Mini-Project 3: Type Inference

CAS CS 320: Principles of Programming Languages

Due May 1, 2025 by 8:00PM

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:

```
val principle_type : ty -> constr list -> ty_scheme option
val type_of : stc_env -> expr -> ty_scheme option
val is_well_typed : prog -> bool
val eval_expr : dyn_env -> expr -> value
```

The types used in the above signature appear in the module `Utils`. Your implementation of these functions should appear in the file `interp3/lib/interp3.ml`. **Please read the following instructions completely and carefully.**

Part 1: Parsing

You'll be given *part of* the parser for the language you'll be implementing. 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. It will be up to you to read through the existing parser and determine what is missing from it, based on the given grammar (*Hint.* the lexer is complete).

Note that there is a desugaring process going on *within* the parser, so the target type `expr` does not match the grammar exactly. 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 2: Type Inference

We'll 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 the expression e (`expr`) has type τ (`ty`) in the context Γ (`stc_env`) relative to the set of constraints $\mathcal{C}$ (`const list`). See the file `lib/utils/utils.ml` for more details on the associated types. What follows is the typing rules of the inference system for our language.

Literals

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

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

```
 <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>
           |   fun <arg> {<arg>} → <expr>
           |   if <expr> then <expr> else <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 | +. | -. | *. | /. | **
           |   < | <= | > | >= | = | <> | && | ||
           |   , | ::
```

Figure 1: The grammar for our language

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

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

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` in the module `Utils` to create fresh variables.

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

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

Lists

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

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

Pairs

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

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

Variables

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

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

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

Operators

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

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

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

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

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

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

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

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

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

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

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

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

Conditionals

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

Functions

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

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

Let-Expressions

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

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

Remarks

The expression `type_of` Γ e should be `Some` τ' where τ' is the *principle type scheme* of the expression 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 scheme τ'. Note that the principle type scheme τ' should be the result of evaluating `principle_type` τ $\mathcal{C}$.

`type_of` Γ e should be `None` if there is no unifier for $\mathcal{C}$. Finally, the expression `is_well_typed` p should be `true` if and only if each individual binding in the collection of top-level let-expressions has a type *relative to the context containing previously defined bindings.* That is, the program

$$\begin{aligned}
&\texttt{let } [\texttt{rec}]\ x_1 \texttt{ = } e_1 \\
&\texttt{let } [\texttt{rec}]\ x_2 \texttt{ = } e_2 \\
&\texttt{let } [\texttt{rec}]\ x_3 \texttt{ = } e_3 \\
&\qquad\quad \vdots \\
&\texttt{let } [\texttt{rec}]\ x_k \texttt{ = } e_k
\end{aligned}$$

is well-typed if

- ▷ e_1 is well-typed with type scheme τ_1 in the empty context $\varnothing$
- ▷ e_2 is well-typed with type scheme τ_2 in the context $\{x_1 : \tau_1\}$
- ▷ e_3 is well-typed with type scheme τ_3 in the context $\{x_1 : \tau_1, x_2 : \tau_2\}$

and so on. The empty program is well-typed.

The last thing we'll mention: type schemes are defined using the `VarSet` module defined in `Utils`. Make sure to take a look at this module for hints on how to use it. It defines a standard set type with operations like `union`, `singleton`, and `to_list`.

Part 3: 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:

- an integer (an element of the set $\mathbb{Z}$) denoted 1, -234, 12, etc.
- a floating-point number (an element of the set $\mathbb{R}$), denote 1.2, -3.14, etc.
- a Boolean value (an element of the set $\mathbb{B}$) denoted $\top$ and $\bot$
- unit, denoted $\bullet$
- a closure, denoted $(\!|\ \mathcal{E}\ ,\ s \mapsto e\ |\!)$, where $\mathcal{E}$ is an environment, s is a name, and e is an expression. We'll write $(\!|\ \mathcal{E}\ ,\ \cdot \mapsto e\ |\!)$ for a closure without a name.
- a pairs of values, denoted (u, v)
- a list of values, denoted $[v_1, v_2, \dots, v_k]$
- an option value, denoted None or $\mathsf{Some}(v)$

The expression `eval_expr` $\mathcal{E}$ e should the value v (`value`) in the case that $\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow v$ 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.
- `CompareFunVals`, a polymorphic comparison operator (e.g., `=` or `<`) was applied to closures.

Literals

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

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

Options

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

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

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

Lists

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

$$\frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow [v_1, v_2, \dots, v_k] \qquad \langle\ \mathcal{E}[h \mapsto v_1][t \mapsto [v_2, \dots, v_k]]\ ,\ e_1\ \rangle \Downarrow v_1}{\langle\ \mathcal{E}\ ,\ \texttt{match}\ e\ \texttt{with |}\ h\ \texttt{::}\ t \rightarrow e_1\ \texttt{| []} \rightarrow e_2\ \rangle \Downarrow v_1}\ (\mathsf{evalMatchListCons})$$

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

Pairs

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

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

Variables, Annotations, Assertions

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

Operators

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

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

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

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

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

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

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

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

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 < v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{<}\ e_2\ \rangle \Downarrow \top}\ (\mathsf{ltTrue}) \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 \geq v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{<}\ e_2\ \rangle \Downarrow \bot}\ (\mathsf{ltFalse})$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 \leq v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{<=}\ e_2\ \rangle \Downarrow \top}\ (\mathsf{lteTrue}) \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 > v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{<=}\ e_2\ \rangle \Downarrow \bot}\ (\mathsf{lteFalse})$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 > v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{>}\ e_2\ \rangle \Downarrow \top}\ (\mathsf{gtTrue}) \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 \leq v_2}{\langle\ \mathcal{E}\ ,\ e_1\ \texttt{>}\ e_2\ \rangle \Downarrow \bot}\ (\mathsf{gtFalse})$$

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

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

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

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

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

Conditionals

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

Functions

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

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

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

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

Let-Expressions

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

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

Putting Everything Together

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

```
dune exec interp3 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 `interp3/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.
- 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.