Name (printed): ____________________________________________________________

Username (PennKey login id): ________________________________

My signature below certifies that I have complied with the University of Pennsylvania’s Code of Academic Integrity in completing this examination.

Signature: ____________________________________ Date: ___________________

Directions:

• This exam contains both standard and advanced-track questions. Questions with no annotation are for both tracks. Other questions are marked “Standard Track Only” or “Advanced Track Only.”

  Do not waste time (or confuse the graders) by answering questions intended for the other track. To make sure, please look for the questions for the other track as soon as you begin the exam and cross them out!

• Before beginning the exam, please write your PennKey (login ID) at the top of each even-numbered page (so that we can find things if a staple fails!).

Mark the box of the track you are following.

☐ Standard ☐ Advanced
1  (8 points)

Regular Expressions

Recall the definitions of regular expressions and regular expression matching in Coq. (For reference, we have included the relevant definitions on page 2 in the appendix.)

One interesting property of a regular expression is its cardinality—i.e., the number of strings that it matches. For example, here are some regular expressions along with their cardinalities:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmptySet</td>
<td>0</td>
</tr>
<tr>
<td>EmptyStr</td>
<td>1</td>
</tr>
<tr>
<td>Char &quot;x&quot;</td>
<td>1</td>
</tr>
<tr>
<td>Union (Char &quot;x&quot;) (Char &quot;y&quot;)</td>
<td>2</td>
</tr>
<tr>
<td>Star (Char &quot;x&quot;)</td>
<td>infinite</td>
</tr>
</tbody>
</table>

Choose the correct cardinality for each of the following regular expressions:

1.1 App (Union (Char "x") (Char "y")) (Union (Char "z") (Char "w"))
   - 0    □ 1    □ 2    □ 3    □ 4    □ infinite

1.2 App (Char "x") (Char "x")
   - 0    □ 1    □ 2    □ 3    □ 4    □ infinite

1.3 Union (Char "x") (Char "x")
   - 0    □ 1    □ 2    □ 3    □ 4    □ infinite

1.4 Star EmptyStr
   - 0    □ 1    □ 2    □ 3    □ 4    □ infinite

1.5 Star EmptySet
   - 0    □ 1    □ 2    □ 3    □ 4    □ infinite

1.6 Star (Union (Char "x") EmptySet)
   - 0    □ 1    □ 2    □ 3    □ 4    □ infinite

1.7 Star (Star EmptyStr)
   - 0    □ 1    □ 2    □ 3    □ 4    □ infinite

1.8 Star (Star EmptySet)
   - 0    □ 1    □ 2    □ 3    □ 4    □ infinite
Inductively Defined Relations

The following inductively defined predicate classifies regular expressions whose cardinality is nonzero (i.e., those that match at least one string).

\[
\text{Inductive nonempty } { T : \text{Type} } , \text{reg_exp } T \rightarrow \text{Prop} := \\
\begin{align*}
| \text{ne_char } (c : T) : & \quad \text{nonempty } \text{Char } c \\
| \text{ne_app } (e1 \ e2 : \text{reg_exp } T) : & \quad \text{nonempty } e1 \rightarrow \text{nonempty } e2 \rightarrow \text{nonempty } (\text{App } e1 \ e2) \\
| \text{ne_union_l } (e1 \ e2 : \text{reg_exp } T) : & \quad \text{nonempty } e1 \rightarrow \text{nonempty } (\text{Union } e1 \ e2) \\
| \text{ne_union_r } (e1 \ e2 : \text{reg_exp } T) : & \quad \text{nonempty } e2 \rightarrow \text{nonempty } (\text{Union } e1 \ e2) \\
| \text{ne_star } (e : \text{reg_exp } T) : & \quad \text{nonempty } (\text{Star } e) \\
| \text{ne_emptystr} : & \quad \text{nonempty } \text{EmptyStr} \\
\end{align*}
\]

For example:

\[
\text{nonempty } (\text{Char } "x") \\
\text{nonempty } (\text{App } (\text{Char } "x") (\text{Char } "x")) \\
\sim \text{nonempty } (\text{App } (\text{Char } "x") \text{EmptySet})
\]
The inductively defined predicate `contains_char` classifies regular expressions that contain some strings with at least one character in them. That is, expressions that are empty or only contain the empty string do not satisfy this predicate. For example:

```lean
contains_char (Char "x")
contains_char (App (Char "x") (Char "x") )
¬ contains_char EmptyStr.
¬ contains_char EmptySet
¬ contains_char (App (Char "x") EmptySet)
```

Complete the definition of `contains_char`.

```lean
Inductive contains_char {T : Type} : reg_exp T -> Prop :=
```

The inductively defined predicate `infinite` classifies regular expressions of infinite cardinality (i.e., ones that match infinitely many strings). For example:

```lean
infinite (Star (Char "x" ))
¬ infinite (Star EmptyStr)
¬ infinite (Star (App EmptySet (Char "x")) )
```

Complete the definition of `infinite`. Your solution may refer to `nonempty` and/or `contains_char`.

```lean
Inductive infinite {T : Type} : reg_exp T -> Prop :=
```
Each of the following Hoare triples contain a variable $c$, representing an arbitrary Imp command. For each triple, check the appropriate box to indicate whether the triple is always valid (valid for all choices of $c$), sometimes valid (valid for some, but not all, choices of $c$), or never valid (invalid for all choices of $c$). If you choose sometimes valid, give an example of a command $c_1$ that makes the triple valid and a command $c_2$ that makes the triple invalid.

3.1 \[ \{ \{ X < Y \} \} \]

\[ X := Y; \]

\[ c \]

\[ \{ \{ X = Y \} \} \]

\[ \square \] Always valid \[ \square \] Sometimes valid \[ \square \] Never valid

$c_1 =$

$c_2 =$

3.2 \[ \{ \{ X = 2 \} \} \]

\[ \text{while } (0 < X) \text{ do } \]

\[ c; \]

\[ X := X + 1 \]

end

\[ \{ \{ X = 2 \} \} \]

\[ \square \] Always valid \[ \square \] Sometimes valid \[ \square \] Never valid

$c_1 =$

$c_2 =$

3.3 \[ \{ \{ \text{True} \} \} \]

\[ \text{if } (X < 10) \]

\[ \quad \text{then } \text{while } (X < 10) \text{ do } c \text{ end } \]

\[ \text{else } c \]

end

\[ \{ \{ X > 10 \} \} \]

\[ \square \] Always valid \[ \square \] Sometimes valid \[ \square \] Never valid

$c_1 =$

$c_2 =$
3.4 \[
\{ X = Y + 2 \}\]
   \text{if} (Y < X) 
     \text{then} \quad \text{while} (0 < Y) \text{ do } c \text{ end} 
     \text{else} \quad c 
   \text{end} 
\{ Y > X \}\]

\begin{tabular}{ccc}
\checkmark & Always valid & \checkmark & Sometimes valid & \checkmark & Never valid
\end{tabular}

c1 = 
c2 = 

3.5 \[
\{ X = Y \}\]
   Y := X + Y; 
   \text{while} (X < Y) \text{ do } c \text{ end} 
\{ X = Y \}\]

\begin{tabular}{ccc}
\checkmark & Always valid & \checkmark & Sometimes valid & \checkmark & Never valid
\end{tabular}

c1 = 
c2 =
STLC with Nondeterminism

We've seen how nondeterminism in the form of a havoc command can be added to the Imp language. In this problem, we'll consider adding a similar feature to the STLC.

First, we add a new term, havoc, to the syntax of terms. This term has type Nat and can step to any constant natural number. Here is the fragment of the step relation that pertains to havoc.

\[
\text{Inductive step : } \text{tm } \rightarrow \text{tm } \rightarrow \text{Prop} := \\
| \text{ST_Havoc : forall (n : nat), } \\
| \quad \langle\text{ havoc }\rangle \rightarrow \langle n \rangle \\
\]

This version of the simply typed lambda calculus with havoc also has let bindings, the fix combinator, and basic arithmetic. These features all have the same semantics as they were given in lecture and homework assignments. The full definition can be found on page 3 of the appendix, for reference.

Since a single term can now step (and hence also multistep) to many different values, it is interesting to compare terms according to the sets of final values they can reduce to. We say that a term t1 refines a term t2 if the set of values that t1 reduces to is a subset of the set of values that t2 reduces to.

For example, the term

\[1\ast 1\]

refines the term

\[\text{ havoc }+ 1\]

because the first reduces (only) to the numeric value 1, while the second term can reduce to any numeric value except 0. Conversely, a counterexample for the claim “term t1 refines the term t2” is a value v that t1 reduces to but that t2 cannot reduce to.

Here are several pairs of terms in the simply typed lambda calculus extended with havoc. For each pair, first answer whether the one on the left refines the one on the right, then answer whether the one on the right refines the one on the left.
4.1 \[\text{havoc} \quad 2 \cdot \text{havoc}\]

Does the expression on the left refine the expression on the right?
If not, show a counterexample.

[ ] Yes
[ ] No (Give final value below)
\[v = \]

4.2 \[\text{havoc} \quad \text{havoc} \cdot \text{havoc}\]

Does the expression on the right refine the expression on the left?
If not, show a counterexample.

[ ] Yes
[ ] No (Give final value below)
\[v = \]

4.3 \[\text{havoc} \quad \text{havoc} \cdot \text{havoc}\]

Does the expression on the left refine the expression on the right?
If not, show a counterexample.

[ ] Yes
[ ] No (Give final value below)
\[v = \]

4.4 \[\text{havoc} \quad \text{havoc} \cdot \text{havoc}\]

Does the expression on the right refine the expression on the left?
If not, show a counterexample.

[ ] Yes
[ ] No (Give final value below)
\[v = \]

4.5 \[\text{let } x := \text{havoc} \text{ in } x \cdot x \quad \text{havoc} \cdot \text{havoc}\]

Does the expression on the left refine the expression on the right?
If not, show a counterexample.

[ ] Yes
[ ] No (Give final value below)
\[v = \]

4.6 \[\text{let } x := \text{havoc} \text{ in } x \cdot x \quad \text{havoc} \cdot \text{havoc}\]

Does the expression on the right refine the expression on the left?
If not, show a counterexample.

[ ] Yes
[ ] No (Give final value below)
\[v = \]
4.7

\[ (x : \text{Nat}, \]
\[ \quad \text{if } x = \text{havoc} \text{ then } 1 \text{ else } 2) \]
\[ \text{havoc} \]
\[ \]
\[ (x : \text{Nat}, \]
\[ \quad \text{if } x = \text{havoc} \text{ then } 1 \text{ else } 2) \]
\[ 1 \]

Does the expression on the left refine the expression on the right?  
If not, show a counterexample.

\[ \]
\[ \quad \text{Yes} \]
\[ \quad \text{No} \text{ (Give final value below)} \]
\[ v = \]

Does the expression on the right refine the expression on the left?  
If not, show a counterexample.

\[ \]
\[ \quad \text{Yes} \]
\[ \quad \text{No} \text{ (Give final value below)} \]
\[ v = \]

4.8

\text{fix} (f : \text{Nat} \rightarrow \text{Nat}, \]
\[ x : \text{Nat}, \]
\[ \text{if } x = 0 \text{ then } x \text{ else } f \text{ havoc) \]
\[ 0 \]

Does the expression on the left refine the expression on the right?  
If not, show a counterexample.

\[ \]
\[ \quad \text{Yes} \]
\[ \quad \text{No} \text{ (Give final value below)} \]
\[ v = \]

Does the expression on the right refine the expression on the right?  
If not, show a counterexample.

\[ \]
\[ \quad \text{Yes} \]
\[ \quad \text{No} \text{ (Give final value below)} \]
\[ v = \]

4.9

\text{havoc}  

4.10

\text{fix} (f : \text{Nat} \rightarrow \text{Nat}, \]
\[ x : \text{Nat}, \]
\[ \text{if } x = \text{havoc} \text{ then } x \text{ else } f \text{ havoc) \]
\[ 1 \]

Does the expression on the left refine the expression on the right?  
If not, show a counterexample.

\[ \]
\[ \quad \text{Yes} \]
\[ \quad \text{No} \text{ (Give final value below)} \]
\[ v = \]
Does the expression on the right refine the expression on the right?  
If not, show a counterexample.

\[
\text{fix}\ (\forall f : \text{Nat} \to \text{Nat}, \ x : \text{Nat}, \ \text{let}\ y := \text{havoc}\ \text{in}\  
\begin{align*}
\text{if}\ y = 0 \\
\quad &\text{then}\ y \\
\text{else}\ f\ x)
\end{align*}
\]

Does the expression on the left refine the expression on the right?  
If not, show a counterexample.

\[\text{fix}\ (\forall f : \text{Nat} \to \text{Nat}, \ x : \text{Nat}, f\ x)\]

\[0\]

Does the expression on the right refine the expression on the right?  
If not, show a counterexample.

\[
\text{fix}\ (\forall f : \text{Nat} \to \text{Nat}, \ x : \text{Nat}, f\ x)
\]

\[0\]
Contextual Equivalence in the STLC

We saw in the Equiv chapter how to define and reason about “behavioral equivalence” of programs in Imp. Intuitively, two Imp programs are equivalent if they embody the same partial function from input states to output states. In the STLC, there are no “states,” so, to define equivalence, we can just compare outputs. For example, the terms \((\lambda x: \text{Bool}. x) \text{true}\) and \((\lambda x: \text{Bool}. \text{true}) \text{false}\) are equivalent because they both reduce to the same value, \text{true}.

But what should we say about equivalence of terms with function types? For example, \((\lambda x: \text{Bool}. x)\) and \((\lambda x: \text{Bool}. \text{if } x \text{ then true else false})\) are “obviously” equivalent, even though they do not reduce to the same thing (both are already values). One popular answer to this puzzle is the notion of contextual equivalence—the idea that two functions should be considered equal if they behave the same when placed in any larger context.

Formally, we define a term context \(P\) to be an incomplete term containing one or more holes, written [], that are waiting to be filled with a term.

```
Inductive tctx : Type :=
| P_hole          
| P_var : string -> tctx
| P_true : tctx
| P_false : tctx
| P_app : tctx -> tctx -> tctx
| P_abs : string -> ty -> tctx -> tctx
| P_if : tctx -> tctx -> tctx -> tctx.
```

We’ll write term contexts using the same concrete notations as for ordinary terms, except that, in formal Coq definitions, we’ll enclose them in [{}...] brackets to tell the parser that they may contain holes.

We next define a function, fill, that “plugs in” an arbitrary term into the holes in a term context.

```
Fixpoint fill (t:tm) (P:tctx) : tm :=
  match P with
  | P_var x => tm_var x
  | [] => t
  | [true ] => <true >
  | [false ] => <false >
  | [P1 P2] => let t1' := fill t P1 in
              let t2' := fill t P2 in
              <t1' t2'>
  | [\x:T, P] => let t' := fill t P in
               <\x : T, t'>
  | [if P1 then P2 else P3] =>
    let t1' := fill t P1 in
    let t2' := fill t P2 in
    let t3' := fill t P3 in
    <if t1' then t2' else t3'>
  end.
```
Notice that the term $t$ being plugged into the hole is not required to be closed: it can have free variables, which can then be captured by variable binders above the hole in the term context $P$. That is, hole filling can be thought of as a sort of "non-capture-avoiding substitution." For instance,

\[
\text{fill } \{ x \ (\ y : \text{Bool} , y) \} \rightarrow \{ \ \ x : (\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool} , [~] \}
\]

yields the term:

\[
\ x : (\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool} , \ x \ (\ y : \text{Bool} , y)
\]

Finally, we say that terms $x$ and $y$ are contextually equivalent if they can be plugged into any term context $P$ and produce the same results. Formally:

\[
\text{Definition \ contextually_equivalent \ (t1 \ t2 : tm) : Prop :=}
\forall (P : tctx) (v : tm),
\text{value } v \rightarrow
\left((\text{fill } t1 \ P) \rightarrow* v \leftrightarrow (\text{fill } t2 \ P) \rightarrow* v\right).
\]

Note that this definition does not require that the terms being compared be either well typed (we are ignoring typing completely) or closed. In particular, understanding whether two terms are contextually equivalent requires considering how free variables in each might be bound when they are plugged into a term context.

For instance, these two terms are contextually equivalent

\[
\text{if true then } x \ \text{else } y \quad \text{if false then } y \ \text{else } x
\]

because, whenever they are plugged into a term context that binds variables $x$ and $y$, both of them will reduce to the same value. For example, plugging the first into the term context

\[
(\ x : \text{Bool} , \ y : \text{Bool} , [~]) \ \text{true} \ \text{false}
\]

yields the term

\[
(\ x : \text{Bool} , \ y : \text{Bool} , \text{if true then } x \ \text{else } y) \ \text{true} \ \text{false}
\]

which reduces to $\text{true}$, while plugging the second term into the same context yields

\[
(\ x : \text{Bool} , \ y : \text{Bool} , \text{if false then } y \ \text{else } x) \ \text{true} \ \text{false}
\]

which again reduces to $\text{true}$.
By contrast, these terms are not contextually equivalent

\[
\begin{align*}
\text{\(\forall y: \text{Bool}, y\)} & & \text{\(\forall y: \text{Bool}, x\)}
\end{align*}
\]

because we can exhibit a distinguishing context for them. A distinguishing context for terms \(t_1\) and \(t_2\) is a term context \(P\) such that \(\text{fill} \ t_1 \ P \rightarrow^* v_1\) and \(\text{fill} \ t_2 \ P \rightarrow^* v_2\), where \(v_1\) and \(v_2\) are distinct boolean values. Here is a distinguishing context \(P\) for these terms.

\((\forall x: \text{Bool}, [] \text{false}) \text{true}\)

Filling the hole in \(P\) with the first term yields

\((\forall x: \text{Bool}, (\forall y: \text{Bool}, y) \text{false}) \text{true}\)

which reduces to \text{false}. On the other hand, filling \(P\) with the second term yields

\((\forall x: \text{Bool}, (\forall y: \text{Bool}, x) \text{false}) \text{true}\)

which reduces to \text{true}.

For each of the following, determine if the two given terms are contextually equivalent (CE). If they are not, give an example of a distinguishing context \(P\).

5.1

\[
\begin{align*}
\text{if x then true else false} & & \text{if (if x then false else true) then false else true}
\end{align*}
\]

[] CE

[] Not CE (give a distinguishing context \(P\) below)

\(P = \)

5.2

\[
\begin{align*}
(\forall y: \text{Bool}, (\forall x: \text{Bool}, y)) & & (\forall y: \text{Bool}, (\forall x: \text{Bool}, x))
\end{align*}
\]

[] CE

[] Not CE (give a distinguishing context \(P\) below)

\(P = \)

5.3

\[
\begin{align*}
\text{if x then true else false} & & \text{if x then true else true}
\end{align*}
\]

[] CE

[] Not CE (give a distinguishing context \(P\) below)

\(P = \)
5.4

\[(\langle f: \text{Bool} \rightarrow \text{Bool}, \\
(\langle a: \text{Bool}, f \ a \rangle) \ x \ y)\]

[] CE
[] Not CE (give a distinguishing context P below)

\[p = \]

5.5

If false then x else true

if true then x else false

[] CE
[] Not CE (give a distinguishing context P below)

\[p = \]

5.6

\[(\langle y: \text{Bool}, y \rangle)\]

\[(\text{if } x \ \text{then } (\langle y: \text{Bool}, x \rangle) \\
\quad \text{else } (\langle y: \text{Bool}, y \rangle))\]

[] CE
[] Not CE (give a distinguishing context P below)

\[p = \]
STLC with I/O

Suppose we want to add input and output to the simply typed lambda calculus (with base type \texttt{Nat}). A straightforward way to do this is to add three new forms of term:

- \texttt{print n}, which adds \texttt{n} to a list of outputs from the program;
- \texttt{read}, which returns the next number from a list that is provided to the program when it begins executing; and
- \texttt{t1; t2}, which evaluates \texttt{t1}, ignores its result, and returns the result of evaluating \texttt{t2}, thus forcing all of the side effects (reads and writes) of \texttt{t1} to occur before those of \texttt{t2}.

Here are a few examples showing how these new forms of terms reduce. Note that the input list is the first list element of the tuple, and the output list is the second.

\[
\langle\langle (\lambda x : \texttt{Nat}, \lambda y : \texttt{Nat}, x) \texttt{read read }\rangle, [5;6;7], [1;2]\rangle
\rightarrow^* \langle\langle 5 \rangle, [7], [1;2]\rangle
\]

(That is, if the remaining input list contains 5, 6, and 7 and if 2 and 1 have been output, then this expression reads 5 and 6, yields 5 as its result, and leaves the input list two elements shorter and the output list unchanged.)

\[
\langle\langle \texttt{print 1; print 2; print 3 }\rangle, [], []\rangle
\rightarrow^* \langle\langle 3 \rangle, [], [3;2;1]\rangle
\]

(That is, printing 1, then 2, then 3, starting from empty input and output lists, yields the result 3 and the output list $[3;2;1]$, in that order.)

\[
\langle\langle \texttt{print read }\rangle, [], []\rangle
\rightarrow^* \langle\langle 0 \rangle, [], [0]\rangle
\]

(Reading from an empty input list yields 0.)
Here is the definition of the has_type relation for this language.

\[
\text{Inductive has_type : context -> tm -> ty -> Prop :=}
\text{(* pure STLC *)}
| T_Var : forall Gamma x T1, 
\quad Gamma x = Some T1 -> 
\quad Gamma |- x \in T1 
| T_Abs : forall Gamma x T1 T2 t1, 
\quad (x \mapsto T2 ; Gamma) |- t1 \in T1 -> 
\quad Gamma |- \lambda : T2, t1 \in (T2 \rightarrow T1) 
| T_App : forall T1 T2 Gamma t1 t2, 
\quad Gamma |- t1 \in (T2 \rightarrow T1) -> 
\quad Gamma |- t2 \in T2 -> 
\quad Gamma |- t1 t2 \in T1 
\text{(* numbers *)}
| T_Nat : forall Gamma (n : nat), 
\quad Gamma |- n \in Nat 
\text{(* I/O *)}
| T_Read : forall Gamma, 
\quad Gamma |- \text{read} \in \text{Nat} 
| T_Print : forall Gamma t, 
\quad Gamma |- t \in \text{Nat} -> 
\quad Gamma |- \text{print } t \in \text{Nat} 
| T_Seq : forall Gamma t1 t2 T1 T2, 
\quad Gamma |- t1 \in T1 -> 
\quad Gamma |- t2 \in T2 -> 
\quad Gamma |- t1 ; t2 \in T2 
\text{where "Gamma } \Gamma \text{' |- } \Gamma \text{' t ' \in'} \ T := (has_type Gamma t T).
Your task is to complete the definition of the `step` relation below. A correct definition will satisfy the following **progress** and **preservation** theorems.

**Theorem progress**: \(\forall (t : \text{tm}) (\text{input output : list nat}) \ (T : \text{ty}),\)
\[
\text{empty} \models t \in T \rightarrow \ (\text{value } t \text{ } \text{\/ } \exists t' \text{ } \text{input'} \text{ } \text{output'},
\text{ } \ (t, \text{input}, \text{output}) \rightarrow (t', \text{input'}, \text{output'})).
\]

**Theorem preservation**: \(\forall (t \ t' : \text{tm}) (\text{input output input'} \text{output'} : \text{list nat}) \ (T : \text{ty}),\)
\[
\text{empty} \models t \in T \rightarrow \ (t,\text{input},\text{output}) \rightarrow (t',\text{input'},\text{output'}) \rightarrow \text{empty} \models t' \in T.
\]

Pay close attention to the provided header.

**Inductive step**: \((\text{tm} \times \text{list nat} \times \text{list nat}) \rightarrow (\text{tm} \times \text{list nat} \times \text{list nat}) \rightarrow \text{Prop} :=\)
Subtyping

For each of the following pairs of types, $S$ and $T$, select the appropriate description of how they are ordered in the subtype relation.

7.1 $S = (\text{Top} \rightarrow \text{Top}) \rightarrow \text{Top}$
$T = \text{Top} \rightarrow \text{Top} \rightarrow \text{Top}$

- $S : < T$
- $T : < S$
- equivalent
- unrelated

7.2 $S = (\text{Top} \times \text{Top}) \rightarrow \text{Top}$
$T = \text{Top} \rightarrow (\text{Top} \rightarrow \text{Top})$

- $S : < T$
- $T : < S$
- equivalent
- unrelated

7.3 $S = \{x : \text{Bool}, y : \text{Top}, z : \text{Bool}\} \rightarrow \text{Top}$
$T = \{x : \text{Bool}, y : \text{Bool}\} \rightarrow \text{Top}$

- $S : < T$
- $T : < S$
- equivalent
- unrelated

7.4 $S = \text{Bool} \times \text{Bool} \rightarrow \text{Bool}$
$T = \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

- $S : < T$
- $T : < S$
- equivalent
- unrelated

7.5 $S = (\text{Top} \rightarrow \text{Bool}) \rightarrow \text{Top}$
$T = (\text{Bool} \rightarrow \text{Top}) \rightarrow \text{Bool}$

- $S : < T$
- $T : < S$
- equivalent
- unrelated

7.6 $S = \{x : \text{Bool}, y : \text{Nat}\} \rightarrow \{y : \text{Nat}, x : \text{Bool}\}$
$T = \{y : \text{Nat}, x : \text{Bool}\} \rightarrow \{x : \text{Bool}, y : \text{Nat}\}$

- $S : < T$
- $T : < S$
- equivalent
- unrelated
Designing a Typed Stack Machine

In this problem your task will be to design a typing relation for the world’s simplest stack machine. The machine itself consists of a straight-line program—a list of instructions—plus a stack that can hold both numeric and boolean values.

\[
\text{Inductive } \text{stack} \text{_element} := \\
| \text{elt} \text{_bool} : \text{bool} \to \text{stack_element} \\
| \text{elt} \text{_nat} : \text{nat} \to \text{stack_element}.
\]

\[
\text{Definition } \text{stack} := \text{list stack} \text{_element}.
\]

\[
\text{Inductive } \text{instr} := \\
| \text{instr} \text{_push} : \text{nat} \to \text{instr} \\
| \text{instr} \text{_add} : \text{instr} \\
| \text{instr} \text{_and} : \text{instr} \\
| \text{instr} \text{_eq} : \text{instr}.
\]

\[
\text{Definition } \text{prog} := \text{list instr}.
\]

A single step of the machine executes a single instruction, taking a starting stack to an ending stack.

\[
\text{Reserved Notation } "i /\ s '->' s" \text{ (at level 40).}
\]

\[
\text{Inductive } \text{step} : \text{instr} \to \text{stack} \to \text{stack} \to \text{Prop} := \\
| \text{ST} \text{_push} : \forall s \text{ n}, \text{instr} \text{_push} \text{ n / s --> } (\text{elt} \text{_nat} \text{ n :: s)} \\
| \text{ST} \text{_add} : \forall s \text{ n1 n2}, \text{instr} \text{_add} / (\text{elt} \text{_nat} \text{ n1 :: elt} \text{_nat} \text{ n2 :: s}) --> (\text{elt} \text{_nat} \text{(n1+n2) :: s)} \\
| \text{ST} \text{_and} : \forall s \text{ b1 b2}, \text{instr} \text{_and} / (\text{elt} \text{_bool} \text{ b1 :: elt} \text{_bool} \text{ b2 :: s}) --> (\text{elt} \text{_bool} \text{(b1&&b2) :: s)} \\
| \text{ST} \text{_eq} : \forall s \text{ n1 n2}, \text{instr} \text{_eq} / (\text{elt} \text{_nat} \text{ n1 :: elt} \text{_nat} \text{ n2 :: s}) --> (\text{elt} \text{_bool} \text{(beq} \text{_nat} \text{ n1 n2) :: s)}
\]

where "i /\ s '->' s" := (\text{step} \ i \ s \ s').

This is lifted to a multi-step reduction relation that executes one or more instructions and threads the ending stack from each into the starting stack for the next.

\[
\text{Reserved Notation } "p '/\ s -->> s" \text{ (at level 40).}
\]

\[
\text{Inductive } \text{multistep} : \text{prog} \to \text{stack} \to \text{stack} \to \text{Prop} := \\
| \text{multi} _\text{refl} : \text{forall} \ (s : \text{stack}), [] / s -->> s \\
| \text{multi} _\text{step} : \text{forall} \ i \ p (s \text{ s} \text{_mid} \text{ s'} : \text{stack}), \text{ i / s -->> s} \text{_mid} --> \\
\text{ p / s} \text{_mid} -->> \text{ s'} --> \\
\text{ (i::p) / s -->> s' }
\]

where "p '/\ s -->> s" := (\text{multistep} \ p \ s \ s').
Notice that this machine can get stuck in various situations, when there are not enough operands on the stack, or the top stack elements do not have the right types. E.g.

**Example stepg0**: 
~ exists s',
   instr_add / [elt_nat 4] --> s'.

**Example stepg1**: 
~ exists s',
   instr_add / [elt_nat 4; elt_bool false] --> s'.

To typecheck programs for this machine, we first begin by assigning types to individual stack elements and to whole stacks.

\[
\text{Inductive } \text{ty} := \\
| \text{ty}_\text{Bool} \\
| \text{ty}_\text{Nat}.
\]

\[
\text{Definition } \text{stack\_ty} := \text{list ty}.
\]

\[
\text{Inductive } \text{elt\_has\_type} : \text{stack\_element} \to \text{ty} \to \text{Prop} := \\
| \text{ET\_nat} : \forall n, \text{elt\_has\_type} (\text{elt\_nat n}) \text{ty}_\text{Nat} \\
| \text{ET\_bool} : \forall b, \text{elt\_has\_type} (\text{elt\_bool b}) \text{ty}_\text{Bool}.
\]

\[
\text{Reserved Notation } "\text{\_\_} s "\text{'in'}*" \text{sty}" \text{(at level 40)}.
\]

\[
\text{Inductive } \text{stack\_has\_type} : \text{stack} \to \text{stack\_ty} \to \text{Prop} := \\
| \text{SHT\_nil} : \text{\_\_} \text{s 'in'}* \text{sty} --> \text{sty} --> \text{Prop} := \\
| \text{SHT\_cons} : \forall s \text{ sty e T}, \\
   \text{s 'in'}* \text{sty} -> \\
   \text{elt\_has\_type} e \text{T} -> \\
   \text{(e::s) 'in'}* (\text{T:: sty})
\]

where "\text{\_\_} s "\text{'in'}*" \text{sty}" := (\text{stack\_has\_type} \text{s sty}).

Your task is to fill in the details of the following definitions.

**8.1** First, complete the following definition of the typing relation for individual instructions. Notice that it relates an instruction and two types, one describing the stack before the instruction executes and one for the state after. For example:

\[
\text{Example eg0} : \\
\text{\_\_ instr\_push 4 \text{'in'} [ty\_Nat] --> [ty\_Nat; ty\_Nat].}
\]

\[
\text{Reserved Notation } "\text{\_\_} i \text{'in'} st --> \text{st!}" \text{(at level 40)}.
\]

\[
\text{Inductive } \text{instr\_has\_type} : \text{instr} \to \text{stack\_ty} \to \text{stack\_ty} \to \text{Prop} :=
\]

**8.2** Next, use the instruction typing relation to define a similar relation describing how executing a whole program changes the shape of the stack.
Example egl:
|- [instr_push 4; instr_push 6]
\in [] --\* [ty_Nat; ty_Nat].

Reserved Notation "|- p \in st --\* st" (at level 40).

Inductive prog_has_type : prog -> stack_ty -> stack_ty -> Prop :=
9   [Advanced Track Only] (16 points)

Progress and Preservation for the Typed Stack Machine

Next (for those on the advanced track or taking the exam as WPE-I), we will prove correctness of
the stack machine typing relation from your solution to the previous problem.

Feel free to state (and prove!) auxiliary lemmas if you need them.

Write your proofs in good, clear English, stating any induction hypotheses explicitly.

Since the \texttt{and} and \texttt{eq} instructions are very similar to \texttt{add}, you can elide the cases for these instructions
and just give the cases for \texttt{push} and \texttt{add}.

9.1 First, show that well-typed machine states (program plus stack) are not stuck.

\begin{verbatim}
Theorem progress : forall i s sty sty',
  |- i \in sty --> sty' ->
  |- s \in* sty -->
  exists s', i / s --> s'.
Proof.
\end{verbatim}

9.2 Next, prove that stepping a single instruction “preserves typing,” in the sense that the type
you assign to the instruction correctly describes the way the instruction transforms the shape
of the stack.

\begin{verbatim}
Lemma step_preserves_typing : forall i s sty s' sty',
  |- i \in sty --> sty' ->
  |- s \in* sty -->
  i / s --> s' -->
  |- s' \in* sty'.
Proof.
\end{verbatim}

9.3 Finally, show that multistep reduction preserves typing in the same sense.

\begin{verbatim}
Theorem multistep_preserves_typing : forall p sty sty',
  |- p \in sty -->* sty' ->
  forall s s',
  |- s \in* sty -->
  p / s -->* s' -->
  |- s' \in* sty'.
Proof.
\end{verbatim}
Correctness of factorial in STLC

For this problem, we will be working in a variant of the simply typed lambda-calculus enriched with numbers and fixed points, summarized on page 6 in the appendix.

Recall the definition of the factorial function as a term in this system:

\[
\text{Definition stlcfact :=}
\{\text{fix} \quad (\text{\textbackslash}f: \text{Nat} \rightarrow \text{Nat}, \\text{\textbackslash}a: \text{Nat}, \quad \text{if} \text{0 a then 1 else } (a * (f \ (\text{pred} a))))\}.
\]

To simplify the reasoning in this problem, we make one technical change to the way the system is defined, replacing the two reduction rules for the “fix” construct with the following “macro rule”

\[
\text{ST_FixAbsApp : forall } f : T1 \ T1', t1 : v2, \quad \text{value } v2 \rightarrow \\
\{\text{fix } (\text{\textbackslash}f: T1' \rightarrow T1', \text{\textbackslash}x: T1, t1) v2 \} \rightarrow \\
\{\ [x := v2] (\{f := \text{fix } (\text{\textbackslash}f: T1' \rightarrow T1', \text{\textbackslash}x: T1, t1)\} t1)\}.
\]

and adding a clause to the definition of values

\[
\text{v_fix : forall } v1, \quad \text{value } v1 \rightarrow \\
\text{value } \{\text{fix } v1\}.
\]

so that a term like

\[
\text{fact } (2+1)
\]

will reduce to

\[
\text{fact 3}
\]

before any reduction steps involving the body of \text{stlcfact} take place.

The \text{ST_FixAbsApp} rule is not quite as powerful as the two rules it replaces

\[
\text{ST_Fix1 : forall } t1 : t1', \quad t1' \rightarrow t1' \rightarrow \\
\{\text{fix } t1 \} \rightarrow \{\text{fix } t1' \}\]}

\[
\text{ST_FixAbs : forall } x : T1 t1, \\
\{\text{fix } (\text{\textbackslash}x : T1, t1)\} \rightarrow \\
\{[x := \text{fix } (\text{\textbackslash}x : T1, t1)] t1\}.
\]

(it does not support mutually recursive functions, for example), but it is a little simpler to reason about because it takes larger steps.
Now, it is intuitively clear that the term \texttt{stlcfact} represents the “real” factorial function that we defined earlier in the semester in Gallina:

\begin{verbatim}
Fixpoint realfact (x:nat) :=
  match x with
  0 => 1
  | S x' => x * (realfact x')
  end.
\end{verbatim}

To state this claim rigorously, we can say that an STLC term \texttt{tf} of type \texttt{Nat->Nat represents} a Coq function \texttt{f} of type \texttt{nat->nat} if applying them to the same input yields the same result.

\begin{verbatim}
Definition represents (tf:tm) (f:nat->nat) : Prop :=
  forall n,
  (tm_app tf (tm_const n)) -->* tm_const (f n).
\end{verbatim}

In particular:

\begin{verbatim}
Theorem fact_correct: represents stlcfact realfact.
\end{verbatim}

Write a careful \textit{informal} proof of this theorem in English. If your proof uses induction, make sure to state the induction hypothesis explicitly. Feel free to state (and prove!) auxiliary lemmas if this is useful.

Since this is an informal proof, the \texttt{<\ldots>\texttt{braces around STLC terms are not needed here (they are used in the textbook just to avoid confusing the Coq parser).}
For Reference
Regular Expressions

**Inductive reg_exp (T : Type) : Type :=**

- EmptySet
- EmptyStr
- Char (t : T)
- App (r1 r2 : reg_exp T)
- Union (r1 r2 : reg_exp T)
- Star (r : reg_exp T).

**Reserved Notation "s =~ re" (at level 80).**

**Inductive exp_match {T} : list T -> reg_exp T -> Prop :=**

- MEmpty : [] =~ EmptyStr
- MChar x : [x] =~ (Char x)
- MApp s1 re1 s2 re2
  - (H1 : s1 =~ re1)
  - (H2 : s2 =~ re2)
  - : (s1 ++ s2) =~ (App re1 re2)
- MUnionL s1 re1 re2
  - (H1 : s1 =~ re1)
  - : s1 =~ (Union re1 re2)
- MUnionR re1 s2 re2
  - (H2 : s2 =~ re2)
  - : s2 =~ (Union re1 re2)
- MStar0 re : [] =~ (Star re)
- MStarApp s1 s2 re
  - (H1 : s1 =~ re)
  - (H2 : s2 =~ (Star re))
  - : (s1 ++ s2) =~ (Star re)

where "s =~ re" := (exp_match s re).
STLC with let, booleans, fix, and havoc

Inductive ty : Type :=
| Ty_Arrow : ty -> ty -> ty
| Ty_Nat : ty
| Ty_Bool : ty.

Inductive tm : Type :=
(* pure STLC *)
| tm_var : string -> tm
| tm_app : tm -> tm -> tm
| tm_abs : string -> ty -> tm -> tm
(* numbers *)
| tm_const : nat -> tm
| tm_succ : tm -> tm
| tm_pred : tm -> tm
| tm_mult : tm -> tm -> tm
(* let *)
| tm_let : string -> tm -> tm -> tm
(* fix *)
| tm_fix : tm -> tm
(* havoc *)
| tm_havoc : tm
(* bools *)
| tm_tru : tm
| tm_fls : tm
| tm_eq : tm -> tm -> tm
| tm_if : tm -> tm -> tm -> tm.

Inductive value : tm -> Prop :=
| v_abs : forall x T2 t1,
  value <\{x:T2, t1}\>
| v_nat : forall n : nat,
  value <\{n\}>
| v_tru : value <\{ true \}>
| v_fls : value <\{ false \}>

Reserved Notation "$' [' x '::=' s ']' t" (in custom stlc at level 20, x constr).

Fixpoint subst (x : string) (s : tm) (t : tm) : tm :=
match t with
(* pure STLC *)
| tm_var y =>
  if eqb_string x y then s else t
| <\{y:T, t1\}> =>
  if eqb_string x y then t else <\{y:T, [x:=s] t1\}>
| <\{t1 t2\}> =>
  <\{((x:=s) t1) ([x:=s] t2)\}>
(* numbers *)
| tm_const _ =>
  t
| <\{succ t1\}> =>
\begin{verbatim}
<\{succ [x := s] t1\}>
| <\{pred t1\} =>
  <\{pred [x := s] t1\}>
| <\{t1 * t2\} =>
  <\{( [x := s] t1) * ( [x := s] t2)\}>
| <\{t1 = t2 \} => <\{( [x := s] t1 ) = ( [x :=s] t2 )\}>
(* booleans *)
| <\{if t1 then t2 else t3\} =>
  <\{if [x := s] t1 then [x := s] t2 else [x := s] t3\}>
| <\{ true \} => <\{ true \}>
| <\{ false \} => <\{ false \}>
(* let *)
| <\{let y := t1 in t2\} =>
  <\{let y := [x:=s] t1
  in (\{ if eqb_string x y then t2 else \{ [x:=s] t2 \}\})\}>
(* fix *)
| <\{ fix t1 \} =>
  <\{ fix ([x:=s] t1) \}>
(* havoc *)
| <\{ havoc \} => <\{ havoc \}>
end

where "\[ x \:=\ s \] t" := (subst x s t) (\textit{in custom stlc}).

Inductive step : tm -> tm -> Prop :=
  (* pure STLC *)
  | ST_AppAbs : forall x T2 t1 v2, value v2 ->
    <\{(\x:T2, t1) v2\} > --\< \{ [x:=v2] t1 \}>
  | ST_App1 : forall t1 t1 t2, t1 --> t1' ->
    <\{t1 t2\} --> <\{t1' t2\}>
  | ST_App2 : forall v1 t2 t2', value v1 ->
    t2 --> t2' ->
    <\{v1 t2\} --> <\{v1 t2'\}>
  (* numbers *)
  | ST_Succ : forall t1 t1',
    t1 --> t1' ->
    <\{succ t1\} --> <\{succ t1'\}>
  | ST_SuccNat : forall n : nat,
    <\{succ n\} --> <\{S n\}>
  | ST_Pred : forall t1 t1',
    t1 --> t1' ->
    <\{pred t1\} --> <\{pred t1'\}>
  | ST_PredNat : forall n:nat,
    <\{pred n\} --> <\{n - 1\}>
  | ST_Mulconsts : forall n1 n2 : nat,
    <\{n1 * n2\} --> <\{n1 * n2\}>
  | ST_Mult1 : forall t1 t1' t2,
    t1 --> t1' ->
    <\{t1 * t2\} --> <\{t1' * t2\}>
\end{verbatim}
| ST.Mult2 : forall v1 t2 t2',
  value v1 ->
  t2 --> t2' ->
  <{v1 * t2}> --> <{v1 * t2'}>
| ST.Eq1 : forall t1 t1' t2,
  t1 --> t1' ->
  <{ t1 = t2 }>,
  <{ t1' = t2 }>
| ST.Eq2 : forall (n : nat) t2 t2',
  t2 --> t2' ->
  <{ n = t2 }>,
  <{ n = t2'}>
| ST.Eq_true : forall (n: nat),
  <{ n = n }>,
  <{ true }>
| ST.Eq_false : forall (n m : nat),
  n <> m ->
  <{ n = m }>,
  <{ false }>
(* booleans *)
| ST.If : forall t1 t1' t2 t3,
  t1 --> t1' ->
  <{if t1 then t2 else t3}> --> <{if t1' then t2 else t3}>
| ST.If_true : forall t2 t3,
  <{if true then t2 else t3}> --> t2
| ST.If_false : forall t2 t3,
  <{if false then t2 else t3}> --> t3
(* let *)
| ST.Let1 : forall x t1 t1' t2,
  t1 --> t1' ->
  <{ let x := t1 in t2}> --> <{ let x := t1' in t2 }>
| ST.LetValue : forall x v1 t2,
  value v1 ->
  <{ let x := v1 in t2 }>,
  <{ [x:=v1]t2 }>
(* fix *)
| ST.Fix1 : forall t1 t1',
  t1 --> t1' ->
  <{ fix t1 }>,
  <{ fix t1' }>
| ST.FixAbs : forall x T1 t1,
  <{ fix \ x : T1, t1 }>,
  <{ [x := fix \ x : T1, t1 ] t1 }>
(* havoc *)
| ST.Havoc : forall (n : nat),
  <{ havoc }>,
  <{ n }>

where "t' -->' t" := (step t t').
STLC with let, products, sums, and alternate rules for fix

Inductive ty : Type :=
  Ty_Arrow : ty -> ty -> ty
  Ty_Nat : ty.

Inductive tm : Type :=
  (* pure STLC *)
  | tm_var : string -> tm
  | tm_app : tm -> tm -> tm
  | tm_abs : string -> ty -> tm -> tm
  (* numbers *)
  | tm_const: nat -> tm
  | tm_succ : tm -> tm
  | tm_pred : tm -> tm
  | tm_mult : tm -> tm -> tm
  | tm_if0 : tm -> tm -> tm -> tm
  (* fix *)
  | tm_fix : tm -> tm.

Inductive value : tm -> Prop :=
  | v_abs : forall x T2 t1,
    value <\{x:=T2, t1\}>
  | v_nat : forall n : nat,
    value <\{n\}>
  (* NEW: fix applied to something is a value if its argument is *)
  | v_fix : forall v1,
    value v1 ->
    value <\{fix v1\}>

Inductive step : tm -> tm -> Prop :=
  (* pure STLC *)
  | ST_AppAbs : forall x T2 t1 v2,
    value v2 ->
    <\{x:=T2, t1\} v2> --> <\{ [x:=v2]t1 \}>
  | ST_App1 : forall t1 t1' t2,
    t1 --> t1' ->
    <\{t1 t2\} --> <\{t1' t2\}>
  | ST_App2 : forall v1 t2 t2',
    value v1 ->
    t2 --> t2' ->
    <\{v1 t2\} --> <\{v1 t2\}>
  (* numbers *)
  | ST_Succ : forall t1 t1',
    t1 --> t1' ->
    <\{succ t1\} --> <\{succ t1\}'>
  | ST_SuccNat : forall n : nat,
    <\{succ n\}> --> <\{S n\} >
  | ST_Pred : forall t1 t1',
    t1 --> t1' ->
    <\{pred t1\} --> <\{pred t1\}>'
  | ST_PredNat : forall n:nat,
\{\text{pred }n\} \rightarrow \{\{n - 1\}\}

| ST_Mulconsts : for all \(n_1 \ n_2 : \text{nat}\),
| \{n_1 \ast n_2\} \rightarrow \{\{n_1 \ast n_2\}\}

| ST_Multi : for all \(t_1 \ t_1' \ t_2\),
| \(t_1 \rightarrow t_1' \rightarrow \)
| \{t_1 \ast t_2\} \rightarrow \{t_1' \ast t_2\}

| ST_Multi2 : for all \(v_1 \ t_2 \ t_2'\),
| \(\text{value }v_1 \rightarrow \)
| \(t_2 \rightarrow t_2' \rightarrow \)
| \{v_1 \ast t_2\} \rightarrow \{v_1 \ast t_2'\}

| ST_If0 : for all \(t_1 \ t_1' \ t_2 \ t_3\),
| \(t_1 \rightarrow t_1' \rightarrow \)
| \{\text{if0 }t_1 \text{ then } t_2 \text{ else } t_3\} \rightarrow \{\text{if0 }t_1' \text{ then } t_2 \text{ else } t_3\}

| ST_If0_Zero : for all \(t_2 \ t_3\),
| \{\text{if0 }0 \text{ then } t_2 \text{ else } t_3\} \rightarrow t_2

| ST_If0_Nonzero : for all \(n \ t_2 \ t_3\),
| \{\text{if0 }\{S \ n\} \text{ then } t_2 \text{ else } t_3\} \rightarrow t_3

(* NEW: macro rule for fix *)

| ST_FixAbsApp : for all \(f \ x \ T_1 \ T_1' \ t_1 \ v_2\),
| \(\text{value }v_2 \rightarrow \)
| \{\text{fix }\(\forall f:T->T'\), \(\forall x:T_1, t_1\) v_2\}\rightarrow
| \{\text{[x := v_2]} \((\text{[f := fix }\{f:T->T'\}, \{x:T_1, t_1\}] t_1)\)\}

(No change to typing or substitution rules.)