Suppose we have the following function for computing the reverse of a list:

```ml
fun rev (nil : int list): int list = nil
| rev (x::l) = (rev l) @ [x];
```

This is, of course, not the only way of computing the reverse of a list functionally. We can, for example, use a helper function with an accumulator which will “eat up” the elements of the list to be reversed in reverse order. Here’s the code:

```ml
fun trevh (nil : int list, acc : int list): int list = acc
| trevh (x::l, acc ) = trevh (l, x::acc );

fun trev (l: int list): int list = trevh (l, []);
```

Maybe this is as intuitive as `rev`, but it has an important advantage. It is tail recursive!

**Definition 1** (Tail recursive). A program is tail recursive if the last operation performed is the recursion call.

When implementing a programming language (not a program!), tail recursion (or, more generally, tail calls) can be optimized because there is no need to add a new frame to the call stack. Think about it. If we have to use the result of the recursive call in an operation, than we need to stack a new frame for the recursive call and store the return address for using the result. If the function returns immediately after the recursive call, the function’s frame can be replaced by that of the recursive call.

If the programming language implements the optimizations of tail recursion properly, changing a program from general to tail recursive improves its performance significantly.

Now, even if I tell you, you don’t have to believe me that those two functions are the same. Why would you? We should only believe in things that have a proof. So let’s try to prove those to programs are the same. But..

What does it mean for two programs to be “the same”?

Give the same answer for the same inputs? What about ill-behaved inputs that infinite loop? What about exceptions? If we want to prove that two programs are the same, we need to first nail down precisely this “same” relation.

It sounds reasonable to think that programs are the same when they behave the same way. Program behaviour is a good place to start. What are the possible behaviors of a program (in SML)?

1. Syntax error;
2. Type error;
3. Infinite loop;
4. Exception;
5. Value.

Well, it wouldn’t make much sense to use programs that have syntax or type errors in a proof, so we will assume they are syntactically correct and type check. Therefore, we do not need to care about behaviors 1 or 2.

This leaves us with the following. Two programs are the same iff:

- They infinite loop for the same inputs;
- They throw the same exceptions for the same inputs;
- They return the same values for the same input.

This looks reasonable and intuitive, but how do we use this in a proof?? Formally speaking, this is still very loose. It is hard to reason with this. We will need a cleaner, more mathematical, notion of program equivalence. What do we expect of such relation?

Well, first of all, as the name indicates, it needs to be an equivalence relation. This means it is symmetric, transitive and reflexive.

We will also consider two programs to be the same if we can use one or another without noticing the difference. This is called a congruence relation. Notice that we do not care what is happening inside the program, as long as it behaves the same to the external world. We can say we want our programs to be extensionally equal. The dual notion is that of intensional equality (that’s intensional, with an s).

Finally, we would like that program equivalence is preserved under evaluation. This means that when an expression is evaluated to something else, these two things are equivalent. For example, in math, if $1 + (2 + 3)$ is evaluated (in one step) to $1 + 5$, then $1 + (2 + 3) = 1 + 5$. We will also require that of programs.

**Definition 2** (Program equivalence ($\cong$)). Let $e$, $e_1$, $e_2$, and $e_3$ be programs (expressions). We define the relation $\cong$ as:

1. $\cong$ is an equivalence relation:
   - Symmetric: $e_1 \cong e_2$ iff $e_2 \cong e_1$
   - Transitive: if $e_1 \cong e_2$ and $e_2 \cong e_3$, then $e_1 \cong e_3$
   - Reflexive: for every $e$, $e \cong e$

2. $\cong$ is a congruence relation: if $e_1 \cong e_2$, then $e[e_1] \cong e[e_2]$.

3. $\cong$ is preserved under evaluation: if $e_1 \mapsto e_2$, then $e_1 \cong e_2$.

That is a drier definition for sure, but clearer and easier to work with.

Nevertheless, there is still something which is vague about program equivalence. Can you guess what it is? We don’t really know what evaluation means... What is this other relation $\mapsto$?? We need to give a meaning to this.

$\mapsto$ is SML’s evaluation semantics.

We will not give a formal definition of evaluation semantics in this course because this requires that we delve in too deep in the realms of programming language design. Luckily, the informal notion will be enough and it turns out to be quite intuitive.

We start with a simple example. How would you evaluate, step by step, the expression: $(2 \times 3) \times (3+4)$? We cannot start with the outermost multiplication because its operands are not values. [Next, some definitions for you] Instead they are redices (singular: redex), which are expressions that can be reduced. When an expression can no longer be reduced, it is called a value.

\[^2e[e_i] \text{ means that } e_i \text{ is in the context of } e\]
So we need to start with the innermost redices. We have the choice of the left or the right one. SML’s convention is to start with the leftmost. Following these rules, here’s the step by step evaluation of that expression:

\[(2 \times 3) \times (3 + 4) \mapsto 6 \times (3 + 4) \mapsto 6 \times 7 \mapsto 42\]

The evaluation semantics in SML follows the strategy: **inner to outer, left to right**.

Let’s try to evaluate more complicated expressions. Given the following code for computing factorial:

```ml
1 fun fact (n: int): int = 
2   if n = 0 then 1 
3   else n * fact(n-1)
```

Here’s the evaluation of `fact(3)`.

\[
\begin{align*}
\text{fact}(3) & \mapsto \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 \times \text{fact}(3 - 1) \\
& \mapsto \text{if false then } 1 \text{ else } 3 \times \text{fact}(3 - 1) \\
& \mapsto 3 \times \text{fact}(3 - 1) \\
& \mapsto 3 \times \text{fact}(2) \\
& \mapsto 3 \times (\text{if } 2 = 0 \text{ then } 1 \text{ else } 2 \times \text{fact}(2 - 1)) \\
& \vdots \\
& \mapsto 3 \times (2 \times (1 \times \text{fact}(1 - 1))) \\
& \mapsto 3 \times (2 \times (1 \times \text{fact}(0))) \\
& \mapsto 3 \times (2 \times (1 \times 1)) \\
& \mapsto 6
\end{align*}
\]

The first thing we do is replace the call by the body of the function with the parameter value substituted in the right places. Following the left to right strategy, we evaluate \(3 = 0\). On the third step we evaluate the `if` expression. Since its condition is false, we step into the `else` branch. Before a function is applied, we need to evaluate its argument (this is called call-by-value, which is opposite to call-by-name). Then we again replace the function call by its body and follow the same strategy. Eventually we reach the final value 6 (*→* indicates that zero or more evaluation steps were performed).

What if the function was in clausal form? Let’s try:

```ml
1 fun fact (0: int): int = 1 
2   | fact n = n * fact(n-1)
```

In the case of function clauses, when a call is evaluated, it steps directly to the corresponding case which matches the pattern:

\[
\begin{align*}
\text{fact}(3) & \mapsto 3 \times \text{fact}(3 - 1) \\
& \mapsto 3 \times \text{fact}(2) \\
& \mapsto 3 \times (2 \times \text{fact}(2 - 1)) \\
& \mapsto 3 \times (2 \times \text{fact}(1)) \\
& \mapsto 3 \times (2 \times (1 \times \text{fact}(1 - 1))) \\
& \mapsto 3 \times (2 \times (1 \times \text{fact}(0))) \\
& \mapsto 3 \times (2 \times (1 \times 1)) \\
& \mapsto 6
\end{align*}
\]

As you can see, evaluation is much quicker and cleaner. Therefore, when we are evaluating programs within our proofs, it will be much easier if those programs are in clausal form as opposed to using `if`s.

Finally we are ready to prove that the two list reverse programs are equivalent! Here they are again:

```ml
fun rev (nil: int list): int list = nil 
| rev (x::l) = (rev l) @ [x];
```
Let's formally state the property we want to prove:

**Property 1.** For every \( l: \text{int list} \), \( \text{rev } l \cong \text{trev } l \).

Since we now have a formal definition of \( \cong \) and we know how to evaluate the programs, we can proceed more confidently. As usual, we will prove Property 1 by induction.

**Proof.** The proof proceeds by structural induction on \( l \). There are two cases:

- **Base case:** \( l = [] \)
  
  To show: \( \text{rev } [] \cong \text{trev } [] \)

  The evaluation of both sides are:
  
  \[
  \text{rev } [] \mapsto [] \\
  \text{trev } [] \mapsto \text{trevh } ([], []) \mapsto []
  \]

  Since \( \cong \) is preserved under \( \mapsto \), we have:
  
  \[
  \text{rev } [] \cong [] \\
  \text{trev } [] \cong []
  \]

  Due to symmetry and transitivity of \( \cong \), we get: \( \text{rev } [] \cong \text{trev } [] \).

- **Inductive case:** \( l = x::l' \)
  
  To show: \( \text{rev } x::l' \cong \text{trev } x::l' \)

  IH: \( \text{rev } l' \cong \text{trev } l' \)

  The evaluations are:
  
  \[
  \text{rev } (x::l') \mapsto (\text{rev } l') \odot [x] \\
  \text{trev } x::l' \mapsto \text{trevh } (x::l', []) \mapsto \text{trevh } (l', [x])
  \]

  Hmm... Looks like we are stuck. The IH can be applied to the result of evaluating \( \text{rev } (x::l') \), but that does not get us anywhere. We need an auxiliary lemma.

  In the proof above, we got stuck because we needed to show that \( (\text{rev } l') \odot [x] \cong \text{trevh } (l', [x]) \). So let’s try to prove a small generalization of this statement:

  **Lemma 1.** For every \( l, a: \text{int list} \), \( (\text{rev } l) \odot a \cong \text{trevh } (l, a) \).

  Obviously the proof proceeds by structural induction, but now we can choose to induce on the list \( l \) or \( a \). What would you choose? Since the function \( \text{trevh} \) is inductive on \( l \), it seems natural to choose this one.

  **Proof.** By structural induction on \( l \). The proof has 2 cases:
• **Base case:** \( l = [] \)
  
  To show: for every \( a \), \((\text{rev} \; []) @ a \) \( \cong \) \( \text{trevh} (\; [], a) \)

  The evaluation of the two sides are:

  \[
  \begin{align*}
  (\text{rev} \; []) @ a & \mapsto [] @ a \\
  \text{trevh} (\; [], a) & \mapsto a
  \end{align*}
  \]

  Due to the definition of the operator @ (see below): \( [] @ a \mapsto a \).
  
  Therefore \((\text{rev} \; []) @ a \cong \text{trevh} (\; [], a)\).

• **Inductive case:** \( l = x :: l' \)
  
  To show: for every \( a \), \((\text{rev} \; x :: l') @ a \) \( \cong \) \( \text{trevh} (x :: l', a) \)

  **IH:** for every \( a \), \((\text{rev} \; l') @ a \) \( \cong \) \( \text{trevh} (l', a) \)

  The evaluations are:

  \[
  \begin{align*}
  (\text{rev} \; x :: l') @ a & \mapsto ((\text{rev} \; l') @ [x]) @ a \\
  \text{trevh} (x :: l', a) & \mapsto \text{trevh} (l', x :: a)
  \end{align*}
  \]

  Using the IH with \( a \) instantiated to \( x :: a \) we have: \( \text{trevh} (l', x :: a) \cong (\text{rev} \; l') @ x :: a \).

  To finish the proof, we need to show that \((\text{rev} \; l') @ [x] @ a \cong (\text{rev} \; l') @ x :: a \).

  \[
  \begin{align*}
  (\text{rev} \; l') @ [x] @ a & \cong (\text{rev} \; l') @ ([x] @ a) \quad \text{[by associativity of @]} \\
  & \cong (\text{rev} \; l') @ x :: a \quad \text{[by definition of @]}
  \end{align*}
  \]

  Here's the definition of append:

  1) `infixr @`

  2) `fun (nil: int list @ l': int list): int list = l'`

  3) `| (x::l) @ l' = x::(l @ l')`

---

In principle we would need to formally show this property as well, but in general this will be given to you as a lemma to use. Be sure to indicate where you use it though!
Extra example

Let’s take a modified (albeit equivalent) version of the append function on lists and prove that append is associative. This is an example of a program equivalence proof on a more complicated code.

```hs
fun app (l1: int list, l2: int list) = let
  val l = app(l1, l2)
  in
  x :: l
  end
```

Theorem 1. For all \(l_1, l_2, l_3 : \text{int list}\), \(\text{app}(l_1, \text{app}(l_2, l_3)) \simeq \text{app}(\text{app}(l_1, l_2), l_3)\)

Proof. The proof proceeds by structural induction on \(l_1\).

- **Base case:** \(l_1 = []\)
  
  Left-hand side evaluation:
  
  \[ \text{app}([], \text{app}(l_2, l_3)) \rightarrow \text{app}(l_2, l_3) \]
  
  Right-hand side evaluation:
  
  \[ \text{app}(\text{app}([], l_2), l_3) \rightarrow \text{app}(l_2, l_3) \]
  
  Because \(\simeq\) is preserved under evaluation, we can conclude:
  
  \[ \text{app}([], \text{app}(l_2, l_3)) \rightarrow \text{app}(l_2, l_3) \]

- **Inductive case:** \(l_1 = x :: l'\)

  To show: \(\text{app}(x :: l', \text{app}(l_2, l_3)) \simeq \text{app}(\text{app}(x :: l', l_2), l_3)\)

  IH: \(\text{app}(l', \text{app}(l_2, l_3)) \simeq \text{app}(\text{app}(l', l_2), l_3)\)

  Left-hand side evaluation:
  
  \[ \text{app}(x :: l', \text{app}(l_2, l_3)) \rightarrow \text{let val l = app(l', l_2)in x :: l end} \]
  
  We can replace the function call to \text{app} in the expression \(x :: l\) because \text{app} is total and will always return a value.

  Right-hand side evaluation:
  
  \[ \text{app}(\text{app}(x :: l', l_2), l_3) \rightarrow \text{let val l = app(app(l', l_2), l_3)in x :: l end} \]

  Since \(\simeq\) is preserved under evaluation, we can conclude:

  \[ \text{app}(x :: l', \text{app}(l_2, l_3)) \simeq x :: \text{app}(l', \text{app}(l_2, l_3)) \]
  
  \[ \text{app}(app(x :: l', l_2), l_3) \simeq x :: \text{app}(\text{app}(l', l_2), l_3) \]

  Using the IH and the fact that \(\simeq\) is a congruence relation, we have:

  \[ x :: \text{app}(\text{app}(l', l_2), l_3) \simeq x :: \text{app}(l', \text{app}(l_2, l_3)) \]

  Therefore, by symmetry and transitivity of \(\simeq\):

  \[ \text{app}(x :: l', \text{app}(l_2, l_3)) \simeq \text{app}(\text{app}(x :: l', l_2), l_3) \]

\(\square\)