
MP 9 – CPS and a Lazy Interpreter for MicroML

CS 421 – Spring 2007
Revision 1.1

Assigned Wednesday, April 18, 2007

Due Wednesday, April 25, 2007 at 23:59

Extension two days (20% penalty)

1 Change Log

1.1 Changed “non-empty” to “empty” in Q7.

1.0 Initial release

2 Overview

There are two problem sections in this MP: the first is on Continuation Passing Style (CPS) and the second asks you to write a CPS interpreter for MicroML that supports *lazy evaluation* based on an operational semantics for MicroML. The first part is required for the whole class; the second part is required for the graduate students and extra credit for the undergraduate students.

Upon completion of this MP (if you do both parts), you should have the following skills:

- the ability to use CPS effectively,
- the ability to implement a call-by-need language,
- the ability to write a realistic interpreter for an ML-like language, and
- the ability to laugh sincerely when someone makes a continuation passing style joke at a cocktail party

Enjoy!

3 Files

There are two graders, **mp9graderA** and **mp9graderB**. The first grader is for the first part of this assignment, and the second is for the second part of the assignment.

The only files you should modify and submit are **mp9A.ml** and **mp9B.ml**.

mp9graderA is fairly simple; it contains only the file **mp9A-skeleton.ml**, which you should rename to **mp9A.ml**.

mp9graderB is a bit more complicated, but it is very similar to the grader for the last assignment; in particular, upon running **make** or **gmake** you will get three executables: **mp9B**, **mp9BSol** and **grader**.

The file **mp9common.cmo** contains the datatype for expressions (input to your evaluator) and the datatypes for values (output of your evaluator). For this MP, **mp9common.cmo** also defines the `thunk` type, to be explained below.

You are also given files **mp9lex.cmo**, **mp9yacc.cmo**, etc...

4 Continuation Passing Style

These exercises are designed to give you a feel for continuation passing style before you build your interpreter.

A function that is written in continuation passing style does not return once it has finished its computation. Instead, it calls another function (the continuation) with the result.

Here is a small example:

```
# let report x =
  print_string "Result: ";
  print_int x;
  print_newline()
val report : int -> unit = <fun>

# let inck i k = k (i+1)
val inck : int -> (int -> 'a) -> 'a = <fun>
```

The `inck` function takes an integer and a continuation. After adding 1 to the integer, it passes the result to its continuation.

```
# inck 3 report;;
Result: 4
- : unit = ()
# inck 3 inck report;;
Result: 5
- : unit = ()
```

In line 1, `inck` increments 3 to be 4, and then passes the 4 to `report`. In line 4, the first `inck` adds 1 to 3, and passes the resulting 4 to the second `inck`, which then adds 1 to 4, and passes the resulting 5 to `report`.

5 Part A - CPS Problems (required for all)

You have been given a file called `mp9A-skeleton.ml`. You should write your solutions to the CPS problems in that file.

The following six problems demonstrate different techniques of CPS. None of the functions you write may directly return a value; all must be written in CPS.

Simple Continuations

1. (5 pts) Write the functions `subk`, `catk`, `doublek`, `plusk`, `multk`, and `is_posk` in CPS. `subk` subtracts one integer from another; `catk` concatenates two strings; `double k` concatenates a string with itself, `plusk` adds two floats, `multk` multiplies two floats, and `is_posk` determines if an integer is greater than 0.

```
# let subk n m k = ...;;
val subk : int -> int -> (int -> 'a) -> 'a = <fun>
# let catk a b k = ...;;
val catk : string -> string -> (string -> 'a) -> 'a = <fun>
# let doublek a k = ...;;
val doublek : string -> (string -> 'a) -> 'a = <fun>
# let plusk x y k = ...;;
val plusk : float -> float -> (float -> 'a) -> 'a = <fun>
# let multk x y k = ...;;
val multk : float -> float -> (float -> 'a) -> 'a = <fun>
# let is_posk n k = ...;;
```

```

val is_posk : int -> (bool -> 'a) -> 'a = <fun>
# subk 10 5 report;;
Result: 5
- : unit = ()
# catk "hi " "there" (fun x -> x);;
- : string = "hi there"
# doublek "pom " (fun x -> x);;
- : string = "pom pom "
# plusk 3.0 4.0
  (fun x -> multk x x
   (fun y -> (print_string "Result: ";print_float y; print_newline())));;
  Result: 49.
- : unit = ()
# is_pos 7 (fun b -> (report (if b then 3 else 4)));;
Result: 3
- : unit = ()

```

Nesting Continuations One common technique used in CPS is that of nesting continuations. For example, consider the following code:

```

# let add3k a b c k =
  addk a b (fun ab -> addk ab c k);;
val add3k : int -> int -> int -> (int -> 'a) -> 'a = <fun>
# add3k 1 2 3 report;;
Result: 6
- : unit = ()

```

We needed to add three numbers together, but `addk` itself only adds two numbers. On line 2, we give the first call to `addk` a function that saves the sum of `a` and `b` in the variable `ab`. Then this function adds `ab` to `c` and passes its result to the continuation `k`.

- (5 pts)** Using `multk` and `plusk` as helper functions, write a function `abcdk`, which takes four float arguments $a b c d$ and “returns” $a * (b + c) * d$. You may not use the normal `*` operator or `+` operators; you must instead use `multk` and `plusk`.

```

# let abcdk a b c d k = ...
val abcdk : float -> float -> float -> float -> (float -> 'a) -> 'a = <fun>
# abcdk 2.0 3.0 4.0 5.0 (fun y -> report (int_of_float y));;
Result: 70
- : unit = ()

```

Recursion and Continuation How do we write recursive programs in CPS? Consider the following recursive function:

```

# let rec factorial n =
  if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120

```

To put the function into CPS, we must make factorial take an additional argument, a continuation, to which the result of the factorial function should be passed. When the recursive call is made to factorial, instead of it returning a result to build the next higher factorial, it needs to take a continuation for building that next value from its result. Thus the code becomes:

```
# let rec factorialk n k =
if n = 0 then k 1 else factorialk (n - 1) (fun m -> k (n * m));;
val factorialk : int -> (int -> 'a) -> 'a = <fun>
# factorialk 5 report;;
120
- : unit = ()
```

To make a recursive call, we must built an intermediate continuation to:

- take the recursive value: m
- build it to the final result: n * m

And pass it to the final continuation: k (n * m). Notice that this is an extension of the "nested continuation" method.

3. (10 pts) Write gcdk m n k, the CPS version of:

```
let rec gcd m n =
  let d1 = m - n
  in if d1 > 0 then gcd d1 n
     else let d2 = n - m in if d2 > 0 then gcd m d2 else m
```

You should use subk for the subtractions and is_posk for the comparisons to zero. You may not use - or > or < in this problem.

Also, you must ensure that evaluations are done in the same order as they are done in gcd (so, for example, you must evaluate d1 first, and then d1 > 0, etc...)

```
# let rec gcdk m n k = ...
val gcdk : int -> int -> (int -> 'a) -> 'a = <fun>
# gcdk 6 8 report;;
Result: 2
- : unit = ()
```

4. (10 pts)

Write a function forallk fk lst k which takes a CPS boolean function fk and a list lst and "returns" true if fk holds for all elements in lst, and false otherwise.

```
# let rec forallk fk lst k = ...
val forallk : ('a -> (bool -> 'b) -> 'b) -> 'a list -> (bool -> 'b) -> 'b = <fun>
# forallk is_pos [5;6;7;8;9] (fun x -> x);;
- : bool = true
# forallk is_pos [5;6;-7;-8;-4] (fun x -> x);;
- : bool = false
```

Mapping Recursion Suppose we want to use CPS to write a function that increments every element of a list. Here is the resulting code:

```
# let rec incList lst k =
  match lst with
  | [] -> k []
  | x::xs -> incK x (fun v ->
                    incList xs (fun vs ->
                                k (v::vs)));;
val incList : int list -> (int list -> 'a) -> 'a = <fun>
# incList [2;3;4] report_list;;
Result: [3; 4; 5]
- : unit = ()
```

This is really just again a variation of the “nested continuation” method. The first continuation passed to `incK` saves the result of incrementing `x` as `v`. The continuation then calls `incList` on `xs`, which will increment the tail of the list. This result is saved in the second continuation as `vs`. This second continuation then combines the parts together, and gives the result to the original continuation.

5. (10 pts) Write the function `list_appk` that passes the final continuation a list of the results of applying to the given argument each function in the list. The functions in the list should be applied in reverse order (i.e., right to left) even though the order of the result list should not be reversed.

```
# let rec list_appk funs arg k = ...;;
val list_appk : ('a -> ('b -> 'c) -> 'c) list -> 'a -> ('b list -> 'c) -> 'c =
<fun>
# let funs = [(fun x k -> (let y = x + 2 in (report y; k y)));
              (fun x k -> (let y = x + 5 in (report y; k y))];;
  val funs : (int -> (int -> 'a) -> 'a) list = [<fun>; <fun>]
# list_appk funs 7 (fun x -> x);;
Result: 12
Result: 9
- : int list = [9; 12]
```

Folding Recursions The other common recursion is folding recursion. Look at this example and observe how it differs from the previous example.

```
let rec sumList list k =
  match list with
  | [] -> k 0
  | x::xs -> sumList xs (fun y -> addk x y k)
```

Notice here that the recursive call comes first. The result is saved in a variable `y` to remind you of the original `fold_right` definition. You know already that the recursion has to occur before the current element can be combined with it; this definition makes that explicit. This is a major feature of CPS: the order of operations is made explicit.

```
# sumList [2;3;4;5] report;;
Result: 14
- : unit = ()
```

6. (10 pts) Write the function `map_appendk fk lst1 lst2 k`, which “returns” `fk` mapped over `lst1`, all pre-pended to `lst2`, but written in continuation-passing style.

Here also the applications of `fk` should happen in reverse order.

```
let rec map_appendk fk lst1 lst2 k = ...;;
val map_appendk :
  ('a -> ('b -> 'c) -> 'c) -> 'a list -> 'b list -> ('b list -> 'c) -> 'c =
  <fun>
# map_appendk (fun n -> report n; is_posk n) [-2;-1;0;1;2] [false;false;true]
      (fun x -> x);;
Result: 2
Result: 1
Result: 0
Result: -1
Result: -2
- : bool list = [false; false; false; true; true; false; false; true]
```

You may not use any functions from the List library in this problem.

Saving Continuations Recall from lecture that by saving the continuation passed to the initial call of a recursive function we have the ability to abort a computation before it has even begun. The usual method of accomplishing this is to make a local auxiliary function do all the recursion.

```
let prodList list ka =
  let rec aux list k =
    match list with
    | [] -> k 1
    | 0::xs -> ka 0
    | x::xs -> aux xs (fun y -> k (x * y))
  in aux list ka
```

If a zero is encountered, the continuation `k` is thrown out and the original continuation `ka` is used instead.

7. Write a function `list_hds_k` that when applied to a list of lists supplies to its continuation the list of the heads of the element lists, if every element list is non-empty. If there is an element list that is empty, then `list_hds_k` should directly supply to the original continuation the empty list.

```
# let list_hds_k l k = ...
val list_hds_k : 'a list list -> ('a list -> 'b) -> 'b = <fun>
# list_hds_k [[1;2;24]; [3;5;-5]; [7]] (fun l -> List.map report l);;
Result: 1
Result: 3
Result: 7
- : unit list = [(); (); ()]
# list_hds_k [[15;23]; [145;93]; []; [51;23;67]] (fun l -> List.map report l);;
- : unit list = []
```

General Problems Try the rest of these problems.

8. Write `listifyk` that supplies a continuation with a list of lists that is the result of making every element in an input list into a singleton list.

```
# let rec listifyk l k = ...
  val listifyk : 'a list -> ('a list list -> 'b) -> 'b = <fun>
# listifyk [1;2;3] (fun x -> x);;
- : int list list = [[1]; [2]; [3]]
# listifyk [5;6;7] (fun l -> list_hds_k l) (fun x -> x);;
- : int list = [5; 6; 7]
```

9. Write `splitk` which takes a CPS function `fk`, a list `lst`, and a continuation `k` and passes on to `k` a list of lists whose concatenation is `lst`. Each list in the list should have at most one element for which `fk` passes on `true` to its continuation, and if one such element does occur, it must occur last in its list. If no such element occurs, then the list must contain the last element of the original list.

```
# let rec splitk fk lst k = ...
val splitk :
  ('a -> (bool -> 'b) -> 'b) -> 'a list -> ('a list list -> 'b) -> 'b = <fun>
# splitk is_posk [1;2;-99;-4;8;-99;3;-2] (fun x -> x);;
- : int list list = [[1]; [2]; [-99; -4; 8]; [-99; 3]; [-2]]
```

6 Part B (extra credit for undergrads)

Before you start any work, there are a few new things you should familiarize yourself with.

6.1 Values and Memories

Since we are evaluating MicroML, we must talk about the values in the language. The values of a language encompass everything that an expression can evaluate to. They are defined in `mp9common.cmo` as the `value` disjoint datatype.

We also need to keep track of memories (value environments), which are mappings from variables to their values. Compare this with type environments, which map variables to their types. In general, throughout this writeup, the symbol *m* is used to denote an arbitrary memory.

6.2 Thunks and Forcing

Evaluation for this MP is **call-by-need**, meaning that an expression is not evaluated until it is first used. The difference between this and the lazy evaluation of the lambda-calculus (which is **call-by-name**) is that once an expression is evaluated, it is never evaluated again. Instead, its value is memoized. For an example of this, see problem 17.

To accomplish this, we introduce a special kind of value, called a *thunk* (Thunk). There are two kinds of thunks:

- A suspended thunk, `Suspension`, which holds an expression and a memory.
- An evaluated thunk `Value`, which just points to a non-thunk value.

A suspended thunk is similar to a closure. Remember that when a function is created, the result is a triple consisting of: the variable to which the argument is bound, the function expression, and the memory that was active when the function was created. A suspended thunk is similar, except that the enclosed expression is able to be something

other than a function, and there is no variable. When a suspension is evaluated, we get the other kind of thunk, a `Value`, which contains an evaluated expression.

This is how we implement lazy evaluation, and in particular, call-by-need parameter passing. In call-by-value, we would evaluate a function's argument to a value, then pass that value to the function. In call-by-need, we create a thunk out of the expression for the argument, and pass the thunk value to the function. It only gets computed when needed! Whenever an operation does require the actual (non-thunk) value of something, we force the thunk. Forcing means that we actually evaluate the suspended expression to get a value. In addition, the thunk gets changed to an evaluated thunk, where it stores the resulting value. This way, we don't have to evaluate the thunk again if it is used more than once. Thunks are represented by the `Thunk` constructor, which takes a thunk. The `thunk` datatype is defined in `mp9common.cmo`. Although you will never have to play with the internals of the `thunk` datatypes directly, it is worth noting that the `thunk` datatype stores a **reference** to either a `Suspension` or a `Value`. This is how we are able to change a thunk from a suspended one to an evaluated one (which would be hard without somehow using side-effects).

There are some support functions in `mp9common.cmo` to help you use thunks:

- `mkThunk` — takes an expression and an environment, and creates a thunk with them.
- `force` — takes a thunk and a continuation, forces the thunk, and passes the resulting thunk to the continuation. Suspensions are evaluated and turned into values. Values are “returned” (passed to the given continuation) as is.

6.3 Types

Since you cannot access the source for `mp9common.cmo`, here are the important types:

```
type thunker = Suspension of exp * memory | Value of value
and thunk = thunker ref
and memory = (string * value) list
and value = Unitval | Intval of int | Boolval of bool | Stringval of string
| Realval of float | Charval of char
| BuiltInOpval of string
| Pairval of value * value
| Listval of value list
| Closure of string * exp * memory
| Recvar of string * exp * memory
| Thunk of thunk
```

```
type dec = Valbind of (string * exp) | ValbindRec of (string * exp)
| Local of (dec * dec) | Seq of (dec * dec)
and exp =
(*special constants*)
Unit | Bool of bool | Char of char | Int of int | Real of float
| String of string
(*pair*)
| Pair of exp * exp
(*identifier*)
| Id of string
(*let in, application, functions*)
| LetIn of dec * exp | App of exp * exp | Fn of string * exp
| IfThenElse of exp * exp * exp
```

6.4 The Evaluator

The two main functions are `eval_expk ex m k` and `eval_deck dec m k`. They both take three arguments: an expression, a memory, and a continuation. The continuation, when invoked, takes a value (or memory, for `eval_deck dec m k`) and then completes the rest of the evaluation based on that value (or memory).

These functions are appropriately called from `eval_top`.

To run the interpreter, you should be able to type `gmake` and then type `mp9B`, `mp9BSol`, or `grader`.

7 Evaluator Problems

7.1 Notation

There will be a text description of each of the features, along with a mathematical description. We will use the following notation. A suspension will be represented by $\langle e, m \rangle$. The e is the suspended expression, and m is the memory (environment). Value thunks will be represented by $\langle v \rangle$. Closures will be represented as $\langle x, e, m \rangle$. Memory elements will be represented by $\{x \rightarrow v\}$, indicating that variable x is mapped to value v (either a thunk, a *rec*(...), or a computer string, nat, bool, list, etc...). The evaluator for expressions will be represented by a three-argument function $\mathcal{E}_e(e, m, k)$, where e is the expression to evaluate, m is the memory, and k is the continuation. The evaluator for declarations is called $\mathcal{E}_d(d, m, k)$. The function “force” is represented by \mathcal{F} .

You should be sure to use an “iterative” method for programming. A lot of pain can be avoided by making sure that you get one feature working before trying to add another.

10. Simple Types (5 pts)

Create `eval_expk ex m k` and make it handle unit, bools, ints, reals, characters and strings.

$$\begin{aligned}\mathcal{E}_e((), m, k) &= k () \\ \mathcal{E}_e(b, m, k) &= k b \\ \mathcal{E}_e(i, m, k) &= k i \\ \mathcal{E}_e(r, m, k) &= k r \\ \mathcal{E}_e(c, m, k) &= k c \\ \mathcal{E}_e(s, m, k) &= k s\end{aligned}$$

11. Atomic Declarations (5 pts)

Implement `eval_deck dec m k`, which takes a declaration `val x = e`, a memory m , and a continuation k , and passes to k a fresh “increment” memory where $x \rightarrow \langle e, m \rangle$.

$$\mathcal{E}_d(\text{val } x = e, m, k) = k\{x \rightarrow \langle e, m \rangle\}$$

```
> val x = ();
```

```
final memory:
val x = ();
```

```
> val x = 3;
```

```
final memory:
val x = 3;
```

```
> val x = true;

final memory:
val x = true;
```

12. **Identifiers (thunks)** (5 pts) Extend `eval_expk` to handle identifiers that go to thunks.

At the point where we actually *need* to use the value of an identifier bound to a thunk, the thunk must be forced. Note that if the thunk has been forced before, then it is already a value thunk and nothing will be done. If it is a suspension, however, it will be evaluated to a value thunk.

$$\mathcal{E}_e(x, m, k) = \mathcal{F}(v_t, k) \quad \text{when } x \rightarrow v_t \text{ is in } m, v_t \text{ is a thunk value}$$

```
> val x = (op mod);

final memory:
val x = (op mod);

> val x = (op +);

final memory:
val x = (op +);
```

13. **Pairs** (5 pts) Extend `eval_expk` to handle pairs.

We are not going to give you the rule for this one; you are on your own.

$$\mathcal{E}_e((e_1, e_2), m, k) = \dots$$

```
> val x = (3, 4);

final memory:
val x = (3, 4);

> val x = (true, 'c');

final memory:
val x = (true, 'c');
```

14. **Let...In...End Declarations** (5 pts)

Extend `eval_expk` `ex m k` to handle `let...in...end` declarations.

$$\mathcal{E}_e(\text{let } d \text{ in } e \text{ end}, m, k) = \mathcal{E}_d(d, m, \lambda \Delta. \mathcal{E}_e(e, \Delta + m, k))$$

```
> val x = let val y = 3 in 4 end;

final memory:
```

```

val x = 4;

> val x = let val y = 3 in y end;

final memory:
val x = 3;

```

15. Application of built-in operators (10 pts)

Extend `eval_expk` to handle applications of built-in operators. The function `app_builtin oper v`, which you wrote for the last MP, will come in handy. It is included in `Mp8common.cmo`, so you can use it immediately.

Since evaluation of these constants requires calling low-level OCaml functions, it is necessary to fully evaluate their arguments (this situation differs from that of function application, which should be handled in a call-by-need fashion.)

$$\mathcal{E}_e(e_1 e_2, m, k) = \mathcal{E}_e(e_1, m, \lambda o. \mathcal{E}_e(e_2, m, \lambda v_2. k(o v_2)))$$

Here o is a built-in operator. The expression $o v_2$ is implemented by calling the function `app_builtin`.

Note: This rule assumes that e_1 evaluates to a built-in operator. Sometimes, however, e_1 evaluates to a closure. For now, it is sufficient to match against the return value of the evaluation of e_1 and raise a `Failure` exception for everything but built-in operators.

In problem 17, you will implement the closure case.

```

> val x = (2 + 3) * 8;

final memory:
val x = 40;

> val x = not(true orelse false);

final memory:
val x = false;

> val x = head(2 :: 1 :: nil) :: (tail [2, 1, 0]);

final memory:
val x = [2; 1; 0];

```

16. Functions (10 pts)

Extend `eval_expk` to handle functions. You will need pass a closure to the given continuation.

$$\mathcal{E}_e(\text{fn } x \Rightarrow e, m, k) = k\langle x, e, m \rangle$$

```

> val x = fn x => x + x;

final memory:
val x = <some closure>;

```

17. Function application (10 pts)

Extend `eval_expk` to handle function application in a lazy fashion.

$$\mathcal{E}_e(e_1 e_2, m, k) = \mathcal{E}_e(e_1, m, \lambda(x, e_1', m').\mathcal{E}_e(e_2, \{x \rightarrow \langle e_2, m \rangle\} + m', k))$$

You first evaluate e_1 to a closure $\langle x, e_1', m' \rangle$. Then you evaluate e_2 in an environment where $x \rightarrow \langle e_2, m \rangle$. This is a thunk representing the evaluation of e_2 in m . Note that this evaluation will not actually take place until it is needed.

Note: In your implementation, you are adding an additional case (the case where e_1 evaluates to a closure) to your code from problem 15.

```
> val x = (fn x => fn y => x y) (fn x => x + x) 10;
```

```
final memory:
val x = 20;
```

Is your evaluation lazy (call-by-need)?

```
> val x = (fn x => 4) (print @"hi");
```

```
final memory:
val x = 4;
```

```
> val x = (fn x => (x, (x, 4))) (print @"hi");
"hi"
```

```
final memory:
val x = (((), (((), 4)));
```

The first time, `x` is never referred to so nothing gets printed. The second time, `x` is referred to twice but "hi" is only printed once, because once a thunk is evaluated, its value is memoized and never computed again.

Note: In call-by-name, "hi" would have been printed twice because it was referred to twice.

18. If constructs (10 pts)

Extend `eval_expk` to handle if constructs. We are not going to give you the rule for this one; you are on your own.

$$\mathcal{E}_e(\text{if } e_1 \text{ then } e_2 \text{ else } e_3, m, k) = \dots$$

Hint: you will need to call \mathcal{E}_e on e_1, m and k , and pass the result to some continuation that knows what to do with the result.

```
> val x = 3 + if 3 > 2 then 3 else 2;
```

```
final memory:
val x = 6;
```

19. **Local Declarations** (5 pts) Extend `eval_deck dec m k` to handle local declarations.

$$\mathcal{E}_d(\text{local } d_1 \text{ in } d_2 \text{ end}, m, k) = \mathcal{E}_d(d_1, m, \lambda\Delta.\mathcal{E}_d(d_2, \Delta + m, k))$$

```
> local val y = 7 in val x = y * y end;
```

```
final memory:
val x = 49;
```

20. **Sequential Declarations** (5 pts) Extend `eval_deck dec m k` to handle sequential declarations.

$$\mathcal{E}_d(d_1 d_2, m, k) = \mathcal{E}_d(d_1, m, \lambda\Delta.\mathcal{E}_d(d_2, \Delta + m, \lambda\Delta'.k(\Delta' + \Delta)))$$

```
> val x = 3    val y = x + x    val z = @"hi there!";
```

```
final memory:
val x = 3;
val y = 6;
val z = "hi there!";
```

21. **Recursive Declarations** (10 pts)

Extend `eval_deck dec m k` to handle declarations of recursive variables.

$$\mathcal{E}_e(\text{val rec } x = e, m, k) = \mathcal{E}_e(e, \{x \rightarrow \text{rec}\langle x, e, m \rangle\} + m, \lambda v.k\{x \rightarrow v\})$$

```
> fun f x = if x < 1 then 1 else x * f(x - 1);
```

```
final memory:
val f = <some closure>;
```

22. **Recursive variables** (10 pts)

Extend `eval_expk` to handle recursive variables. These are variables that go to `rec⟨x, e, m⟩` for some variable x , expression e , memory m .

$$\mathcal{E}_e(x, m, k) = \mathcal{E}_e(e', \{y \rightarrow \text{rec}\langle y, e', m' \rangle\} + m', k) \quad \text{when } x \rightarrow \text{rec}\langle y, e', m' \rangle \text{ is in } m$$

```
> fun f x = if x < 1 then 1 else x * f(x - 1)    val result = f 10;
```

```
final memory:
val f = <some closure>;
val result = 3628800;
```