

# Mini-Project 2: The Environment Model

CAS CS 320: Principles of Programming Languages

Due November 20, 2025 by 8:00PM

In this project, you'll be building *another* interpreter for a subset of OCaml. It'll be your task to implement the following functions:

```
val desugar : prog -> expr
val type_of : expr -> (ty, error) result
val eval : expr -> value
val interp : string -> (value, error) result
```

The types used in the above signature appear in the module `Utils`. Your implementation of these functions should appear in the file `interp2/lib/interp2.ml`. **Please read the following instructions completely and carefully.**

## Part 0: Parsing

This time around we're giving you the parser. We'll still give you the grammar. You should spend at bit of time looking through the lexer and parser to make sure you understand how it works.

```
<prog> ::= {<toplet>}
<toplet> ::= let [rec] <var> {<arg>} : <ty> = <expr>
<arg> ::= ( <var> : <ty> )
<ty> ::= int | bool | unit | <ty> -> <ty> | ( <ty> )
<expr> ::= let [rec] <var> {<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
```

Operators and their associativity are presented *in order of increasing precedence* in the following table. Note that neither the function type arrow nor function application are "operators" but they're included in the table for completeness.

| Operator   | Associativity |
|--|---------------|
| <code>  </code>                                    | right         |
| <code>&amp;&amp;</code>                            | right         |
| <code>&lt;, &lt;=, &gt;, &gt;=, =, &lt;&gt;</code> | left          |
| <code>+, -</code>                                  | left          |
| <code>*, /, mod</code>                             | left          |
| <i>function application</i>                        | left          |
| <code>→</code>                                     | right         |

## Part 1: Desugaring

You'll notice this time around that the kinds of expressions that we'll eventually type-check and evaluate aren't the same as programs 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 sometimes called *desugaring*.<sup>1</sup> The function `desugar` should translate a program of type `prog` to an expression of type `expr` according to the following rules.

- ▷ A sequence of top-level let-statements is shorthand for collection of nested let-expressions ending in a variable (the last defined name). That is, the surface-level syntax:

```
let [rec] x1 {<arg>} : τ1 = e1
let [rec] x2 {<arg>} : τ2 = e2
...
let [rec] xk {<arg>} : τk = ek
```

is shorthand for:

```
let [rec] x1 {<arg>} : τ1 = e1 in
let [rec] x2 {<arg>} : τ2 = e2 in
...
let [rec] xk {<arg>} : τk = ek in
xk
```

The empty program is equivalent to the unit expression `()`.

- ▷ 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

```
let [rec] f ( x1 : τ1 ) ... ( xk : τk ) : τ = e1 in e2
```

is shorthand for

```
let [rec] f : τ1 → ... τk → τ = fun ( x1 : τ1 ) ... ( xk : τk ) → e1 in e2
```

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

```
fun ( x1 : τ1 ) ... ( xk : τk ) → e
```

is shorthand for

```
fun ( x1 : τ1 ) → ... fun → ( xk : τk ) → 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, so all this means is that you have to determine the valid translation from `prog` to `expr`.

<sup>1</sup>You may have heard the term *syntactic sugar* with regards to convenient syntactic constructs in programming languages you've used.

## Part 2: 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  $\tau$  in the context  $\Gamma$ , where  $e$  is an `<expr>` after desugaring, and  $\tau$  is a `<ty>`. In practice, you'll 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 away the type-annotations, which will make the type-inference problem more difficult).

The function `type_of`, given an expression  $e$  of type `expr`, should evaluate to `Ok  $\tau$`  if  $\emptyset \vdash e : \tau$  is derivable. Otherwise, it should evaluate to `Error err`, where `err` is one of the following constructors of the type `error`.

- ▷ `UnknownVar  $x$` : The variable  $x$  is unbound.
- ▷ `IfTyErr  $\tau_e \tau_a$` : The else-case of an if-expression was expected to be type  $\tau_e$  (based on the then-case) but was determined to be type  $\tau_a$ .
- ▷ `IfCondTyErr  $\tau_a$` : The condition of an if-expression was expected to be type `bool` but was determined to be type  $\tau_a$ .
- ▷ `OpTyErrL  $\square \tau_e \tau_a$` : The left argument of an operator  $\square$  was expected to be type  $\tau_e$  but was determined to be type  $\tau_a$ .
- ▷ `OpTyErrR  $\square \tau_e \tau_a$` : The right argument of an operator  $\square$  was expected to be type  $\tau_e$  but was determined to be type  $\tau_a$ .
- ▷ `FunArgTyErr  $\tau_e \tau_a$` : The argument of a function of type  $\tau_e \rightarrow \tau$  was expected to be type  $\tau_e$  but was determined to be type  $\tau_a$ .
- ▷ `FunAppTyErr  $\tau_a$` : An expression determined to be type  $\tau_a$  was applied as a function but  $\tau_a$  is not a function type.
- ▷ `LetTyErr  $\tau_e \tau_a$` : The binding of a let-expression was expected to be type  $\tau_e$  (according to its annotation) but was determined to be type  $\tau_a$ .
- ▷ `LetRecErr  $f$` : The binding of  $f$  in a recursive let-expression was found to be something other than an anonymous function.
- ▷ `AssertTyErr  $\tau_a$` : The argument of an assert-expression was expected to be type `bool` but was determined to be type  $\tau_a$ .

It'll 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**.

### Literals

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

### Variables

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

## Operators

$$\begin{array}{c}
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ (add)} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 - e_2 : \text{int}} \text{ (sub)} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 * e_2 : \text{int}} \text{ (mul)} \\
\\
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 / e_2 : \text{int}} \text{ (div)} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{ mod } e_2 : \text{int}} \text{ (mod)} \\
\\
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 < e_2 : \text{bool}} \text{ (lt)} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \leq e_2 : \text{bool}} \text{ (lte)} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 > e_2 : \text{bool}} \text{ (gt)} \\
\\
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \geq e_2 : \text{bool}} \text{ (gte)} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 = e_2 : \text{bool}} \text{ (eq)} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 <> e_2 : \text{bool}} \text{ (neq)} \\
\\
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \&\& e_2 : \text{bool}} \text{ (and)} \quad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 || e_2 : \text{bool}} \text{ (or)}
\end{array}$$

## Conditionals

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{ (if)}$$

## Functions

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun } (x : \tau_1) \rightarrow e : \tau_1 \rightarrow \tau_2} \text{ (fun)} \quad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \text{ (app)}$$

## Let-Expressions

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x : \tau_1 = e_1 \text{ in } e_2 : \tau_2} \text{ (let)} \quad \frac{e_1 = \text{fun } (y : \tau) \rightarrow e \quad \Gamma, x : \tau_1 \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let rec } x : \tau_1 = e_1 \text{ in } e_2 : \tau_2} \text{ (letRec)}$$

It's worth thinking about why we need this side-condition. The reason is a bit subtle, but in short we need  $e_1$  to evaluate to an *unnamed* closure.

## Assert

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

## Part 3: Evaluation

The evaluation of a program 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                          |
|----------------------------|--------------------------------------|
| $\emptyset$                | 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 Boolean value (an element of the set  $\mathbb{B}$ ) denoted  $\top$  and  $\perp$
- ▷ unit, denoted  $\bullet$
- ▷ a closure, denoted  $\langle \mathcal{E}, s \mapsto e \rangle$ , where  $\mathcal{E}$  is an environment,  $s$  is a name, and  $e$  is an expression. We'll write  $\langle \mathcal{E}, \cdot \mapsto e \rangle$  for a closure without a name. For this project, we will only consider closures whose expressions are functions, so our **VClos** constructor is specific to this case.

The function **eval**, given an expression  $e$  of type **expr**, should evaluate to  $v$  in the case that  $\langle \emptyset, e \rangle \Downarrow v$  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 (or **mod**) 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

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

$$\frac{}{\langle \mathcal{E}, \text{false} \rangle \Downarrow \perp} \text{ (falseEval)} \quad \frac{}{\langle \mathcal{E}, () \rangle \Downarrow \bullet} \text{ (unitEval)}$$

## 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

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

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

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

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_1 < v_2}{\langle \mathcal{E}, e_1 < e_2 \rangle \Downarrow \top} \text{ (ltTrue)} \quad \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_1 \geq v_2}{\langle \mathcal{E}, e_1 < e_2 \rangle \Downarrow \perp} \text{ (ltFalse)}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_1 \leq v_2}{\langle \mathcal{E}, e_1 \leq e_2 \rangle \Downarrow \top} \text{ (lteTrue)} \quad \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_1 > v_2}{\langle \mathcal{E}, e_1 \leq e_2 \rangle \Downarrow \perp} \text{ (lteFalse)}$$

$$\begin{array}{c}
\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_1 > v_2}{\langle \mathcal{E}, e_1 > e_2 \rangle \Downarrow \top} \text{(gtTrue)} \quad \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_1 \leq v_2}{\langle \mathcal{E}, e_1 > e_2 \rangle \Downarrow \perp} \text{(gtFalse)} \\
\\
\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_1 \geq v_2}{\langle \mathcal{E}, e_1 \geq e_2 \rangle \Downarrow \top} \text{(gteTrue)} \quad \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_1 < v_2}{\langle \mathcal{E}, e_1 \geq e_2 \rangle \Downarrow \perp} \text{(gteFalse)} \\
\\
\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_1 = v_2}{\langle \mathcal{E}, e_1 = e_2 \rangle \Downarrow \top} \text{(eqTrue)} \quad \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_1 \neq v_2}{\langle \mathcal{E}, e_1 = e_2 \rangle \Downarrow \perp} \text{(eqFalse)} \\
\\
\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_1 \neq v_2}{\langle \mathcal{E}, e_1 <> e_2 \rangle \Downarrow \top} \text{(neqTrue)} \quad \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_1 = v_2}{\langle \mathcal{E}, e_1 <> e_2 \rangle \Downarrow \perp} \text{(neqFalse)} \\
\\
\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \perp}{\langle \mathcal{E}, e_1 \&\& e_2 \rangle \Downarrow \perp} \text{(andFalse)} \quad \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \top \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, e_1 \&\& e_2 \rangle \Downarrow v_2} \text{(andTrue)} \\
\\
\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \top}{\langle \mathcal{E}, e_1 || e_2 \rangle \Downarrow \top} \text{(orTrue)} \quad \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \perp \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, e_1 || e_2 \rangle \Downarrow v_2} \text{(orFalse)}
\end{array}$$

## Conditionals

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \top \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v}{\langle \mathcal{E}, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rangle \Downarrow v} \text{(ifTrue)} \quad \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \perp \quad \langle \mathcal{E}, e_3 \rangle \Downarrow v}{\langle \mathcal{E}, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rangle \Downarrow v} \text{(ifFalse)}$$

## Functions

$$\frac{}{\langle \mathcal{E}, \text{fun } (x : \tau) \rightarrow e \rangle \Downarrow \llbracket \mathcal{E}, \cdot \mapsto \text{fun } (x : \tau) \rightarrow e \rrbracket} \text{(funEval)} \\
\\
\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \llbracket \mathcal{E}', \cdot \mapsto \text{fun } (x : \tau) \rightarrow e \rrbracket \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}'[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v} \text{(appEval)} \\
\\
\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \llbracket \mathcal{E}', s \mapsto \text{fun } (x : \tau) \rightarrow e \rrbracket \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}'[s \mapsto \llbracket \mathcal{E}', s \mapsto \text{fun } (x : \tau) \rightarrow e \rrbracket][x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v} \text{(appRecEval)}$$

## Let-Expressions

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{let } x : \tau = e_1 \text{ in } e_2 \rangle \Downarrow v_2} \text{(letEval)} \\
\\
\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \llbracket \mathcal{E}', \cdot \mapsto \text{fun } (x : \tau) \rightarrow e \rrbracket \quad \langle \mathcal{E}[f \mapsto \llbracket \mathcal{E}', f \mapsto \text{fun } (x : \tau) \rightarrow e \rrbracket], e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{let rec } f : \tau_1 = e_1 \text{ in } e_2 \rangle \Downarrow v_2} \text{(letRecEval)}$$

Because of the side condition in the (letRec) rule, the expression  $e_1$  in the rule (letRecEval) is guaranteed to be an unnamed closure.

## Assert

$$\frac{\langle \mathcal{E}, e \rangle \Downarrow \top}{\langle \mathcal{E}, \text{assert } e \rangle \Downarrow ()} \text{ (assertEval)}$$

In your implementation, if  $e$  does not evaluate to  $\top$ , then you should raise an assertion failure. This is a *side effect*, it's not reflected in the semantics proper.

## Putting Everything Together

After you're done with `parse`, `type_of`, and `eval`, you should combine 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've implemented `interp`, you should be able to run:

```
dune exec interp2 filename
```

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)
```

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 `interp2/lib/interp2.ml`. If you don't do this, we can't grade your submission. You are, of course, allowed to add your own definitions.
- ▷ 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 assume this skeleton.
- ▷ We will not immediately release examples or the autograder. You should test yourself as best as you can first.

Good luck, happy coding.