have already been converted from the expression tokens.
- A `COMPUTE` instance encapsulates a list of continuations converted from expression tokens, and a previous value computed. Then do corresponding operations on those expression tokens and the previous value to convert them into a new value.

```
type state =
  | EXAMINE of expr * env * continuation
  | COMPUTE of continuation * value
```

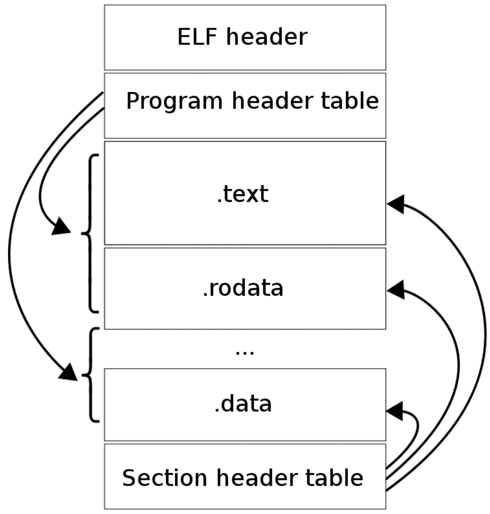
> 💡 Comments:

Question 4

How can multiple source files be compiled separately and linked together? How and which parts of ELF help you do that?

- Source files will be compiled into separate object files. Name resolution is then done to note all undefined symbols in the object files. A recursive library search is used to find the definitions of those symbols, and add those definitions into the ELF file, specifically, into one of the sections.
    - The section header of an ELF file is used at link-time. For an executable file, there are four main sections: text (executable code), data (initialised data with read/write access rights), rodata (read-only data), bss (uninitialised data with read/write access rights), so those definitions of those symbols will be added into one of the sections.
- If no link errors are generated, then we can concatenate the code segments to form output code segment, concatenate data segments to form output data segment, then perform address relocations to update code or data segments at the specified offset, create a single address space.
    - A segment is a group of sections, defined by the program header table of the ELF file. It is used at run-time. It tells the system how to create a process image, the kernel will map those segments into the virtual address space.
- ELF (executable-and-linkable format) is the common standard file format for executable files and object code. It supports different CPU or instruction set architecture, allowing it to be adopted by different operating systems and hardware platforms. Each ELF file is made up of:
    - One ELF header: (1) identify the file format (ELF), 32-bit or 64-bit, uses little or big endianness and machine instruction set architecture; (2) metadata about program header table and section header table
    - Followed by file data: (1) consists of a program header table describing zero or more sections, data referred to by entries from the program header table, then the section header table; (2) program header shows the segments used

at run-time, (3) section header defines sections, used for linking and relocation.



source: https://linuxhint.com/understanding_elf_file_format/

💡 Comments:

Question 5(y2017p3q4)

Consider the following simple evaluator for a language of expressions written in OCaml.

```
type expr =
   | Integer of int            (* integer                *)
   | Pair of expr * expr       (* pair                   *)
   | Apply of string * expr    (* apply a named function *)

type value =
     | INT of int
     | PAIR of value * value

(* eval : expr -> value *)
let rec eval = function
   | Integer n      -> INT n
   | Pair (e1, e2)  -> PAIR (eval e1, eval e2)
   | Apply (f, e)   -> eval_function(f, eval e)
```

In this code the function `eval_function` has type `string * value -> value` and is used to evaluate some "built in" functions. For example,

```
eval_function("add", PAIR(INT 10, INT 7))
```

could return the value `INT 17`.

(*a*) Rewrite the `eval` function in continuation passing style (CPS) to produce a function `eval_cps` so that the function

```
let eval_2 e = eval_cps (fun x -> x) e
```

will produce the same results as the function `eval`. [10 marks]

- The function `eval` is a recursive function using OCaml's runtime stack, so we try to indicate the explicit evaluation order for `e1` and `e2` in `Pair (e1, e2)`, where continuation `fun a -> ...` expects the computation of `eval(e1)`, the continuation `fun b -> ...` expects the computation of `eval(e2)`, then construct a pair `PAIR(a,b)`.
- In `Apply (f, e)`, the continuation `fun c ->` expects the computation of `eval(e)`.

```
let rec eval_cps_util (exp, cnt) =
    match exp with
    |Integer n -> cnt (INT n)
    |Pair (e1, e2) -> eval_cps_util(e1,
        fun a -> eval_cps_util(e2, fun b -> cnt (PAIR(a, b))))
    |Apply (f, e) -> eval_cps_util(e, fun c -> eval_function(f, CNT(c)))
```

```
let eval_cps cnt exp = eval_cps_util (exp, cnt)
```

(b) Eliminate higher-order continuations from your `eval_cps` function. That is, introduce a data type `cnt` to represent continuations and write functions of type

```
eval_cps_dfn : cnt -> expr -> value
apply_cnt    : cnt * value -> value
eval_3       : expr -> value
```

using the technique of defunctionalisation. Note that functions `eval_cps_dfn` and `apply_cnt` will be mutually recursive. [10 marks]

```
type cnt =
    |CNTA of expr * cnt
    |CNTB of value * cnt
    |CNTC of string * cnt
    |ID

let rec apply_cnt = function
    |(CNTA(e2, cnt),a) -> eval_cps_dfn (CNTB(a, cnt)) e2
    |(CNTB(a, cnt),b) -> apply_cnt(cnt, PAIR(a, b))
    |(CNTC(f, cnt),c) -> eval_function(f, apply_cnt(cnt, c))
    |(ID,x) -> x

and eval_cps_dfn cnt exp =
    match exp with
    |Integer n -> apply_cnt(cnt, INT n)
    |Pair (e1, e2) -> eval_cps_dfn (CNTA(e2, cnt)) e1
    |Apply (f, e) -> eval_cps_dfn (CNTC(f, cnt)) e

let eval_3 e = eval_cps_dfn (ID) e
```

Comments:

Question 6

Write a program with the following specification on Linux outputting an ELF binary using x86_64 instruction set:

Input: An integer N (supplied as command line argument or standard input)

Output: N lines. In the first of which there is a `*` and in each subsequent lines there is an extra star.

Example:

```
Input: 5
Output:
*
**
***
****
*****
```

```
/*
example file: stars.c
*/
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int n = argc == 2 ? atoi(argv[1]) : 0;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            printf("*");
        }
        printf("\n");
    }
    return 0;
}

/* after compiling with gcc stars.c to produce the binary file a.out
a.out:
ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.32, not stripped
*/

/* inspect the ELF header */
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400570
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6488 (bytes into file)
  Flags:                             0x0
```

```
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29

/* inspect the ELF sections */
There are 30 section headers, starting at offset 0x1958:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .hash             HASH             0000000000400278  00000278
       0000000000000028  0000000000000004   A       4     0     8
  [ 4] .dynsym           DYNSYM           00000000004002a0  000002a0
       0000000000000078  0000000000000018   A       5     1     8
  [ 5] .dynstr           STRTAB           0000000000400318  00000318
       0000000000000151  0000000000000000   A       0     0     1
  [ 6] .gnu.version      VERSYM           000000000040046a  0000046a
       000000000000000a  0000000000000002   A       4     0     2
  [ 7] .gnu.version_r    VERNEED          0000000000400478  00000478
       0000000000000020  0000000000000000   A       5     1     8
  [ 8] .rela.dyn         RELA             0000000000400498  00000498
       0000000000000018  0000000000000018   A       4     0     8
  [ 9] .rela.plt         RELA             00000000004004b0  000004b0
       0000000000000048  0000000000000018  AI       4    23     8
  [10] .init             PROGBITS         00000000004004f8  000004f8
       000000000000001a  0000000000000000  AX       0     0     4
  [11] .plt              PROGBITS         0000000000400520  00000520
       0000000000000040  0000000000000010  AX       0     0     16
  [12] .plt.got          PROGBITS         0000000000400560  00000560
       0000000000000008  0000000000000000  AX       0     0     8
  [13] .text             PROGBITS         0000000000400570  00000570
       00000000000001e2  0000000000000000  AX       0     0     16
  [14] .fini             PROGBITS         0000000000400754  00000754
       0000000000000009  0000000000000000  AX       0     0     4
  [15] .rodata           PROGBITS         0000000000400760  00000760
       0000000000000004  0000000000000004  AM       0     0     4
  [16] .eh_frame_hdr     PROGBITS         0000000000400764  00000764
       000000000000003c  0000000000000000   A       0     0     4
  [17] .eh_frame         PROGBITS         00000000004007a0  000007a0
       000000000000010c  0000000000000000   A       0     0     8
  [18] .init_array       INIT_ARRAY       0000000000600e00  00000e00
       0000000000000008  0000000000000008  WA       0     0     8
  [19] .fini_array       FINI_ARRAY       0000000000600e08  00000e08
       0000000000000008  0000000000000008  WA       0     0     8
  [20] .jcr              PROGBITS         0000000000600e10  00000e10
       0000000000000008  0000000000000000  WA       0     0     8
  [21] .dynamic          DYNAMIC          0000000000600e18  00000e18
       00000000000001e0  0000000000000010  WA       5     0     8
  [22] .got              PROGBITS         0000000000600ff8  00000ff8
```

```
       0000000000000008  0000000000000008  WA       0     0     8
  [23] .got.plt          PROGBITS          0000000000601000  00001000
       0000000000000030  0000000000000008  WA       0     0     8
  [24] .data             PROGBITS          0000000000601030  00001030
       0000000000000010  0000000000000000  WA       0     0     8
  [25] .bss              NOBITS            0000000000601040  00001040
       0000000000000008  0000000000000000  WA       0     0     1
  [26] .comment          PROGBITS          0000000000000000  00001040
       000000000000003e  0000000000000001  MS       0     0     1
  [27] .symtab           SYMTAB            0000000000000000  00001080
       0000000000000600  0000000000000018          28    46     8
  [28] .strtab           STRTAB            0000000000000000  00001680
       00000000000001df  0000000000000000           0     0     1
  [29] .shstrtab         STRTAB            0000000000000000  0000185f
       00000000000000f5  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

💡 Comments:

Question 1

Read the paper by Reynolds (1972) up to and including section seven. Later sections are delightful as well, but less relevant to the course. Section one can be skimmed, but it nicely sets up the scene. He talks about Algol a lot. For the unfamiliar it is the archetype of block structured programming languages *e.g.* C, but unlike C it has *some* high-order function support.

> This is a great paper in computer science. Apart from historical value, it is also what the lecturer basis his interpreter upon. It will allow you to deeply understand how and why we bother with defunctionalisation & continuation-passing style.

> I strongly suggest you read it before you attempt the rest of the workset.

Notes for the paper

- A defined language is defined by an interpreter that is written in a defining language.
    - The variety of values provided in the defining language is richer than in the defined language, we can represent each defined-language value by the same defining-language value
- Applicative features of a language include evaluation and the definition and application of functions.
    - Purely applicable languages are based on lambda calculus, but the semantics of the “real” lambda calculus implies a different order of application than most applicative programming languages
- Imperative features of a language include statement sequencing, labels, jumps, assignment, and procedural side-effects. e.g., purely imperative language such as machine languages
- Higher-order programming language: procedures or labels can be treated as data (e.g., used as arguments to procedures, as results of functions, or as values of assignable variables)
    - In contrast, we have first-order language.
- Function order of application:
    - (1) Call by value: the application process does not begin until after the operator and all of its operands have been evaluated.
    - (2) Call by name: the application process would begin as soon as the operator had been evaluated, and each operand would only be evaluated when and if the function being applied actually depended upon its value.
    - If any arguments cause an infinite loop and are not used in the function body, then call-by-name can terminate while call-by-value cannot.
- Expressions: the meaningful phrases of a program (e.g., constants and variables)
- Evaluation: the process of evaluating an expression
- Value: the result of evaluating an expression
- Environment: binds variables to values, aid in the evaluation of expressions
- For lambda function application $\lambda(x_1, ..., x_n).r$, the environment in which the body is evaluated during application is an extension of the earlier environment in

which the lambda expression was evaluated (i.e., we have binds the value to the variables $x_1, ..., x_n$ in the body r.

  - However when evaluating $x_1, ..., x_n$, we used the same environment, this means we cannot use `let` to define recursive functions, but introduce `letrec`

    ```
    let x = x + 1 and y = x - 1 in x * y
    (* with input x = 4, output 15, y = 4 - 1 = 3,
       not using the updated environment *)

    (* use let to define recursive functions is invalid *)
    let f = λx. if x = 0 then 1 else x × f(x-1) in ...
    (* the occurrence of f inside the declaration
       cannot feel the binding of f *)

    LAMBDA = [fp: VAR, body: EXP]
    evlambda = λ(l,e).λa.eval(body(l), ext(fp(l),a,e))
    where ext = λ(z,a,e).λx.if x = z then a else e(x)
    (* ext produces an extended environment
       we evaluate l in env e, evaluate body(l) in env ext(fp(l),a,e) *)

    LETREC = [dvar: VAR, dexp: LAMBDA, body: EXP]
    letrec?(r) -> (* if r is a recursive function *)
    letrec e' = λx. if x = dvar(r)
      then evalambda(dexp(r), e') else e(x) in eval(body(r), e'))
    ```

- Continuation: provides an additional degree of freedom that can be used to meet the condition of order-of-application independence.
  - Instead of performing actions after the function has returned, we embed the further actions in the continuation as an argument to the function.