Mini-Project 2: The Environment Model

CAS CS 320: Principles of Programming Languages

Due November 26, 2024 by 11:59PM

In this project, you'll be building an *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 parse : string -> prog option`
- ▷ `val desugar : prog -> expr`
- ▷ `val type_of : expr -> (ty, error) result`
- ▷ `val eval : expr -> value`
- ▷ `val interp : string -> (value, error) result`

This signature appears in the file `interp02/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 `interp02/lib/lib.ml`. **Please read the following instructions completely and carefully.**

Part 1: Parsing

A program in our language is given by the grammar in Figure 1. The start symbol of this grammar is <prog>. We present the operators and their associativity *in order of increasing precedence* in Figure 2. This table does not include the function type arrow `->` because it's not an operator. **The function type arrow is *right*-associative.**

Your implementation of `parse` should return `None` in the case that the input string is not recognized by this grammar. It should target the type `prog`, which appears in the module `Utils`. It should also be whitespace agnostic. This means that in your lexer, you should make sure to eliminate whitespace between tokens. We will use the following regular expressions to represent whitespace, numbers and variables.

```
let whitespace = [' ' '\t' '\n' '\r']+
let num = '-'? ['0'-'9']+
let var = ['a'-'z' '_'] ['a'-'z' 'A'-'Z' '0'-'9' '_' '\'']*
```

You can used these identifiers without modification in your lexer.

Part 2: Desugaring

You'll notice that, this time around, a program is not the same thing as an expression. The kinds of expressions that we'll eventually type-check and evaluate aren't the same as expressions that we write in our language. In other words, our *concrete* (or *surface-level*) syntax is not the same as our *abstract* syntax.

The process of translating surface-level syntax into abstract syntax is called *desugaring*.[1] The function `desugar` should translate a program of type `prog` to an expression of type `expr` according to the following rules.

[1] You may have heard the term *syntactic sugar* with regards to convenient syntactic constructs in programming languages you've used.

```
<prog>    ::= {<toplet>} EOF
<toplet>  ::= let <var> {<arg>} : <ty> = <expr>
            | let rec <var> <arg> {<arg>} : <ty> = <expr>
<arg>     ::= ( <var> : <ty> )
<ty>      ::= int | bool | unit | <ty> -> <ty> | ( <ty> )
<expr>    ::= let <var> {<arg>} : <ty> = <expr> in <expr>
            | let rec <var> <arg> {<arg>} : <ty> = <expr> in <expr>
            | if <expr> then <expr> else <expr>
            | fun <arg> {<arg>}-> <expr>
            | <expr2>
<expr2>   ::= <expr2> <bop> <expr2>
            | assert <expr3>
            | <expr3> {<expr3>}
<expr3>   ::= () | true | false
            | <num> | <var>
            | ( <expr> )
<bop>     ::= + | - | * | / | mod | < | <= | > | >= | = | <> | && | ||
<num>     ::= handled by lexer
<var>     ::= handler by lexer
```

Figure 1: The grammar for our language

Operators	Associativity
`||`	right
`&&`	right
`<`, `<=`, `>`, `>=`, `=`, `<>`	left
`+`, `-`	left
`*`, `/`, `mod`	left
function application	left

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

▷ A sequence of top-level let-statements is shorthand for collection of nested let-expressions ending in a unit. That is, the surface-level syntax

$$\begin{aligned}&\texttt{let [rec]}\ x_1\ \texttt{\{<arg>\}}\ \texttt{:}\ \tau_1\ \texttt{=}\ e_1\\&\texttt{let [rec]}\ x_2\ \texttt{\{<arg>\}}\ \texttt{:}\ \tau_2\ \texttt{=}\ e_2\\&\ldots\\&\texttt{let [rec]}\ x_k\ \texttt{\{<arg>\}}\ \texttt{:}\ \tau_k\ \texttt{=}\ e_k\end{aligned}$$

is shorthand for

$$\begin{aligned}&\texttt{let [rec]}\ x_1\ \texttt{\{<arg>\}}\ \texttt{:}\ \tau_1\ \texttt{=}\ e_1\ \texttt{in}\\&\texttt{let [rec]}\ x_2\ \texttt{\{<arg>\}}\ \texttt{:}\ \tau_2\ \texttt{=}\ e_2\ \texttt{in}\\&\ldots\\&\texttt{let [rec]}\ x_k\ \texttt{\{<arg>\}}\ \texttt{:}\ \tau_k\ \texttt{=}\ e_k\ \texttt{in}\\&\texttt{()}\end{aligned}$$

▷ A let-expression with arguments is shorthand for a let expression with no arguments whose value is an anonymous function. That is, the surface-level syntax

$$\texttt{let [rec]}\ f\ \texttt{(}\ x_1\ \texttt{:}\ \tau_1\ \texttt{)}\ldots\texttt{(}\ x_k\ \texttt{:}\ \tau_k\ \texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

is shorthand for

$$\texttt{let [rec]}\ f\ \texttt{:}\ \tau_1\ \texttt{->}\ \ldots\ \tau_k\ \texttt{->}\ \tau\ \texttt{=}\ \texttt{fun (}\ x_1\ \texttt{:}\ \tau_1\ \texttt{)}\ldots\texttt{(}\ x_k\ \texttt{:}\ \tau_k\ \texttt{)}\ \texttt{->}\ e_1\ \texttt{in}\ e_2$$

▷ An anonymous function with multiple arguments shorthand for Curry-ed collection of single-argument anonymous functions. That is, the surface-level syntax

$$\texttt{fun (}\ x_1\ \texttt{:}\ \tau_1\ \texttt{)}\ldots\texttt{(}\ x_k\ \texttt{:}\ \tau_k\ \texttt{)}\ \texttt{->}\ e$$

is shorthand for

$$\texttt{fun (}\ x_1\ \texttt{:}\ \tau_1\ \texttt{)}\ \texttt{->}\ \ldots\ \texttt{fun ->}\ \texttt{(}\ x_k\ \texttt{:}\ \tau_k\ \texttt{)}\ \texttt{->}\ e$$

These rules should be applied until there are no more occurrences of let-statements, let-expressions with arguments, or anonymous functions with multiple arguments. Note that the type `expr` does not allow these constructs.

Part 3: Type-Checking

An expression is well-typed if it is derivable according to the following rules. We write $\Gamma \vdash e : \tau$ to mean that e has type τ in the context Γ, where e is an `<expr>` after desurgaring, and τ is a `<ty>`. In practice, you will be implementing a type-*inference* algorithm, but because all function arguments and values are annotated with types, this is only slightly more difficult than type-checking (in mini-project 3, we'll take some of the type-annotations, which will make the type-inference problem more difficult).

The function `type_of`, given an expression e of type `expr`, should return `Ok` τ if there is a context Γ for which $\Gamma \vdash e : \tau$ is derivable. Otherwise, it should return `Error` *err*, where *err* is one of the following constructors of the type `error`.

▷ `UnknownVar` x The variable x is unbound.

▷ `IfTyErr` τ_e τ_a: The else-case of an if-expression was expected to be type τ_e (based on the then-case) but was determined to be type τ_a.

- ▷ `IfCondTyErr` τ_a: The condition of an if-expression was expected to be type `bool` but was determined to be type τ_a.
- ▷ `OpTyErrL` *op* τ_e τ_a: The left argument of an operator *op* was expected to be type τ_e but was determined to be type τ_a.
- ▷ `OpTyErrR` *op* τ_e τ_a: The right argument of an operator *op* was expected to be type τ_e but was determined to be type τ_a.
- ▷ `FunArgTyErr` τ_e τ_a: The argument of a function of type τ_e `->` τ was expected to be type τ_e but was determined to be type τ_a.
- ▷ `FunAppTyErr` τ_a: An expression determined to be type τ_a was applied as a function but τ_a is not a function type.
- ▷ `LetTyErr` τ_e τ_a: The value of a let-expression was expected to be type τ_e (according to its annotation) but was determined to be type τ_a.
- ▷ `AssertTyErr` τ_a: The argument of an assert-expression was expected to be type `bool` but was determined to be type τ_a.

It will be up to you to determine in which cases you should return a given error. If an expression is not well-typed, you should return the error corresponding to the *leftmost-innermost* subexpression which is ill-typed. In practice, this means **processing premises of a typing rules from left to right**. What remains are the typing rules.

Literals

$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}}\ (\textsf{unit}) \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ (\textsf{true}) \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ (\textsf{false}) \qquad \frac{\text{n is an integer literal}}{\Gamma \vdash n : \texttt{int}}\ (\textsf{int})$$

Variables

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ (\textsf{var})$$

Operators

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}\ (\textsf{add}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{int}}\ (\textsf{sub}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ * } e_2 : \texttt{int}}\ (\textsf{mul})$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ / } e_2 : \texttt{int}}\ (\textsf{div}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ mod } e_2 : \texttt{int}}\ (\textsf{mod})$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ < } e_2 : \texttt{bool}}\ (\textsf{lt}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ <= } e_2 : \texttt{bool}}\ (\textsf{lte}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ > } e_2 : \texttt{bool}}\ (\textsf{gt})$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ >= } e_2 : \texttt{bool}}\ (\textsf{gte}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ = } e_2 : \texttt{bool}}\ (\textsf{eq}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ <> } e_2 : \texttt{bool}}\ (\textsf{neq})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{ \&\& } e_2 : \texttt{bool}}\ (\textsf{and}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ || } e_2 : \texttt{bool}}\ (\textsf{or})$$

Conditionals

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\ (\textsf{if})$$

Functions

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun (} x\ \texttt{:}\ \tau_1\ \texttt{) -> } e : \tau_1\ \texttt{->}\ \tau_2}\ (\textsf{fun}) \qquad \frac{\Gamma \vdash e_1 : \tau_2\ \texttt{->}\ \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}\ (\textsf{app})$$

Let-Expressions

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{:}\ \tau_1\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2}\ (\textsf{let}) \qquad \frac{\Gamma, x : \tau_1 \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec}\ \ x\ \texttt{:}\ \tau_1\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2}\ (\textsf{let})$$

Assert

$$\frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{assert}\ e : \texttt{unit}}\ (\textsf{assert})$$

Part 2: Evaluation

The evaluation of a program in our language is given by the big-step operational semantics presented below. We write $\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow v$ to indicate that the expression e evaluates to the value v in the 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 as 1, -234, 12, etc.
- ▷ a Boolean value (an element of the set $\mathbb{B}$) denoted as *true* and *false*
- ▷ unit, denoted as $\bullet$
- ▷ 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 will write $(\!|\ \mathcal{E}\ ,\ \cdot \mapsto \lambda x.e\ |\!)$ for a closure without a name.

The function `eval`, given an expression e of type `expr`, should return v in the case that $\langle\ \varnothing\ ,\ e\ \rangle \Downarrow v$ is derivable according to the given semantics. There are two cases in which this function may raise one of the two exceptions.

- ▷ `DivByZero`, the second argument of a division operator was 0
- ▷ `AssertFail`, an assertion *within our language* (not an OCaml `assert`) failed

It will be up to you to determine when to raise these exceptions

Literals

$$\frac{\text{n is an integer literal}}{\langle\ \mathcal{E}\ ,\ n\ \rangle \Downarrow n}\ \text{(intEval)} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ \texttt{true}\ \rangle \Downarrow \mathit{true}}\ \text{(trueEval)}$$

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

Variables

$$\frac{x \text{ is a variable}}{\langle\ \mathcal{E}\ ,\ x\ \rangle \Downarrow \mathcal{E}(x)}\ \text{(varEval)}$$

Note that, because of type-checking, we don't need to verify that x appears in the environment $\mathcal{E}$.

Operators

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

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

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

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

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

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

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

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

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

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

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

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

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

Note that the type information is not relevant in evaluation.

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

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

Let-Expressions

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

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

Note that, because of our grammar, e_1 in the rule (letRecEval) is guaranteed to be an unnamed closure.

Assert

$$\frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow \mathit{true}}{\langle\ \mathcal{E}\ ,\ \texttt{assert}\ e\ \rangle \Downarrow \texttt{()}}\ (\mathsf{assertEval})$$

Putting Everything Together

After you're done with `parse`, `type_of` and `eval`, you should combines these into a single function called `interp`. This should, in essence be function composition, except that if `parse` fails, `interp` should return `Error ParseFail`. Once you do this, you should be able to run

```
dune exec interp02 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`:

```
let sum_of_squares (x : int) (y : int) : int =
  let x_squared : int = x * x in
  let y_squared : int = y * y in
  x_squared + y_squared
let _ : unit = assert (sum_of_squares 3 (-5) = 34)
```

Note that we can now use assert statements to test our own implementations (as long as we've correctly implemented the semantics of `assert`).

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 `interp02/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 (e.g., you will probably want to include a directory which contains your parser generated by Menhir).
- ▷ You are 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.

Good luck, happy coding.