Imperative Programming

In this assignment, we will consider the fundamentals of imperative programming, i.e. programming with side effects. Please submit answers to the 3 programming exercises using Learn OCaml and the 3 written problems using Gradescope.

1 Background

1.1 Side Effects

So far this semester, we have been working with *pure expressions*. Evaluation of a pure expression produces a value (or fails to terminate), and you get the same value every time you evaluate that expression. The programmer does not need to worry about any "side effects" that might occur during evaluation that affect the result or the state of the outside world. This means we can use equational reasoning to describe (i.e. specify) program behavior.

There are a great many useful equational properties that we can take advantage of to describe, simplify, and optimize programs, e.g. the map fusion property that you proved in A2, which is used to optimize the performance of commercial map-reduce frameworks (e.g. Hadoop, developed in part here at Michigan) for analyzing large datasets.

Additionally, having equations can simplify the debugging process: you can rule out potential fixes that could not possibly change the meaning of a program that is behaving incorrectly, e.g. reorderings, and focus on the smaller set of changes to the program that could change its meaning.

With a sufficiently powerful expression language, like the one in OCaml and increasingly many other modern languages, pure expressions suffice for a surprising variety of computational tasks. Yet the outside world continues to exist, and we must confront it in all of its impurity! Side effects are necessary to implement programs that are useful to humans: we do not want to simply heat our computer up as it evaluates expressions; we need some way to look at the output and interactively provide input to our programs using physical hardware!

Alas, adding side effects to programs, e.g. expressions that allocate, read, and write to memory, output graphics to the screen, or send data over the network, is not strictly a good thing: it complicates program reasoning and causes the loss of many equational properties that we previously enjoyed. Let's investigate in this assignment.

1.2 Imperative Programming with References

For this assignment, we will focus on side effects on memory.[1] OCaml has support for side effects on memory through the `t ref` type (as well as mutable record fields and mutable arrays, which we will not consider in detail here, but which are fundamentally similar). A value of type `t ref` is a reference to a location in the mutable region of memory (sometimes called "the heap") containing a value of the type `t` (which can be any type at all, e.g. an integer, a function, a list, or even another reference).

We can perform the side effect of allocating a fresh location in memory and initializing it with a given value `e`, using the `ref e` operation. The value of `ref e` is the location (i.e. memory address) of this freshly allocated memory. For example, we can allocate a fresh counter and initialize it to zero as follows:

```
let counter = ref 0 in ...
```

[1]Graphics I/O, network I/O, etc. can often be understood as reading and writing to special OS-defined references.

After evaluating this line, the value of the variable `counter` of type **`int ref`** will be a reference (think pointer) to some OS-determined memory location (a.k.a. address), perhaps `0xbaddecaf`. We cannot inspect or manipulate memory addresses directly in OCaml, nor convert to and from integers, like you can with pointers in C/C++ (we'll talk about why next week). We can only work with (i.e. eliminate) references using two other effectful operations: dereferencing (read) and assignment (write).

Dereferencing, written `!e` when `e` evaluates to a reference, has the side effect of going to the location that the value of `e` refers to and reading the value it points to at that moment. For example, the value of `!counter` if evaluated immediately after the line above will be `0`. The rule is: if `e` has type `t` **`ref`** then `!e` has type `t`. Think of it like dereferencing in C/C++, i.e. `*e`.

Assignment expressions, written `e1 := e2`, have the side effect of writing the value of `e2` to the location that the value of `e1` refers to. For example, `counter := !counter + 1` has the side effect of incrementing the counter. Think of it as similar to the statement `*counter = *counter + 1` in C/C++ (though in OCaml, e_1 can be an arbitrary expression, not just an "l-value"). An assignment expression is useful *only* for this side effect, so its value is the trivial value, i.e. `()`.[2] Consequently, we can summarize its typing as follows: if `e1 : t` **`ref`** and `e2 : t` then `e1 := e2 :` **`unit`**.

1.3 Order Matters

When evaluating pure expressions, order of evaluation does not matter. For example, given a pair `(e1, e2)`, we might choose to evaluate `e1` before `e2`, or *vice versa*. The result will be the same. Indeed, we can even evaluate the two expressions in parallel, as we will discuss later this semester.

When we add side effects on memory, order now matters! For example, if `e1` assigns to a reference that `e2` dereferences, then evaluating `e1` first may produce a different value for `e2`. (For the same reason, it is not sensible to evaluate the two expressions in parallel in general.)

Exercise 1: Map Fusion Counterexample (Written, 20%) Recall the Map Fusion exercise from Assignment 2. The necessary definitions, now in OCaml, are reproduced below.

```
    let compose  = fun f g -> fun x -> f (g x)

    let rec map f xs = match xs with
    | [] -> []
    | hd::tl -> (f hd)::(map f tl)
```

In the pure setting, you proved the following property in A2.

Theorem (Map Fusion). *For all* `xs : 'a list` *and* `g : 'a -> 'b` *and* `f : 'b -> 'c` *, where* `g` *and* `f` *are both total, we have that* `map (compose f g) xs` $\equiv$ `map f (map g xs)` *.*

If we allow `f` and `g` to have side effects on memory, however, this theorem no longer holds. Prove that map fusion does **not** hold in all memory states by filling in holes `_1` and `_2` in the OCaml code below such that filling hole `_3` with `map f (map g [0.7; 0.8; 0.9])` produces a *different* value than filling `_3` with `map (f % g) [0.7; 0.8; 0.9]`.

```
    let a : float ref = _0 in
    let f : float -> float = _1 in
```

[2]In C/C++ and many other imperative languages, assignments are possible only at the statement level. Unlike expressions, statements do not have values, so they are ultimately only useful for their side effects. However, we do not need this distinction to be syntactic and indeed an artificial dichotomy between statements and expressions is the source of many difficulties and is avoided in many modern languages.

```
    let g : float -> float = _2 in
    _3
```

Include all of the following parts in your solution:

1. What expression did you choose to fill hole `_0` with?
2. What expression did you choose to fill hole `_1` with?
3. What expression did you choose to fill hole `_2` with?
4. What does the resulting program evaluate to if hole `_3` is filled with `map f (map g [0.7; 0.8; 0.9])`?
5. What does the resulting program evaluate to if hole `_3` is filled with `map (compose f g) [0.7; 0.8; 0.9]`?

Hint: You can use Learn OCaml to check your answer. Make sure you copy in the definition of `map` above, though, since the built-in `map` is written using a left fold rather than a right fold for efficiency. Note that evaluation order is generally right-to-left in OCaml!

2 Imperative ALFA

We will now implement an imperative extension of the ALFA language from previous assignments.[3] Imperative ALFA supports references in much the same way that OCaml does.

2.1 Syntax

We define the structural syntax and the concrete syntax for Imperative ALFA in Fig. 1. Forms that are inherited from ALFA are grayed out to emphasize the new forms, which are the focus of this assignment. The metavariable ℓ ranges over memory locations (in our implementation, we will use integers starting at `0` for memory locations). We will use the concrete syntax in the remainder of the handout.

OCaml datatypes that implement these structures will be provided for you in the exercises. As in the previous assignments, we have organized them into modules, e.g. the type `Typ.t` classifies the OCaml implementations of Imperative ALFA types. As in previous assignments, we write $\lfloor e \rfloor$ to refer to the OCaml implementation of the Imperative ALFA expression e, and similarly for other structures in our syntax. For example,

$$\lfloor c := \underline{3} \rfloor = \texttt{Exp.BinOp(EVar "c", OpAssign, ENumLiteral 3)}$$

In addition, we will provide a parser and pretty-printer that implements a textual syntax that closely follows the concrete syntax shown in Fig. 1. You can inspect the definition of this textual syntax at the following URL:

`https://github.com/eecs490/parsers/tree/master/alfa-mut`

[3]As in Assignment 6, we choose to extend ALFA rather than Recursive ALFA or Gradual ALFA for simplicity and to keep the focus on the new constructs we are introducing.

			Structural Syntax	**Concrete Syntax**
Typ	τ	$::=$	Num	Num
		$\mid$	Bool	Bool
		$\mid$	$\mathsf{Arrow}(\tau,\tau)$	$\tau \to \tau$ (right assoc., precedence 1)
		$\mid$	$\mathsf{Prod}(\tau,\tau)$	$\tau \times \tau$ (right assoc., precedence 3)
		$\mid$	Unit	Unit or on paper, 1
		$\mid$	$\mathsf{Sum}(\tau,\tau)$	$\tau + \tau$ (right assoc., precedence 2)
		$\mid$	$\mathsf{Ref}(\tau)$	τ ref (postfix, precedence 4)
Expr	e, v	$::=$	$\mathsf{NumLit}[n]$	$\underline{n}$
		$\mid$	$\mathsf{Neg}(e)$	$-e$ (prefix, precedence 7)
		$\mid$	$\mathsf{Plus}(e_1, e_2)$	$e_1 + e_2$ (left assoc., precedence 3)
		$\mid$	$\mathsf{Minus}(e_1, e_2)$	$e_1 - e_2$ (left assoc., precedence 3)
		$\mid$	$\mathsf{Times}(e_1, e_2)$	$e_1 * e_2$ (left assoc., precedence 4)
		$\mid$	$\mathsf{Eq}(e_1, e_2)$	$e_1 =? e_2$ (left assoc., precedence 2)
		$\mid$	$\mathsf{Lt}(e_1, e_2)$	$e_1 < e_2$ (left assoc., precedence 2)
		$\mid$	$\mathsf{Gt}(e_1, e_2)$	$e_1 > e_2$ (left assoc., precedence 2)
		$\mid$	True	True
		$\mid$	False	False
		$\mid$	$\mathsf{If}(e_1, e_2, e_3)$	if e_1 then e_2 else e_3 (prefix, precedence 0)
		$\mid$	$\mathsf{Var}[x]$	x
		$\mid$	$\mathsf{Let}(e_1, x.e_2)$	let x be e_1 in e_2 (prefix, precedence 0)
		$\mid$	$\mathsf{Fix}(\tau, x.e)$	fix $(x : \tau) \to e$ (prefix, precedence 0)
		$\mid$	$\mathsf{Fun}(\tau_{\mathsf{in}}, x.e)$	fun $(x : \tau_{\mathsf{in}}) \to e$ (prefix, precedence 0)
		$\mid$	$\mathsf{Ap}(e_1, e_2)$	$e_1\ e_2$ (left assoc., precedence 6)
		$\mid$	$\mathsf{Pair}(e_1, e_2)$	(e_1, e_2)
		$\mid$	Triv	$()$
		$\mid$	$\mathsf{PrjL}(e)$	e.fst (postfix, precedence 8)
		$\mid$	$\mathsf{PrjR}(e)$	e.snd (postfix, precedence 8)
		$\mid$	$\mathsf{LetPair}(e_1, x.y.e_2)$	let (x, y) be e_1 in e_2 (prefix, precedence 0)
		$\mid$	$\mathsf{InjL}(e)$	L e (prefix, precedence 5)
		$\mid$	$\mathsf{InjR}(e)$	R e (prefix, precedence 5)
		$\mid$	$\mathsf{Case}(e, x.e, y.e)$	case e of $\mathsf{L}(x) \to e$ else $\mathsf{R}(y) \to e$ (prefix, precedence 0)
		$\mid$	$\mathsf{Alloc}(e)$	alloc(e)
		$\mid$	$\mathsf{Loc}[\ell]$	$\#\ell$
		$\mid$	$\mathsf{Deref}(e)$	$!e$ (prefix, precedence 8)
		$\mid$	$\mathsf{Assign}(e_1; e_2)$	$e_1 := e_2$ (right assoc., precedence 1)

Figure 1: Syntax of Imperative ALFA

2.2 Dynamic Semantics

For this assignment, we will forgo the typechecker and implement only the dynamic semantics.

To define the dynamic semantics of Imperative ALFA, we need to enrich the evaluation judgement. Rather than simply relating an expression to its value, $e \Downarrow v$, like we have done so far in this course, we instead need a judgement, $e \parallel \mu_{\text{init}} \Downarrow v \parallel \mu_{\text{final}}$, that relates an *initial evaluation state*, $e \parallel \mu$, where e is the expression to be evaluated and μ_{init} is the initial *memory state*, to a final evaluation state, $v \parallel \mu'$.[4]. The idea is that evaluation will cause the computation of a value, v, and also incur side effects on the memory (i.e. fresh allocations and writes, as well as reads from intermediate memory states that arise) during the transition from the initial state to the final state.

We model the memory state, μ, as a finite mapping of memory locations, ℓ, to values:

$$\ell_1 \hookrightarrow v_1, \cdots, \ell_n \hookrightarrow v_n \qquad (n \geq 0)$$

Think of this as a "snapshot" of the heap. This adequately models dynamically allocated memory while leaving the low-level implementation details of finding an available memory location, the specific representation of memory locations (e.g. as 64-bit integers), byte-level memory layout, and freeing memory as an implementation detail. (Our implementation will simply never free memory, but in practice, a garbage collector might be responsible for this. We will talk more about this when we cover Rust in the next assignment.)

Allocation The following rule governs the $\mathsf{alloc}(e)$ construct, which first evaluates the initializer, e, to its value, v, which might itself have side effects, and then allocates a fresh memory location (i.e. one that is not already allocated in the memory) and initializes it with v. The value that results is a reference to the newly allocated location, written $\#\ell$. Locations are the only values of type τ ref. Users of the language will only ever explicitly write $\mathsf{alloc}(e)$ – it does not make sense for the user to explicitly write down a location, since that is an internal implementation detail of the language implementation and operating system.

$$\frac{e \parallel \mu_0 \Downarrow v \parallel \mu_1 \qquad \ell \notin \mathsf{dom}(\mu_1)}{\mathsf{alloc}(e) \parallel \mu_0 \Downarrow \#\ell \parallel \mu_1, \ell \hookrightarrow v} \text{ (E-Alloc)} \qquad \frac{}{\#\ell \ \mathbf{val}} \text{ (V-Loc)}$$

The $\mathsf{alloc}(e)$ operation corresponds to the **`ref`** `e` operation in OCaml described above. (We give it a different name in Imperative ALFA to make it clear that it is an effectful operation.)

Dereference To do a dereference, $!e$, pronounced "bang e", we first evaluate e to a location, $\#\ell$. We then lookup that location in the memory, written $\ell \hookrightarrow v \in \mu_1$ below. The value we find is the value of $!e$. The process of evaluating e to $\#\ell$ may change memory state. However, the act of looking up $\#\ell$ in memory does not change memory state. Note that lookup still counts as a side effect because its behavior is evaluation order-dependent.

$$\frac{e \parallel \mu_0 \Downarrow \#\ell \parallel \mu_1 \qquad \ell \hookrightarrow v \in \mu_1}{!e \parallel \mu_0 \Downarrow v \parallel \mu_1} \text{ (E-Deref)}$$

Assignment Finally, to perform an assignment, $e_1 := e_2$, we begin by evaluating e_1 to a reference to a location, $\#\ell$. We then evaluate e_2 to the value, v_2, that we want to place at that location. The memory state at the end of these evaluations will necessary contain an existing value for ℓ, but

[4] μ is the Greek letter "mu"

we do not care what it is because we are about to replace it, so we simply write _ below. However, we do want to keep the rest of the memory, written μ_2' below, the same (taking advantage of the fact that finite maps are unordered so we can always write a location of interest at the end). The final result of evaluating the assignment expression is the trivial value, (), in the memory state μ_2' updated with the new mapping of ℓ to v_2.

$$\frac{e_1 \parallel \mu_0 \Downarrow \#\ell \parallel \mu_1 \qquad e_2 \parallel \mu_1 \Downarrow v_2 \parallel \mu_2 \qquad \mu_2 = \mu_2', \ell \hookrightarrow \_}{e_1 := e_2 \parallel \mu_0 \Downarrow () \parallel \mu_2', \ell \hookrightarrow v_2} \text{ (E-Assign)}$$

When you implement this, the most straightforward method is to simply define an "update" operation on your representation of memory, similar to how you have previously implemented update for typing contexts.

The remaining rules are very similar to those in ALFA, differing in that we must be explicit about memory states. For example, the following rule states that a value evaluates to itself and memory does not change in the process:

$$\frac{v \ \mathbf{val}}{v \parallel \mu \Downarrow v \parallel \mu} \text{ (E-Val)}$$

The following rule performs negation by recursively evaluating e to a number literal. That evaluation might have a side effect on memory, so we need to allow the output memory state to differ.

$$\frac{e \parallel \mu_0 \Downarrow \underline{n} \parallel \mu_1 \qquad -1 * n = m}{-e \parallel \mu_0 \Downarrow \underline{m} \parallel \mu_1} \text{ (E-Neg)}$$

The rule for plus expressions requires us to specify an evaluation order. Here, we specify left-to-right evaluation order by specifying that the final memory state after evaluating e_1 is the initial memory state used for evaluating e_2. We could specify right-to-left evaluation order by reversing that decision. Without side effects, this choice would be of no major significance and can be left as an implementation decision, but now it is critical to the meaning of the program (as we saw in Exercise 1 above).

$$\frac{e_\ell \parallel \mu_0 \Downarrow \underline{n_\ell} \parallel \mu_1 \qquad e_r \parallel \mu_1 \Downarrow \underline{n_r} \parallel \mu_2 \qquad n_\ell + n_r = n}{e_\ell + e_r \parallel \mu_0 \Downarrow \underline{n} \parallel \mu_2} \text{ (E-Plus)}$$

The rule for let evaluation is given below. Notice that variables are still given meaning by substitution here, and are not stored in the memory!

$$\frac{e_{\mathsf{def}} \parallel \mu_0 \Downarrow v_{\mathsf{def}} \parallel \mu_1 \qquad [v_{\mathsf{def}}/x]e_{\mathsf{body}} \parallel \mu_1 \Downarrow v \parallel \mu_2}{\mathsf{let}\ x : \tau?\ \mathsf{be}\ e_{\mathsf{def}}\ \mathsf{in}\ e_{\mathsf{body}} \parallel \mu_0 \Downarrow v \parallel \mu_2} \text{ (E-Let)}$$

Notice that the variable, x, is not added to the memory. It is a source of substantial confusion in the computing world that variables are often confused for named locations (here, we considered only "heap locations", but many languages also have named stack locations.) Many imperative languages provide only the latter, but call them the former. This practice is at odds with centuries of mathematical practice. Modern languages like OCaml, Haskell, Rust, and others are starting to avoid this confusion: variables are given meaning by substitution, locations are given meaning by read (here, dereference) and write (here, assignment) operations, which operate as side effects, and you can have both variables and locations in a language!

Exercise 2: Derivation (Written, 20%) Provide a derivation, with each rule named, of the following judgement. The expression below allocates a new reference, which we will assume is named ℓ_c, and then updates it. The result is a trivial value and a memory containing a single location as expected.

$$\textsf{let } d \textsf{ be alloc}(\underline{3} - \underline{1}) \textsf{ in } d := \underline{2} * !d \parallel \cdot \ \Downarrow \ () \parallel \ell_d \hookrightarrow \underline{4}$$

Do not use substitution notation, $[e_1/x]e_2$, explicitly in your derivation. Instead, apply substitution inline as necessary. You also do not need to include the rightmost premise in the (E-Alloc), (E-Deref) and (E-Assign) rules. As in the above problem statement, you can abbreviate the notation for extension of an empty memory, e.g. $\cdot, \ell_c \hookrightarrow \underline{10}$ can be written $\ell_c \hookrightarrow \underline{10}$.

You may define sub-derivation abbreviations, named using a D with a subscript of your choice, as necessary to fit the derivation clearly on the page. Do not define any other kinds of abbreviations.

HINT: You will need to apply E-Val multiple times in your derivation. You will only need the rules given above (plus an E-Times rule analagous to E-Plus), plus the (V-NumLit) rule from A2 that just states that number literals are values. Remember that locations are values by rule (V-Loc) above.

2.2.1 Remaining Evaluation Rules

The remaining evaluation rules operate similarly, and are reproduced without commentary below.

Other Binary Operations on Numbers

$$\frac{e_\ell \parallel \mu_0 \Downarrow \underline{n_\ell} \parallel \mu_1 \qquad e_r \parallel \mu_1 \Downarrow \underline{n_r} \parallel \mu_2 \qquad n_\ell - n_r = n}{e_\ell - e_r \parallel \mu_0 \Downarrow \underline{n} \parallel \mu_2} \text{ (E-Minus)}$$

$$\frac{e_\ell \parallel \mu_0 \Downarrow \underline{n_\ell} \parallel \mu_1 \qquad e_r \parallel \mu_1 \Downarrow \underline{n_r} \parallel \mu_2 \qquad n_\ell * n_r = n}{e_\ell * e_r \parallel \mu_0 \Downarrow \underline{n} \parallel \mu_2} \text{ (E-Times)}$$

$$\frac{e_\ell \parallel \mu_0 \Downarrow \underline{n_\ell} \parallel \mu_1 \qquad e_r \parallel \mu_1 \Downarrow \underline{n_r} \parallel \mu_2 \qquad n_\ell < n_r}{e_\ell < e_r \parallel \mu_0 \Downarrow \textsf{True} \parallel \mu_2} \text{ (E-Lt-T)}$$

$$\frac{e_\ell \parallel \mu_0 \Downarrow \underline{n_\ell} \parallel \mu_1 \qquad e_r \parallel \mu_1 \Downarrow \underline{n_r} \parallel \mu_2 \qquad n_\ell \geq n_r}{e_\ell < e_r \parallel \mu_0 \Downarrow \textsf{False} \parallel \mu_2} \text{ (E-Lt-F)}$$

$$\frac{e_\ell \parallel \mu_0 \Downarrow \underline{n_\ell} \parallel \mu_1 \qquad e_r \parallel \mu_1 \Downarrow \underline{n_r} \parallel \mu_2 \qquad n_\ell > n_r}{e_\ell > e_r \parallel \mu_0 \Downarrow \textsf{True} \parallel \mu_2} \text{ (E-Gt-T)}$$

$$\frac{e_\ell \parallel \mu_0 \Downarrow \underline{n_\ell} \parallel \mu_1 \qquad e_r \parallel \mu_1 \Downarrow \underline{n_r} \parallel \mu_2 \qquad n_\ell \leq n_r}{e_\ell > e_r \parallel \mu_0 \Downarrow \textsf{False} \parallel \mu_2} \text{ (E-Gt-F)}$$

$$\frac{e_\ell \parallel \mu_0 \Downarrow \underline{n} \parallel \mu_1 \qquad e_r \parallel \mu_1 \Downarrow \underline{n} \parallel \mu_2}{e_\ell =? e_r \parallel \mu_0 \Downarrow \textsf{True} \parallel \mu_2} \text{ (E-Eq-T)}$$

$$\frac{e_\ell \parallel \mu_0 \Downarrow \underline{n_\ell} \parallel \mu_1 \qquad e_r \parallel \mu_1 \Downarrow \underline{n_r} \parallel \mu_2 \qquad n_\ell \neq n_r}{e_\ell =? e_r \parallel \mu_0 \Downarrow \textsf{False} \parallel \mu_2} \text{ (E-Eq-F)}$$

Function Application

$$\frac{e_{\mathsf{fun}} \parallel \mu_0 \Downarrow \mathsf{fun}\ (x:\tau) \to e \parallel \mu_1 \qquad e_{\mathsf{arg}} \parallel \mu_1 \Downarrow v_{\mathsf{arg}} \parallel \mu_2 \qquad [v_{\mathsf{arg}}/x]e \parallel \mu_2 \Downarrow v \parallel \mu_3}{e_{\mathsf{fun}}\ e_{\mathsf{arg}} \parallel \mu_0 \Downarrow v \parallel \mu_3}\ \text{(E-Ap)}$$

Fixpoints

$$\frac{[\mathsf{fix}\ (x:\tau?) \to e/x]e \parallel \mu_0 \Downarrow v \parallel \mu_1}{\mathsf{fix}\ (x:\tau?) \to e \parallel \mu_0 \Downarrow v \parallel \mu_1}\ \text{(E-Fix)}$$

Products

$$\frac{e_\ell \parallel \mu_0 \Downarrow v_\ell \parallel \mu_1 \qquad e_r \parallel \mu_1 \Downarrow v_r \parallel \mu_2}{(e_\ell, e_r) \parallel \mu_0 \Downarrow (v_\ell, v_r) \parallel \mu_2}\ \text{(E-Pair)}$$

$$\frac{e \parallel \mu_0 \Downarrow (v_\ell, v_r) \parallel \mu_1}{e.\mathsf{fst} \parallel \mu_0 \Downarrow v_\ell \parallel \mu_1}\ \text{(E-PrjL)} \qquad \frac{e \parallel \mu_0 \Downarrow (v_\ell, v_r) \parallel \mu_1}{e.\mathsf{snd} \parallel \mu_0 \Downarrow v_r \parallel \mu_1}\ \text{(E-PrjR)}$$

$$\frac{e_{\mathsf{def}} \parallel \mu_0 \Downarrow (v_\ell, v_r) \parallel \mu_1 \qquad [v_\ell/x][v_r/y]e_{\mathsf{body}} \parallel \mu_1 \Downarrow v \parallel \mu_2}{\mathsf{let}\ (x, y)\ \mathsf{be}\ e_{\mathsf{def}}\ \mathsf{in}\ e_{\mathsf{body}} \parallel \mu_0 \Downarrow v \parallel \mu_2}\ \text{(E-LetPair)}$$

Sums

$$\frac{e_{\mathsf{cond}} \parallel \mu_0 \Downarrow \mathsf{True} \parallel \mu_1 \qquad e_{\mathsf{then}} \parallel \mu_1 \Downarrow v \parallel \mu_2}{\mathsf{if}\ e_{\mathsf{cond}}\ \mathsf{then}\ e_{\mathsf{then}}\ \mathsf{else}\ e_{\mathsf{else}} \parallel \mu_0 \Downarrow v \parallel \mu_2}\ \text{(E-If-T)}$$

$$\frac{e_{\mathsf{cond}} \parallel \mu_0 \Downarrow \mathsf{False} \parallel \mu_1 \qquad e_{\mathsf{else}} \parallel \mu_1 \Downarrow v \parallel \mu_2}{\mathsf{if}\ e_{\mathsf{cond}}\ \mathsf{then}\ e_{\mathsf{then}}\ \mathsf{else}\ e_{\mathsf{else}} \parallel \mu_0 \Downarrow v \parallel \mu_2}\ \text{(E-If-F)}$$

$$\frac{e \parallel \mu_0 \Downarrow v \parallel \mu_1}{\mathsf{L}\ e \parallel \mu_0 \Downarrow \mathsf{L}\ v \parallel \mu_1}\ \text{(E-InjL)} \qquad \frac{e \parallel \mu_0 \Downarrow v \parallel \mu_1}{\mathsf{R}\ e \parallel \mu_0 \Downarrow \mathsf{R}\ v \parallel \mu_1}\ \text{(E-InjR)}$$

$$\frac{e \parallel \mu_0 \Downarrow \mathsf{L}\ v_\ell \parallel \mu_1 \qquad [v_\ell/x]e_\ell \parallel \mu_1 \Downarrow v \parallel \mu_2}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{L}(x) \to e_\ell\ \mathsf{else}\ \mathsf{R}(y) \to e_r \parallel \mu_0 \Downarrow v \parallel \mu_2}\ \text{(E-CaseL)}$$

$$\frac{e \parallel \mu_0 \Downarrow \mathsf{R}\ v_r \parallel \mu_1 \qquad [v_r/y]e_r \parallel \mu_1 \Downarrow v \parallel \mu_2}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{L}(x) \to e_\ell\ \mathsf{else}\ \mathsf{R}(y) \to e_r \parallel \mu_0 \Downarrow v \parallel \mu_2}\ \text{(E-CaseR)}$$

2.3 Implementation

There are two ways to implement Imperative ALFA in OCaml, differing primarily in how memory is represented.

1. If we represent memory states using a pure (i.e. immutable) data structure, i.e. as snapshots, then we can implement an evaluator for Imperative ALFA as a pure function that takes an initial evaluation state (an expression together with a memory state) and returns the final evaluation state (a value together with the final memory state). In other words, we can follow the rules above very directly.

2. If instead we represent memory states in OCaml using a mutable data structure, then we can implement an evaluator for Imperative ALFA as a function with side effects (a.k.a. a procedure) that takes an expression and a mutable data structure operating as the memory and returns only the final value. As a side effect, evaluation will cause the mutable memory to be changed consistent with the semantics above. In other words, the evaluator will actually perform the side effects in the program being evaluated.

You will be implementing both of these.

Exercise 3: Imperative ALFA Evaluator, Functionally (Learn-OCaml, 20%) We will begin with the pure functional implementation.

To do so, you will need to implement an immutable representation of memory states, μ, with a type `Mem.t` and its associated operations

```
module Loc.t = struct
  (* memory locations *)
  type t = int
end
module Mem = struct
  type t = Value.t list
  let extend : t -> Value.t -> Loc.t * t
  let get : t -> Loc.t -> Value.t
  let set : t -> Loc.t * Value.t -> t
end
```

Due to autograder limitations, we require that you implement the operations on `Mem.t` such that `get mem n` returns the `n`th element of `mem`. You might want to start by implementing the `eval` function specified below first, and add new functionality to the `Mem` module only as needed based on a close reading of the rules we defined above.

You will also need to update the implementation of substitution.

Finally, you should implement a function `eval : Exp.t -> Mem.t -> Value.t * Mem.t` that correctly implements the evaluation semantics above assuming it is given a closed, well-typed expression. In other words, the following correctness theorem should be true:

Theorem 1 (Functional Evaluator Correctness)**.** `eval` $\lfloor e \rfloor$ $\lfloor \mu \rfloor$ $\equiv$ `(`$\lfloor v \rfloor$`,` $\lfloor \mu' \rfloor$`)` *iff* $e \parallel \mu \Downarrow v \parallel \mu'$ *and e is well-typed and mentions only locations in μ (details of typing rules omitted).*

You can raise an `IllTyped` exception if the expression contains a location that is not in the given memory.

Careful: your evaluator can fail to terminate if the given expression has unfounded recursion. Don't write any tests that cause this to happen.

Exercise 4: Imperative ALFA Evaluator, Imperatively (Learn-OCaml, 20%) Next, we will implement an Imperative ALFA evaluator imperatively.

To do so, you will need to implement a *mutable* representation of memory states, μ, with a type `MutMem.t`. We will use a mutable list (similar to what was described in lecture, but where the cell values can be mutated directly) to represent memory.

```
type 'a mutlist = 'a cell ref
and 'a cell =
| MNil
| MCons of 'a * 'a mutlist
module MutMem = struct
  type t = Value.t mutlist
  let extend : t -> Value.t -> Loc.t
  let get : t -> Loc.t -> Value.t
  let set : t -> Loc.t * Value.t -> unit
end
```

As with Exercise 3, you should implement `MutMem` such that `MutMem.get mem n` returns the `n`th element of `mem`.

You should be able to use your mutable memory representation to implement a procedure `eval : Exp.t -> MutMem.t -> Value.t` that correctly implements the evaluation semantics above assuming it is given a closed, well-typed expression. Notice that unlike in the previous problem, the function does not return a memory. Instead, it will have side effects on the given memory. You can raise an `IllTyped` exception if the expression contains a location that is not in the memory when the location is encountered during evaluation.

To specify the correctness of your implementation, we need to give both a functional correctness property and an imperative correctness property.

The functional correctness property specifies only that evaluation returns the correct value, saying nothing about how the memory is affected. We assume some appropriate initial memory.

Theorem 2 (Imperative Evaluator Functional Correctness)**.** $e \parallel \mu \Downarrow v \parallel \mu'$ *iff* `eval` $\lfloor e \rfloor$ `mem` $\equiv \lfloor v \rfloor$ *assuming e is well-typed (details of typing rules omitted) and for some* `mem : MutMem.t`*, below.*

The imperative correctness theorem ensures that the function has the correct side effect on memory. To state it most simply, we rely on the `MutMem.snapshot` function that you will define, which takes an immutable snapshot of the current state of the memory. We have partially specified this property below:

Theorem 3 (Imperative Evaluator Imperative Correctness)**.** $e \parallel \mu \Downarrow v \parallel \mu'$ *iff assuming* `mem : MutMem.t` *and given the following pre-condition:*

`MutMem.snapshot mem` $\equiv \lfloor \mu \rfloor$

evaluating the following expression:

`eval` $\lfloor e \rfloor$ `mem`

ensures the following post-condition:

???

Exercise 5: Imperative Correctness (Written, 5%) What is the (strongest) post-condition in the theorem above?

3 Encapsulated Commands

OCaml and Imperative ALFA, like many programming languages, allow us to freely mix pure and effectful expressions, without the type system distinguishing between the two.

So, for example, while we know a function `f : int -> int` in OCaml can be applied to an integer argument and always returns an integer value, the type tells us nothing about the side effects that `f` might perform in the process of determining that return value. It might print something to the console, or write to a mutable reference that was in scope when it was defined, or do file I/O.

While this provides flexibility, it also makes it necessary to reason defensively: *what if a function whose implementation I do not know, say because it is a higher order argument, has side effects*? We saw in Exercise 1 that many optimizations, like map fusion, cannot be applied unless we know for certain that the functions involved are pure (or that their effects commute, more generally).

The possibility of side effects also increases the documentation burden on programmers. One must document not just the relationship between the input and output values of a function, but also what side effects are performed, perhaps with informal documentation in a comment. Exercise 5 above touched on a more formal approach to documenting side effects, and we will investigate the problem of reasoning about programs with side effects more substantially in A8.

We do, ultimately, want our programs to have side effects on the world, so is this state of affairs, where we have to document and reason through possible side effects everywhere throughout our programs, simply inevitable? Is pure functional programming a mere academic curiosity, unable to contend with the reality of a mutable world!?

As it turns out, the answer is *no*: it is possible to syntactically segregate pure expressions from effectful *commands* within a program, with only a bit of bookkeeping. The benefit of doing so is that it is only when writing the latter that we need to take on the full burden of reasoning about side effects. If we then restrict ourselves to the former as much as possible, we can enjoy the simpler and richer reasoning principles of pure expressions in much of our code, turning to effectful commands only "at the edges" of our program that interact with the outside world.

Consider an imperative function `cookies : int -> unit` that, when given an integer n, *issues commands that cause a robot chef to bake n cookies.* This is necessarily an effectful function.

However, we could end up in the same place in two steps, the first pure, the second effectful. First, we define a pure function, `cookie_commands : int -> io`, that, when given an integer n, returns a *baking recipe*, of some suitable type `io`, that encodes as pure data the input/output commands that a robot chef would need to execute to bake n cookies. Though the return value describes effectful commands, it does not actually execute those commands, so it remains pure. We call this pure description of an effectful command an *encapsulated command.* Think of it like a *cookie recipe* – it is just data!

We could then execute those commands by passing them to a procedure, `exec : io -> unit`, that reads through the input and performs the necessary side effects to cause our robot to act. The composition of `cookie_commands` and `exec` leaves us in the same final state as `cookies`, but only when writing `exec`, which is likely to be quite simple, did we have to reason about side effects.

Since the cookie recipe itself is pure, we have more flexibility than simply passing it immediately to `exec`, however. We can cache it to disk, or send it over the network, or broadcast it to our entire swarm of robot chefs, or analyze it to see if it is doing anything unsafe, or combine it with other recipes, e.g. by sequencing them, before passing the result onto `exec`. We could also define alternative executors, e.g. one that runs the command in a simulator rather than on an actual robot.

We can architect software in this way in essentially any language, and indeed this is becoming quite commonplace (e.g. web programming using React, or machine learning using Tensorflow or other staged computation systems). We will investigate this idea in OCaml below.

In Haskell (as well as Elm, Purescript, and others), this distinction is enforced: expressions must be pure, but a program can evaluate to an encapsulated command which is executed by a separate system, distinct from the evaluator. This allows Haskell implementations to be quite aggressive in, e.g., the optimizations it performs, because every program is a pure expression. Reasoning about Haskell programs, once one is experienced with its syntax and semantics, is also simplified because functions are always pure. They might return encapsulated commands for later execution (which *is* reflected in the type), but they do not themselves have side effects.

3.1 Encapsulated Commands in OCaml

Let us now consider exactly how to to encode an encapsulated command in OCaml. Rather than defining an `io` type where the effects being described are commands for a robot to execute, let us simplify it and consider only commands that involve *printing* and *reading* from the console.

Note that reading from a console is an interesting command because while it has a side effect, it also returns a value that we might need to know in order to determine what command should be issued next. For this reason, we actually need to generalize to a parameterized datatype, `t io`, defined below, which classifies encapsulated commands that return values of type `t` when executed.

```
type _ io =
| Print : string -> unit io
| Read  : string io
| Return : 'a -> 'a io
| Bind : 'a io * ('a -> 'b io) -> 'b io
```

Since each constructor differs in what type of value is returned, we used a feature of OCaml that we haven't considered before: *generalized algebraic datatypes* (GADTs), also sometimes called *indexed types*. This allows us to specify, as a type annotation, not just the argument type of each constructor (which we previously did using **`of`**), but also the return type for each constructor (which must be of the form `t io` for some `t`).

The first constructor, `Print`, encapsulates a command that prints the given string and returns a **`unit`** value. The second constructor, `Read`, encapsulates a command that reads from the console, returning a `string` value. The third command, `Return`, sometimes called `Pure`, is a degenerate command in that it has no side effects but returns the given value (this turns out to be useful in some situations).

With only these three constructors, we could only describe a command that does exactly one of these three things. However, in general we might want to sequence multiple commands. The complication is that the next command in a sequence might depend on the value returned by the previous command, e.g. we may want to decide what to print based on the input that was read from the console. This dependent sequencing operation is captured by the `Bind` constructor, which takes the first command in the sequence, of type `'a io`, and a function that returns the next command in the sequence, of type `'b io`, given the value returned from the first command, of type `'a`. Since the sequence ends in this command, the sequence itself is of type `'b io` as well. We will not actually apply the function to generate the next command, however, until we execute the first command. However, the logic for doing so is ready to go!

Let us see how this works with an example of an encapsulated command that asks the user for their name, then prints a greeting or asks again if the empty string was provided.

```
let rec hello : unit -> unit io =
  fun () -> Bind(
    Print "What is your name? ",
    fun () -> Bind(
      Read,
      fun name ->
        if String.equal name "" then
          hello ()
        else
          Print ("Hello, " ^ name ^ "!")))
```

The recursive function `hello` generates a command when passed in a unit argument. No side effects are performed when it is called: `hello` is a *pure* function! Nothing is printed or read from the console when evaluating `hello ()`. It simply returns an ordinary value of type **`unit`** `io`, i.e. an encapsulated command.

How would we actually cause the side effects to occur? By passing this encapsulated command to an *executor*, which in OCaml we can write ourselves as an effectful function, `exec`, which performs the side effects demanded by the command and outputs the value returned by the command. (We must use **`type`** `a.` to explicitly quantify over types in the signature of `exec` because we are working with a GADT, which complicates OCaml type inference, but otherwise this is also ordinary OCaml.)

```
let rec exec : type a. a io -> a =
  fun cmd ->
    match cmd with
    | Print s -> print_endline s
    | Read -> read_line ()
    | Return x -> x
    | Bind (cmd1, next_cmd) ->
      let v1 = exec cmd1 in
      exec (next_cmd v1)
```

When we write `exec (hello ())`, the system will actually execute the command, performing the necessary side effects and returning the unit value once the program finishes executing. The recursion in `hello` allows us to keep producing new commands as needed until the stop condition is triggered.

3.2 Syntactic Sugar

Although we were able to express `hello` using the definitions above, it was not very convenient to write `hello` using all of those calls to `Bind`. Since this is such a common pattern (in fact, it is an instance of a more general algebraic structure called a *monad*), OCaml has special support for simplifying code like this: it allows us to define custom **`let`** forms, distinguished by a suffix like **`let`**`*` (or **`let`**`+` etc.). These are desugared as follows: **`let`**`* x = e1` **`in`** `e2` desugars to `(`**`let`**`*) e1 (`**`fun`** `x -> e2)`, where `(`**`let`**`*)` is simply a special variable name.

In this case, if we define that variable as follows:

```
let (let*) cmd1 next_cmd = Bind (cmd1, next_cmd)
```

then we can rewrite the `hello` function above as follows:

```
let rec hello : unit -> unit io = fun () ->
  let* _ = Print "What is your name? " in
  let* name = Read in
  if String.equal name "" then
    hello ()
  else
    Print ("Hello, " ^ name ^ "!")
```

As you can see, it's starting to look quite like an effectful function, while remaining pure! It may help to manually desugar this definition of `hello` to see that it is indeed equivalent to the first.

(Haskell also has similar syntactic sugar, called **`do`** notation.)

3.3 Encapsulating Mutable References

For any collection of effectful operations, it is possible to define a similar `io` type. For example, let us consider the effectful operations that allow us to work with mutable references (allocation, dereferencing, and assignment):

```
type _ io =
| Alloc : 'a -> 'a ref io
| Deref : 'a ref -> 'a io
| Assign : 'a ref * 'a -> unit io
| Return : 'a -> 'a io
| Bind : 'a io * ('a -> 'b io) -> 'b io

let (let*) cmd1 next_cmd = Bind (cmd1, next_cmd)
```

We can then define our executor as follows:

```
let rec exec : type a. a io -> a =
  fun cmd ->
    match cmd with
    | Alloc v -> ref v
    | Deref x -> !x
    | Assign (r, v) -> r := v
    | Return v -> v
    | Bind (cmd1, next_cmd) ->
      let v1 = exec cmd1 in
      exec (next_cmd v1)
```

Exercise 6: Encapsulated Mutable Map (Learn OCaml, 15%) Using the definition of `_ io` above, write an encapsulated version of the left-to-right `mutmap` function from Lecture 17. Applying it should not have any side effects, i.e. `mutmap_io xs f` should not mutate `xs` nor even dereference it directly. However, `exec (mutmap_io xs f)` should behave identically to the `mutmap` function from lecture.

```
type 'a mutlist = 'a cell ref
and 'a cell =
| MNil
| MCons of 'a * 'a mutlist
```

Part 1. Use the `let*` syntactic sugar defined above, rather than explicit calls to `Bind`.

```
let rec mutmap_io (xs : 'a mutlist) (f : 'a -> 'a) : unit io =
  ??
```

Part 2. Convert your definition to one with explicit calls to `Bind`.

```
let rec mutmap_io_bind (xs : 'a mutlist) (f : 'a -> 'a) : unit io =
  ??
```