1 Instructions

Begin by downloading the file `hw2-base.ml` from the course website, and renaming it to `hw2.ml`. This file contains the outline of the homework, with incomplete definitions commented out. For each problem, remove the comments and fill in the associated definitions. You can also add functions and modify the outline as you see fit. Make sure to answer the problems in both Part 2 (BNF Grammars and ASTs) and Part 3 (Type Checking). Submit your completed `hw2.ml` via Gradescope. As always, please don’t hesitate to ask for help on Piazza (https://piazza.com/class/jushbmjmi1l3yb).

2 BNF Grammars and ASTs

Here is a BNF grammar:

\[
\begin{align*}
C & ::= \text{let } \langle \text{ident} \rangle = \langle \text{num} \rangle | \text{return } \langle \text{ident} \rangle | C ; C \\
F & ::= \langle \text{ident} \rangle () \{ C \} | F F
\end{align*}
\]

1. (4 points) Write datatypes `ast_C` and `ast_F` of abstract syntax trees for `C` and `F` respectively, where `⟨ident⟩` is represented by the `string` type and `⟨num⟩` is represented by the `int` type. This grammar is completely separate from the grammar for expressions, so make sure that your constructors do not include arguments of type `exp`.

2. (2 points) Define a value `tree1 : ast_F` that is an AST for the term

   \[
   \text{f()} \{ \text{let } x = 5; \text{return } x \}
   \]

3. (3 points) Write a function `count_lets : ast_F \to \text{int}` that counts the number of `let` nodes in an AST for `F`. As an example, `count_lets tree1` should return 1.

   Hint: start by writing a function `count_lets_C` that counts the number of `let` nodes in an AST for `C`. 
4. (3 points) Write a function \texttt{count\_id : string -> ast\_F -> int} that takes a string \texttt{s} and an AST for \texttt{F} and counts the number of times \texttt{s} appears as an identifier in the AST. As an example, \texttt{count\_id "x" tree1} should return \texttt{2}.

Hint: as above, start by writing a function that counts occurrences in \texttt{C}.

3 Type Checking

The rules of the type system for expressions are:

\[
\begin{array}{c}
\frac{}{n \text{ is a number}} \quad \frac{}{n \text{ is a boolean}} \\
\frac{}{n : \text{int}} \quad \frac{}{n : \text{bool}}
\end{array}
\]

\[
\begin{array}{c}
\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 \oplus e_2 : \text{int}} \quad \text{where } \oplus \text{ is an arithmetic operator}
\end{array}
\]

\[
\begin{array}{c}
\frac{e_1 : \text{bool} \quad e_2 : \text{bool}}{e_1 \otimes e_2 : \text{bool}} \quad \text{where } \otimes \text{ is a boolean operator}
\end{array}
\]

\[
\begin{array}{c}
\frac{e_1 : \tau \quad e_2 : \tau}{e_1 = e_2 : \text{bool}} \\
\frac{e : \text{bool} \quad e_1 : \tau \quad e_2 : \tau}{\text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}
\end{array}
\]

The provided \texttt{hw2-base.ml} includes a type \texttt{exp} of ASTs for expressions, and a type \texttt{typ} for types. \texttt{Tint} is the type of ints in the expression language, and \texttt{Tbool} is the type of bools in the expression language; be careful not to confuse them with \texttt{int} and \texttt{bool}, which are types in OCaml itself.

(13 points) Write a function \texttt{type\_of : exp -> typ option} that returns \texttt{Some t} if the given expression has type \texttt{t}, and \texttt{None} if the expression fails to typecheck. For instance, \texttt{type\_of (Add (Int 3, Int 4))} should be \texttt{Some TInt}, and \texttt{type\_of (If (Bool false, Int 3, Bool true))} should be \texttt{None}.

Hint: the structure of the function will closely resemble that of the interpreter \texttt{eval} that we built in class, and the typing rules tell you what to do in each case.