Mini-Project 3: Type Inference

CAS CS 320: Principles of Programming Languages

Due December 10, 2024 by 11:59PM

In this project, you'll be building *yet another* interpreter for a subset of OCaml. It'll be your task to implement the following functions (these are the only functions we will be testing):

- ▷ `val unify : ty -> constr list -> ty_scheme option`
- ▷ `val type_of : stc_env -> expr -> ty_scheme option`
- ▷ `val eval_expr : dyn_env -> expr -> value`

This signature appears (in part) in the file `interp03/lib/lib.mli`. The types used in the above signature appear in the module `Utils`. Your implementation of these functions should appear in the file `interp03/lib/lib.ml`. **Please read the following instructions completely and carefully.**

Part 0: Parsing

For this project, **you will be given the parser for the language**. That said, you should still read through and grok what's going on in the grammar so that you know how to write test cases. It's very simliar to the grammar used in mini-project 2 (and we expect that you could have written it yourself) except that there are a few more constructs, and type annotations are optional.

A program in our language is given by the grammar in Figure 1. We present the operators and their associativity in order of increasing precedence in Figure 2. Note also that the function type arrow is right-associative.

Finally, note that there is a desugaring process going on within the parser, so the targeted type `expr` does not match the grammar exactly.[1] Please make sure you understand how programs in the language correspond to expressions in `expr`, and how the given code for evaluating programs depends on your code for evaluating expressions.

Part 1: Type Inference

We will be using a constraint-based inference system to describe the type inference procedure for our language. We write $\Gamma \vdash e : \tau \dashv \mathcal{C}$ to means that e (`expr`) has type τ (`ty`) in the context Γ (`stc_env` $=$ `ty_scheme env`) relative to the set of constraints $\mathcal{C}$ (`const list`). See the file `lib/utils/utils.ml` for more details.

[1] We separated the desugaring step in mini-project 2 primarily for pedagogical purposes.

```
<prog>    ::= {<toplet>}
<toplet>  ::= let [rec] <var> {<arg>}[<annot>] = <expr>
<annot>   ::= : <ty>
<arg>     ::= <var> | ( <var> <annot> )
<ty>      ::= unit | int | float | bool | <ty> list | <ty> option | <tyvar>
            | <ty> * <ty>  | <ty> -> <ty>  | ( <ty> )
<expr>    ::= let [rec] <var> {<arg>} [<annot>] = <expr> in <expr>
            | if <expr> then <expr> else <expr>
            | fun <arg> {<arg>} -> <expr>
            | match <expr> with | <var> , <var> -> <expr>
            | match <expr> with | Some <var> -> <expr> | None -> <expr>
            | match <expr> with | <var> :: <var> -> <expr> | [] -> <expr>
            | <expr2>
<expr2>   ::= <expr2> <bop> <expr2>
            | assert <expr3> | Some <expr3>
            | <expr3> {<expr3>}
<expr3>   ::= () | true | false | [] | None | [ <expr> {; <expr>} ]
            | <int> | <float> | <var>
            | ( <expr> [<annot>] )
<bop>     ::= + | - | * | / | mod | +. | -. | *. | /. | **
            | < | <= | > | >= | = | <> | && | ||
            | , | :: | @
<int>     ::= handled by lexer
<float>   ::= handled by lexer
<var>     ::= handler by lexer
<tyvar>   ::= handler by lexer
```

Figure 1: The grammar for our language

Operators	Associativity
,	*n/a*
||	right
&&	right
<, <=, >, >=, =, <>	left
@	right
::	right
+, -, +., -.	left
*, /, mod, *., /.	left
**	left
function application	left

Figure 2: The operators (and their associativity) of our language in order of increasing precedence

Literals

$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{unit} \dashv \emptyset}\ (\textsf{unit}) \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool} \dashv \emptyset}\ (\textsf{true}) \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool} \dashv \emptyset}\ (\textsf{false})$$

$$\frac{\text{n is an integer literal}}{\Gamma \vdash n : \texttt{int} \dashv \emptyset}\ (\textsf{int}) \qquad \frac{\text{n is a floating-point literal}}{\Gamma \vdash n : \texttt{float} \dashv \emptyset}\ (\textsf{float})$$

Options

$$\frac{\alpha \text{ is fresh}}{\Gamma \vdash \texttt{None} : \alpha\ \texttt{option} \dashv \emptyset}\ (\textsf{none}) \qquad \frac{\Gamma \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \texttt{Some}\ e : \tau\ \texttt{option} \dashv \mathcal{C}}\ (\textsf{some})$$

Here (and below), *fresh* means not appearing anywhere in the derivation. You should use `gensym` to create fresh variables (as in the demo during lecture).

$$\frac{\Gamma \vdash e : \tau \dashv \mathcal{C} \qquad \alpha \text{ is fresh} \qquad \Gamma, x : \alpha \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ \texttt{Some}\ x\ \texttt{->}\ e_1\ \texttt{|}\ \texttt{None}\ \texttt{->}\ e_2 : \tau_2 \dashv \tau \doteq \alpha\ \texttt{option}, \tau_1 \doteq \tau_2, \mathcal{C}, \mathcal{C}_1, \mathcal{C}_2}\ (\textsf{matchOpt})$$

Note that this isn't really pattern matching. We're not using any notion of a pattern, but instead defining a shallow "destructor".

Lists

$$\frac{\alpha \text{ is fresh}}{\Gamma \vdash \texttt{[]} : \alpha\ \texttt{list} \dashv \emptyset}\ (\textsf{nil}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1\ \texttt{::}\ e_2 : \tau_1\ \texttt{list} \dashv \tau_2 \doteq \tau_1\ \texttt{list}, \mathcal{C}_1, \mathcal{C}_2}\ (\textsf{cons})$$

$$\frac{\Gamma \vdash e : \tau \dashv \mathcal{C} \qquad \alpha \text{ is fresh} \qquad \Gamma, h : \alpha, t : \alpha\ \texttt{list} \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ h\ \texttt{::}\ t\ \texttt{->}\ e_1\ \texttt{|}\ \texttt{[]}\ \texttt{->}\ e_2 : \tau_2 \dashv \tau \doteq \alpha\ \texttt{list}, \tau_1 \doteq \tau_2, \mathcal{C}, \mathcal{C}_1, \mathcal{C}_2}\ (\textsf{matchList})$$

Pairs

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1\ \texttt{,}\ e_2 : \tau_1\ \texttt{*}\ \tau_2 \dashv \mathcal{C}_1, \mathcal{C}_2}\ (\textsf{pair})$$

$$\frac{\Gamma \vdash e : \tau \dashv \mathcal{C} \qquad \alpha, \beta \text{ are fresh} \qquad \Gamma, x : \alpha, y : \beta \vdash e' : \tau' \dashv \mathcal{C}'}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ x\ \texttt{,}\ y\ \texttt{->}\ e' : \tau' \dashv \tau \doteq \alpha\ \texttt{*}\ \beta, \mathcal{C}, \mathcal{C}'}\ (\textsf{matchPair})$$

Variables

$$\frac{(x : \forall \alpha_1.\alpha_2 \ldots \alpha_k.\tau) \in \Gamma \qquad \beta_1, \beta_2, \ldots, \beta_k \text{ are fresh}}{\Gamma \vdash x : [\beta_1/\alpha_1][\beta_2/\alpha_2] \ldots [\beta_k/\alpha_k]\tau \dashv \emptyset}\ (\textsf{var})$$

Note that this rule will require implementing substitution on types.

Annotations

$$\frac{\Gamma \vdash e : \tau' \dashv \mathcal{C}}{\Gamma \vdash \texttt{(}\ e\ \texttt{:}\ \tau\ \texttt{)} : \tau \dashv \tau \doteq \tau', \mathcal{C}}\ (\textsf{annot})$$

Assertions

$$\frac{\alpha \text{ is fresh}}{\Gamma \vdash \texttt{assert false} : \alpha \dashv \emptyset} \text{ (assertFalse)} \qquad \frac{\Gamma \vdash e : \tau \dashv \mathcal{C} \qquad e \neq \texttt{false}}{\Gamma \vdash \texttt{assert } e : \texttt{unit} \vdash \tau \doteq \texttt{bool}, \mathcal{C}} \text{ (assert)}$$

Operators

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int} \dashv \tau_1 \doteq \texttt{int}, \tau_2 \doteq \texttt{int}, \mathcal{C}_1, \mathcal{C}_2} \text{ (add)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{int} \dashv \tau_1 \doteq \texttt{int}, \tau_2 \doteq \texttt{int}, \mathcal{C}_1, \mathcal{C}_2} \text{ (sub)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ * } e_2 : \texttt{int} \dashv \tau_1 \doteq \texttt{int}, \tau_2 \doteq \texttt{int}, \mathcal{C}_1, \mathcal{C}_2} \text{ (mul)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ / } e_2 : \texttt{int} \dashv \tau_1 \doteq \texttt{int}, \tau_2 \doteq \texttt{int}, \mathcal{C}_1, \mathcal{C}_2} \text{ (div)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ mod } e_2 : \texttt{int} \dashv \tau_1 \doteq \texttt{int}, \tau_2 \doteq \texttt{int}, \mathcal{C}_1, \mathcal{C}_2} \text{ (mod)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ +. } e_2 : \texttt{float} \dashv \tau_1 \doteq \texttt{float}, \tau_2 \doteq \texttt{float}, \mathcal{C}_1, \mathcal{C}_2} \text{ (addFloat)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ -. } e_2 : \texttt{float} \dashv \tau_1 \doteq \texttt{float}, \tau_2 \doteq \texttt{float}, \mathcal{C}_1, \mathcal{C}_2} \text{ (subFloat)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ *. } e_2 : \texttt{float} \dashv \tau_1 \doteq \texttt{float}, \tau_2 \doteq \texttt{float}, \mathcal{C}_1, \mathcal{C}_2} \text{ (mulFloat)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ /. } e_2 : \texttt{float} \dashv \tau_1 \doteq \texttt{float}, \tau_2 \doteq \texttt{float}, \mathcal{C}_1, \mathcal{C}_2} \text{ (divFloat)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ ** } e_2 : \texttt{float} \dashv \tau_1 \doteq \texttt{float}, \tau_2 \doteq \texttt{float}, \mathcal{C}_1, \mathcal{C}_2} \text{ (powFloat)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ < } e_2 : \texttt{bool} \dashv \tau_1 \doteq \tau_2, \mathcal{C}_1, \mathcal{C}_2} \text{ (lt)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ <= } e_2 : \texttt{bool} \dashv \tau_1 \doteq \tau_2, \mathcal{C}_1, \mathcal{C}_2} \text{ (lte)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ > } e_2 : \texttt{bool} \dashv \tau_1 \doteq \tau_2, \mathcal{C}_1, \mathcal{C}_2} \text{ (gt)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ >= } e_2 : \texttt{bool} \dashv \tau_1 \doteq \tau_2, \mathcal{C}_1, \mathcal{C}_2} \text{ (gte)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ = } e_2 : \texttt{bool} \dashv \tau_1 \doteq \tau_2, \mathcal{C}_1, \mathcal{C}_2} \text{ (eq)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ <> } e_2 : \texttt{bool} \dashv \tau_1 \doteq \tau_2, \mathcal{C}_1, \mathcal{C}_2} \text{ (neq)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ \&\& } e_2 : \texttt{bool} \dashv \tau_1 \doteq \texttt{bool}, \tau_2 \doteq \texttt{bool}, \mathcal{C}_1, \mathcal{C}_2} \text{ (and)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 \texttt{ || } e_2 : \texttt{bool} \dashv \tau_1 \doteq \texttt{bool}, \tau_2 \doteq \texttt{bool}, \mathcal{C}_1, \mathcal{C}_2} \text{ (or)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \alpha \text{ is fresh}}{\Gamma \vdash e_1 \texttt{ @ } e_2 : \alpha \texttt{ list} \dashv \tau_1 \doteq \alpha \texttt{ list}, \tau_2 \doteq \alpha \texttt{ list}, \mathcal{C}_1, \mathcal{C}_2} \text{ (concat)}$$

Conditionals

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \Gamma \vdash e_3 : \tau_3 \dashv \mathcal{C}_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau_3 \dashv e_1 \doteq \texttt{bool}, e_2 \doteq e_3, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3}\ (\textsf{if})$$

Functions

$$\frac{\alpha \text{ is fresh} \qquad \Gamma, x : \alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \alpha\ \texttt{->}\ \tau \dashv \mathcal{C}}\ (\textsf{fun}) \qquad \frac{\Gamma, x : \tau \vdash e : \tau' \dashv \mathcal{C}}{\Gamma \vdash \texttt{fun (} x\ \texttt{ : } \tau\ \texttt{) -> } e : \tau\ \texttt{->}\ \tau' \dashv \mathcal{C}}\ (\textsf{funAnnot})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \alpha \text{ is fresh}}{\Gamma \vdash e_1\ e_2 : \alpha \dashv \tau_1 \doteq \tau_2\ \texttt{->}\ \alpha, \mathcal{C}_1, \mathcal{C}_2}\ (\textsf{app})$$

Let-Expressions

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2 \dashv \mathcal{C}_1, \mathcal{C}_2}\ (\textsf{let})$$

$$\frac{\alpha, \beta \text{ is fresh} \qquad \Gamma, f : \alpha\ \texttt{->}\ \beta \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma, f : \tau_1 \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash \texttt{let rec}\ f\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2 \dashv \tau_1 \doteq \alpha\ \texttt{->}\ \beta, \mathcal{C}_1, \mathcal{C}_2}\ (\textsf{letRec})$$

This completes the description of our typing rules. The function `type_of`, given a context Γ and an expression e, should return `Some` τ' where τ' is the *principle type (scheme)* of e in the context Γ. That is, given that $\Gamma \vdash e : \tau \dashv \mathcal{C}$, you must

- ▷ determine the most general unifier $\mathcal{S}$ of the unification problem defined by $\mathcal{C}$;
- ▷ determine the type $\mathcal{S}\tau$, i.e., the type τ after the substitution $\mathcal{S}$;
- ▷ quantify over the free variables of $\mathcal{S}\tau$ to get the principle type τ'.

`type_of` should return `None` if there is no unifier for $\mathcal{C}$.

Part 2: Evaluation

The evaluation of a program in our language is given by the big-step operational semantics presented below. It's identical to that of mini-project 2, with some additional constructs. We write $\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow v$ to indicate that the expression e evaluates to the value v in the dynamic environment $\mathcal{E}$. We use the following notation for environments.

Notation	Description
$\varnothing$	empty environment
$\mathcal{E}[x \mapsto v]$	$\mathcal{E}$ with x mapped to v
$\mathcal{E}(x)$	the value of x in $\mathcal{E}$

We take a *value* to be:

- ▷ unit, denoted as $\bullet$
- ▷ a Boolean value (an element of the set $\mathbb{B}$) denoted as *true* and *false*
- ▷ an integer (an element of the set $\mathbb{Z}$) denoted as 1, -234, 12, etc.

- ▷ a floating-point number (an element of the set $\mathbb{R}$), denote as 1.2, -3.14, etc.
- ▷ a pairs of values, denoted, e.g., as (u, v).
- ▷ a list of values, denoted, e.g., as $v_1 :: v_2 :: \cdots :: v_k :: []$. We write $l@r$ for list concatenation.
- ▷ an option value, denoted None or Some(v)
- ▷ a closure, denoted as $(\!|\, \mathcal{E}\ ,\ s \mapsto \lambda x.e \,|\!)$, where $\mathcal{E}$ is an environment, s is a name (represented as a string), and $\lambda x.e$ is a function. We write $(\!|\, \mathcal{E}\ ,\ \cdot \mapsto \lambda x.e \,|\!)$ for a closure without a name.

The function `eval_expr`, given an expression e (`expr`), should return v (`value`) in the case that $\langle\ \varnothing\ ,\ e\ \rangle \Downarrow v$ is derivable according to the given semantics. There are three cases in which this function may raise an exception.

- ▷ `DivByZero`, the second argument of a division operator was 0 (this includes integer modulus).
- ▷ `AssertFail`, an assertion *within our language* (not an OCaml `assert`) failed.
- ▷ `RecWithoutArg`, the value of a recursive let-expression evaluates to a *named* closure (as opposed to an unnamed closure).[2]
- ▷ `CompareFunVals`, a polymorphic comparison operator (e.g., `=` or `<`) was applied to closures.

Literals

$$\frac{}{\langle\ \mathcal{E}\ ,\ \texttt{()}\ \rangle \Downarrow \bullet}\ \text{(unitEval)} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ \texttt{true}\ \rangle \Downarrow \mathit{true}}\ \text{(trueEval)} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ \texttt{false}\ \rangle \Downarrow \mathit{false}}\ \text{(falseEval)}$$

$$\frac{\text{n is an integer literal}}{\langle\ \mathcal{E}\ ,\ n\ \rangle \Downarrow n}\ \text{(intEval)} \qquad \frac{\text{n is an floating-point literal}}{\langle\ \mathcal{E}\ ,\ n\ \rangle \Downarrow n}\ \text{(floatEval)}$$

Options

$$\frac{}{\langle\ \mathcal{E}\ ,\ \texttt{None}\ \rangle \Downarrow \text{None}}\ \text{(evalNone)} \qquad \frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \texttt{Some}\ e\ \rangle \Downarrow \text{Some}(v)}\ \text{(evalSome)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow \text{Some}(v) \qquad \langle\ \mathcal{E}[x \mapsto v]\ ,\ e_1\ \rangle \Downarrow v_1}{\langle\ \mathcal{E}\ ,\ \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ \texttt{Some}\ x\ \texttt{->}\ e_1\ \texttt{|}\ \texttt{None}\ \texttt{->}\ e_2\ \rangle \Downarrow v_1}\ \text{(evalMatchOptSome)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow \text{None} \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ \texttt{Some}\ x\ \texttt{->}\ e_1\ \texttt{|}\ \texttt{None}\ \texttt{->}\ e_2\ \rangle \Downarrow v_2}\ \text{(evalMatchOptNone)}$$

Lists

$$\frac{}{\langle\ \mathcal{E}\ ,\ \texttt{[]}\ \rangle \Downarrow []}\ \text{(evalNil)} \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{::}\ e_2\ \rangle \Downarrow v_1 :: v_2}\ \text{(evalCons)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow v_h :: v_t \qquad \langle\ \mathcal{E}[h \mapsto v_h][t \mapsto v_t]\ ,\ e_1\ \rangle \Downarrow v_1}{\langle\ \mathcal{E}\ ,\ \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ h\ \texttt{::}\ t\ \texttt{->}\ e_1\ \texttt{|}\ \texttt{[]}\ \texttt{->}\ e_2\ \rangle \Downarrow v_1}\ \text{(evalMatchListCons)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow [] \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ h\ \texttt{::}\ t\ \texttt{->}\ e_1\ \texttt{|}\ \texttt{[]}\ \texttt{->}\ e_2\ \rangle \Downarrow v_2}\ \text{(evalMatchListNil)}$$

[2] We dealt with this in the syntax in mini-project 2.

Pairs

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{,}\ e_2\ \rangle \Downarrow (v_1, v_2)}\ \text{(evalPair)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow (v_1, v_2) \qquad \langle\ \mathcal{E}[x \mapsto v_1][y \mapsto v_2]\ ,\ e'\ \rangle \Downarrow v'}{\langle\ \mathcal{E}\ ,\ \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ x\ \texttt{,}\ y\ \texttt{->}\ e'\ \rangle \Downarrow v'}\ \text{(evalMatchPair)}$$

Variables, Annotations, Assertions

$$\frac{x \text{ is a variable}}{\langle\ \mathcal{E}\ ,\ x\ \rangle \Downarrow \mathcal{E}(x)}\ \text{(varEval)} \qquad \frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \texttt{(}\ e\ \texttt{:}\ \tau\ \texttt{)}\ \rangle \Downarrow v}\ \text{(evalAnnot)} \qquad \frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow \mathit{true}}{\langle\ \mathcal{E}\ ,\ \texttt{assert}\ e\ \rangle \Downarrow \bullet}\ \text{(evalAssert)}$$

Operators

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{+}\ e_2\ \rangle \Downarrow v_1 + v_2}\ \text{(addEval)} \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{-}\ e_2\ \rangle \Downarrow v_1 - v_2}\ \text{(subEval)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{*}\ e_2\ \rangle \Downarrow v_1 \times v_2}\ \text{(mulEval)} \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_2 \neq 0}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{/}\ e_2\ \rangle \Downarrow v_1/v_2}\ \text{(divEval)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_2 \neq 0}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{mod}\ e_2\ \rangle \Downarrow v_1\ \mathit{mod}\ v_2}\ \text{(modEval)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{+.}\ e_2\ \rangle \Downarrow v_1 + v_2}\ \text{(addFEval)} \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{-.}\ e_2\ \rangle \Downarrow v_1 - v_2}\ \text{(subFEval)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{*.}\ e_2\ \rangle \Downarrow v_1 \times v_2}\ \text{(mulFEval)} \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{/.}\ e_2\ \rangle \Downarrow v_1/v_2}\ \text{(divFEval)}$$

Note that there is no division-by-zero error in the case of floating-point numbers.

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{**}\ e_2\ \rangle \Downarrow v_1^{v_2}}\ \text{(powFEval)}$$

To save room, all of the following rules do not include the side condition that v_1 and v_2 cannot be closures.

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 < v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{<}\ e_2\ \rangle \Downarrow \mathit{true}}\ \text{(ltTrue)} \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 \geq v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{<}\ e_2\ \rangle \Downarrow \mathit{false}}\ \text{(ltFalse)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 \leq v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{<=}\ e_2\ \rangle \Downarrow \mathit{true}}\ \text{(lteTrue)} \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 > v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{<=}\ e_2\ \rangle \Downarrow \mathit{false}}\ \text{(lteFalse)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 > v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{>}\ e_2\ \rangle \Downarrow \mathit{true}}\ \text{(gtTrue)} \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 \leq v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{>}\ e_2\ \rangle \Downarrow \mathit{false}}\ \text{(gtFalse)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 \geq v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{>=}\ e_2\ \rangle \Downarrow \mathit{true}}\ (\mathsf{gteTrue}) \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 < v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{>=}\ e_2\ \rangle \Downarrow \mathit{false}}\ (\mathsf{gteFalse})$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 = v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{=}\ e_2\ \rangle \Downarrow \mathit{true}}\ (\mathsf{eqTrue}) \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 \neq v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{=}\ e_2\ \rangle \Downarrow \mathit{false}}\ (\mathsf{eqFalse})$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 \neq v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{<>}\ e_2\ \rangle \Downarrow \mathit{true}}\ (\mathsf{neqTrue}) \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 = v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{<>}\ e_2\ \rangle \Downarrow \mathit{false}}\ (\mathsf{neqFalse})$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow \mathit{false}}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{\&\&}\ e_2\ \rangle \Downarrow \mathit{false}}\ (\mathsf{andFalse}) \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow \mathit{true} \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{\&\&}\ e_2\ \rangle \Downarrow v_2}\ (\mathsf{andTrue})$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow \mathit{true}}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{||}\ e_2\ \rangle \Downarrow \mathit{true}}\ (\mathsf{orTrue}) \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow \mathit{false} \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{||}\ e_2\ \rangle \Downarrow v_2}\ (\mathsf{orFalse})$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{@}\ e_2\ \rangle \Downarrow v_1 @ v_2}\ (\mathsf{concatEval})$$

Conditionals

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow \mathit{true} \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \rangle \Downarrow v}\ (\mathsf{ifTrue}) \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow \mathit{false} \qquad \langle\ \mathcal{E}\ ,\ e_3\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \rangle \Downarrow v}\ (\mathsf{ifFalse})$$

Functions

$$\frac{}{\langle\ \mathcal{E}\ ,\ \texttt{fun}\ x\ \texttt{->}\ e\ \rangle \Downarrow (\!|\ \mathcal{E}\ ,\ \cdot \mapsto \lambda x.e\ |\!)}\ (\mathsf{funEval})$$

$$\frac{}{\langle\ \mathcal{E}\ ,\ \texttt{fun (}\ x\ \texttt{:}\ \tau\ \texttt{)->}\ e\ \rangle \Downarrow (\!|\ \mathcal{E}\ ,\ \cdot \mapsto \lambda x.e\ |\!)}\ (\mathsf{funEvalAnnot})$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow (\!|\ \mathcal{E}'\ ,\ \cdot \mapsto \lambda x.e\ |\!) \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad \langle\ \mathcal{E}'[x \mapsto v_2]\ ,\ e\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ e_1\ e_2\ \rangle \Downarrow v}\ (\mathsf{appEval})$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow (\!|\ \mathcal{E}'\ ,\ s \mapsto \lambda x.e\ |\!) \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad \langle\ \mathcal{E}'[s \mapsto (\!|\ \mathcal{E}'\ ,\ s \mapsto \lambda x.e\ |\!)][x \mapsto v_2]\ ,\ e\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ e_1\ e_2\ \rangle \Downarrow v}\ (\mathsf{appRecEval})$$

Let-Expressions

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}[x \mapsto v_1]\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ \texttt{let}\ x\ \texttt{:}\ \tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \rangle \Downarrow v_2}\ (\mathsf{letEval})$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow (\!|\ \mathcal{E}'\ ,\ \cdot \mapsto \lambda x.e\ |\!) \qquad \langle\ \mathcal{E}[f \mapsto (\!|\ \mathcal{E}'\ ,\ f \mapsto \lambda x.e\ |\!)]\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ \texttt{let rec}\ f\ \texttt{:}\ \tau\ \texttt{->}\ \tau'\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \rangle \Downarrow v_2}\ (\mathsf{letRecEval})$$

If e_1 evaluates to a *named* closure in the above rule, then `eval` should raise a `RecWithoutArg` exception.

Putting Everything Together

After you're done with the required functions, you should be able to run

```
dune exec interp03 filename
```

in order to execute code you've written in other files (replace `filename` with the name of the file which contains code you want to execute). Our language is subset of OCaml so you should be able to easily write programs, e.g., here is an implementation of `sum_of_squares` *without type annotations*:

```
(* sum of squares function *)
let sum_of_squares x y =
  let x_squared = x * x in
  let y_squared = y * y in
  x_squared + y_squared
let _ = assert (sum_of_squares 3 (-5) = 34)
```

There are a large number of examples in the file `examples.ml`. If you're code is correct, you should be able to run this entire file (the inverse is not true, being able to run this file does not guarantee that your code is correct). You can pull out individual test cases as you work through the project.

Final Remarks

- ▷ There is a lot of repetition here, this is just the nature of implementing programming languages. So even though there is a lot of code to write, it should go pretty quickly. Despite this, it may be worthwhile to think about how to implement the interpreter without too much code replication.
- ▷ Test along the way. Don't try to write the whole interpreter and test afterwards.
- ▷ You **must** use exactly the same names and types as given at the top of this file. They **must** appear in the file `interp03/lib/lib.ml`. If you don't do this, we can't grade your submission. You are, of course, allowed to add your own functions and directories.
- ▷ You're given a skeleton dune project to implement your interpreter. **Do not change any of the given code**. In particular, don't change the dune files or the utility files. When we grade your assignment we will be assume this skeleton code.
- ▷ We will not immediately release examples or the autograder. You should test yourself as best as you can first.
- ▷ Even though we've given a lot more starter code this time around, you still need to make sure you understand how the starter code works. This means reading code that you didn't write (which is what you'll spend most of your life doing if you go on to be a software engineer). A word of advice: don't immediately ask on Piazza "what does this code do?" Read it, try it out, and try to come up with a more specific question if you're still confused.

Good luck, happy coding.