Assignment 0:
Judgments, Rules, and Church's λ-Calculus

15-312: Principles of Programming Languages (Fall 2023)

Welcome to 15-312! In this assignment, we will explore foundational concepts in programming languages through the lens of Church's λ-calculus.

In this first assignment, we are introducing the use of *inductive definitions* of *judgments* about terms in the λ-calculus. The λ-calculus is a remarkably simple and elegant model of computation formulated by Church in the early 1930's that is based on the concept of a *mathematical variable*, which is given meaning by *substitution*. λ-terms may be thought of as functions acting on λ-terms, one of which is the λ-term itself—the λ-calculus is inherently self-referential, which is the source of its expressive power. In fact, the λ-calculus can express all functions that can be defined by a Turing machine! It is *expected* that you will find the λ-calculus itself to be somewhat mysterious, mystifying, and non-obvious. Your goal is not to master the λ-calculus as a model of computation, but rather to understand it as a microscopic programming language exhibiting core concepts of central importance.

Although this is the first assignment, it is advisable to take it seriously. This assignment is the "on-ramp" to the course. It involves a lot of unfamiliar concepts and requires good programming skills. We recommend you start early; please don't hesitate to ask for help on Piazza and/or at office hours if you get stuck!

1 Course Mechanics

The purpose of this question is to ensure that you get familiar with this course's policies.

Academic Integrity Go to the Academic Integrity page on the course Canvas to understand the whiteboard policy for collaboration regarding the homework assignments, the late policy regarding timeliness of homework submissions, and the use of Piazza. As in any class, you are responsible for following our collaboration policy; violations will be handled according to university policy.

Submission Please read Appendix A to understand how written and code submissions for this class work.

Task 1.1 (5 pts). Decide whether each of the following statements are consistent with course policies. Briefly explain your answers.

1. Jeanne, Daniel, and Nitin split a pizza while talking about their homework, and by the end of lunch, their pizza box is covered with notes and solutions. Daniel throws out the pizza box and the three go to class, writing their solutions up separately that evening.
2. Mia and Yue write out a solution to Problem 4 on a whiteboard in Wean Hall. Then, they erase the whiteboard and run to Gates. Five minutes later, each student types up the solution on their laptop, sitting at separate tables.
3. Pressed for time, Matthew notices that many of the test cases for an implementation task can be passed with a trivial solution. After submitting his code to the autograder, he is relieved to learn that he is guaranteed at least a 72% score on this task.

2 Judgments and Rules

Throughout the semester, we will define programming languages and other concepts via *judgments*, which can be thought of as mathematical relations. In this section, we will consider simple judgments about lists of natural numbers, where natural numbers are defined by the judgment $\boxed{n\ \mathsf{nat}}$ in **PFPL**, Chapter 2. We will usually abbreviate natural numbers via numeric literals, that is, instead of $\mathtt{s}(\,\mathtt{s}(\,\mathtt{z}\,)\,)$, we simply write 2. First, we define what it means to be a list via the judgment $\boxed{l\ \mathsf{list}}$, which can be thought of as a unary relation.

Judgements in this class are usually defined by induction using *inference rules*. In general, an inference rule is of the following form, where $n \geq 0$:

$$\frac{J_1 \qquad \ldots \qquad J_n}{J}$$

Each judgment J_i is a *premise* (i.e., assumption) and the judgment J is the *conclusion*.

Lists of natural numbers are defined inductively as follows:

1. $\mathtt{nil}$ is a list.
2. If x is a natural number and l is a list, then $\mathtt{cons}(\,x\,;\,l\,)$ is a list.

Using the judgment $\boxed{l\ \mathsf{list}}$ as "l is a list", we may express this definition using inference rules:

$$\frac{}{\mathtt{nil}\ \mathsf{list}}\ \text{L:Nil} \qquad \frac{x\ \mathsf{nat} \qquad l\ \mathsf{list}}{\mathtt{cons}(\,x\,;\,l\,)\ \mathsf{list}}\ \text{L:Cons}$$

The annotations L:Nil and L:Cons are the names of the respective rules. Such names are useful to refer to the rules in the text but are not part of the actual rule.

2.1 Sublists

Using the following inference rules, we may define the judgment $\boxed{l_1 \sqsubseteq l_2}$, intended to mean that list l_1 is an ordered sublist of list l_2.

$$\frac{}{\mathtt{nil} \sqsubseteq \mathtt{nil}}\ \text{SL:Nil} \qquad \frac{l_1 \sqsubseteq l_2}{\mathtt{cons}(\,x\,;\,l_1\,) \sqsubseteq \mathtt{cons}(\,x\,;\,l_2\,)}\ \text{SL:Keep} \qquad \frac{l_1 \sqsubseteq l_2}{l_1 \sqsubseteq \mathtt{cons}(\,x\,;\,l_2\,)}\ \text{SL:Drop}$$

By composing these rules into a tree structure, we may provide a derivation of a particular fact, working from the bottom up and satisfying all premises. For example, we can prove that

$$\mathtt{cons}(\,1\,;\,\mathtt{cons}(\,2\,;\,\mathtt{nil}\,)\,) \sqsubseteq \mathtt{cons}(\,1\,;\,\mathtt{cons}(\,5\,;\,\mathtt{cons}(\,3\,;\,\mathtt{cons}(\,1\,;\,\mathtt{cons}(\,2\,;\,\mathtt{nil}\,)\,)\,)\,)\,)$$

via the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{nil} \sqsubseteq \texttt{nil}}\,\text{SL:Nil}}{\texttt{cons}(2;\texttt{nil}) \sqsubseteq \texttt{cons}(2;\texttt{nil})}\,\text{SL:Keep}}{\texttt{cons}(2;\texttt{nil}) \sqsubseteq \texttt{cons}(1;\texttt{cons}(2;\texttt{nil}))}\,\text{SL:Drop}}{\texttt{cons}(2;\texttt{nil}) \sqsubseteq \texttt{cons}(3;\texttt{cons}(1;\texttt{cons}(2;\texttt{nil})))}\,\text{SL:Drop}}{\texttt{cons}(2;\texttt{nil}) \sqsubseteq \texttt{cons}(5;\texttt{cons}(3;\texttt{cons}(1;\texttt{cons}(2;\texttt{nil}))))}\,\text{SL:Drop}}{\texttt{cons}(1;\texttt{cons}(2;\texttt{nil})) \sqsubseteq \texttt{cons}(1;\texttt{cons}(5;\texttt{cons}(3;\texttt{cons}(1;\texttt{cons}(2;\texttt{nil})))))}\,\text{SL:Keep}$$

Notice that no judgments are left unjustified, as there are no premises for rule SL:Nil. We call such a rule an *axiom* or a leaf rule.

Task 2.1 (10 pts). Give a *different* derivation tree showing the same judgment.

You may wish to modify our given LaTeX code, which can be found in `written/assn0.tex`.

To prove theorems about our judgments, we use a technique called *rule induction*. Rule induction is a generalization of structural induction, allowing us to consider all derivation trees for a particular judgment. If we go by induction over the judgment defining a data structure, such as $\boxed{l\ \mathsf{list}}$, rule induction is in fact just structural induction.

Theorem 2.1. *If* l list, *then* $l \sqsubseteq l$.

Task 2.2 (10 pts). Prove Theorem 2.1 by rule induction on l list (i.e., by structural induction on l). As a reminder, in each case, you should argue for the existence of a derivation showing $l \sqsubseteq l$.

This theorem is fairly straightforward and involves a familiar proof technique, going by cases on how a list was construced. However, thanks to the generality of rule induction, we can go by cases on how *any* judgment was derived. For example, we can go by induction on a derivation tree for a sublist judgment, considering all possible cases for how this derivation tree could have been constructed. We consider a worked example here:

Theorem 2.2. *If* $l_1 \sqsubseteq l_2$ *and* $l_2 \sqsubseteq l_3$, *then* $l_1 \sqsubseteq l_3$.

We provide the following proof, annotated with footnotes.

Proof. Assume $l_1 \sqsubseteq l_2$ and $l_2 \sqsubseteq l_3$.[1][2] We prove by rule induction on $l_2 \sqsubseteq l_3$, showing in each case that $l_1 \sqsubseteq l_3$.[3] The cases we have to consider correspond to all rules that could have been used at the root of the derivation tree of the judgment $l_2 \sqsubseteq l_3$.

[1] The structure of a proof should follow the structure of the theorem. Since the theorem is of the form "if P then Q", we should assume P and show Q.

[2] By assuming $l_1 \sqsubseteq l_2$ and $l_2 \sqsubseteq l_3$, we are really assuming that we have derivation trees for both $l_1 \sqsubseteq l_2$ and $l_2 \sqsubseteq l_3$.

[3] In general, if we have multiple assumptions, which one(s) we should induct on, and in what order we should induct on them, depends on the proof itself.

Case $\dfrac{}{\texttt{nil} \sqsubseteq \texttt{nil}}$ SL:Nil :

Here, we know $l_2 = \texttt{nil}$ and $l_3 = \texttt{nil}$. It remains to show that $l_1 \sqsubseteq l_3$; in other words, $l_1 \sqsubseteq \texttt{nil}$. By assumption, $l_1 \sqsubseteq l_2$. Since we know $l_2 = \texttt{nil}$, we have $l_1 \sqsubseteq \texttt{nil}$, which is precisely what we needed to show.

Case $\dfrac{l_2' \sqsubseteq l_3'}{\texttt{cons}(\, x \,;\, l_2' \,) \sqsubseteq \texttt{cons}(\, x \,;\, l_3' \,)}$ SL:Keep :[4]

Here, we know $l_2 = \texttt{cons}(\, x \,;\, l_2' \,)$ and $l_3 = \texttt{cons}(\, x \,;\, l_3' \,)$. Observe that by the premise, we know $l_2' \sqsubseteq l_3'$. It remains to show that $l_1 \sqsubseteq l_3$; in other words, $l_1 \sqsubseteq \texttt{cons}(\, x \,;\, l_3' \,)$. By assumption, $l_1 \sqsubseteq l_2$, so $l_1 \sqsubseteq \texttt{cons}(\, x \,;\, l_2' \,)$.[5] We go by rule induction on $l_1 \sqsubseteq \texttt{cons}(\, x \,;\, l_2' \,)$.

Case $\dfrac{}{\texttt{nil} \sqsubseteq \texttt{nil}}$ SL:Nil :

Here, we know $l_1 = \texttt{nil}$ and $\texttt{cons}(\, x \,;\, l_2' \,) = \texttt{nil}$. However, $\texttt{cons}(\, x \,;\, l_2' \,)$ and $\texttt{nil}$ are different, so this case is *vacuous*.[6]

Case $\dfrac{l_1' \sqsubseteq l_2'}{\texttt{cons}(\, x \,;\, l_1' \,) \sqsubseteq \texttt{cons}(\, x \,;\, l_2' \,)}$ SL:Keep :

Here, we know $l_1 = \texttt{cons}(\, x \,;\, l_1' \,)$ (and $l_2 = \texttt{cons}(\, x \,;\, l_2' \,)$, which we already knew). Observe that by the premise, we must know $l_1' \sqsubseteq l_2'$. Since we know $l_1' \sqsubseteq l_2'$ and $l_2' \sqsubseteq l_3'$ via sub-derivations, we may invoke the *inductive hypothesis* to get that $l_1' \sqsubseteq l_3'$. To show $l_1 \sqsubseteq l_3$, i.e. $\texttt{cons}(\, x \,;\, l_1' \,) \sqsubseteq \texttt{cons}(\, x \,;\, l_3' \,)$, we may use the rule SL:Keep:

$$\frac{l_1' \sqsubseteq l_3'}{\texttt{cons}(\, x \,;\, l_1' \,) \sqsubseteq \texttt{cons}(\, x \,;\, l_3' \,)} \text{ SL:Keep}$$

The premise is justified by our previous reasoning.

Case $\dfrac{l_1 \sqsubseteq l_2'}{l_1 \sqsubseteq \texttt{cons}(\, x \,;\, l_2' \,)}$ SL:Drop :

Observe that by the premise, we must know $l_1 \sqsubseteq l_2'$. Since we know that $l_1 \sqsubseteq l_2'$ and $l_2' \sqsubseteq l_3'$ via sub-derivations, we may invoke the inductive hypothesis to get that $l_1 \sqsubseteq l_3'$. To show $l_1 \sqsubseteq l_3$, i.e. $l_1 \sqsubseteq \texttt{cons}(\, x \,;\, l_3' \,)$, we may use the rule SL:Drop:

$$\frac{l_1 \sqsubseteq l_3'}{l_1 \sqsubseteq \texttt{cons}(\, x \,;\, l_3' \,)} \text{ SL:Drop}$$

The premise is justified by our previous reasoning.

[4] Observe that we may rename variables in the rule as we wish.

[5] Notice that here, there is no immediate reasoning to do; it depends on how we know $l_1 \sqsubseteq \texttt{cons}(\, x \,;\, l_2' \,)$. Therefore, we consider nested rule induction, breaking down our derivation of $l_1 \sqsubseteq \texttt{cons}(\, x \,;\, l_2' \,)$.

[6] As a reminder, we say a case is vacuous when it is immediately contradictory for structural reasons. In this example, we clearly explained the vacuousness, but in your proofs, you may simply state that a particular case (or all other cases not considered) are vacuous, so long as it is true.

In all three sub-cases, we showed that $l_1 \sqsubseteq l_3$, as desired, so this case holds.

Case $\dfrac{l_2 \sqsubseteq l_3'}{l_2 \sqsubseteq \texttt{cons}(\, x \,;\, l_3' \,)}$ SL:Drop :

Here, we know $l_3 = \texttt{cons}(\, x \,;\, l_3' \,)$. Observe that by the premise, we must know $l_2 \sqsubseteq l_3'$. It remains to show that $l_1 \sqsubseteq l_3$; in other words, $l_1 \sqsubseteq \texttt{cons}(\, x \,;\, l_3' \,)$. Since we assumed $l_1 \sqsubseteq l_2$ and we have a sub-derivation showing $l_2 \sqsubseteq l_3'$, we may invoke the inductive hypothesis to get that $l_1 \sqsubseteq l_3'$. To show $l_1 \sqsubseteq \texttt{cons}(\, x \,;\, l_3' \,)$, we may use the rule SL:Drop:

$$\frac{l_1 \sqsubseteq l_3'}{l_1 \sqsubseteq \texttt{cons}(\, x \,;\, l_3' \,)}\ \text{SL:Drop}$$

The premise is justified by our previous reasoning.

In all three cases, we showed that $l_1 \sqsubseteq l_3$, as desired. Thus, the claim holds. □

In this proof, we were fairly verbose about justifying the reasoning. You need not be this verbose in the proofs you submit, although you should be just as careful in structuring your proofs.

A close look at our proof of Theorem 2.1 reveals that it is *constructive*. A constructive proof does not only show that an object exists but also how to construct it. Almost all the proofs in this course are constructive. In our case, the proof shows how to construct a derivation of the judgment $l_1 \sqsubseteq l_3$ from derivations of the judgments $l_1 \sqsubseteq l_2$ and $l_2 \sqsubseteq l_3$. Consider for example the case of SL:Drop in the proof. In this case, the root of the derivation tree of $l_2 \sqsubseteq l_3$ looks as follows.

$$\frac{\begin{array}{c}\mathbf{T_1}\\ l_2 \sqsubseteq l_3'\end{array}}{l_2 \sqsubseteq \texttt{cons}(\, x \,;\, l_3' \,)}\ \text{SL:Drop}$$

In particular, $l_3 = \texttt{cons}(\, x \,;\, l_3' \,)$ and we have a derivation tree $\mathbf{T_1}$ for $l_2 \sqsubseteq l_3'$. The proof then shows how to construct a derivation tree for $l_1 \sqsubseteq l_3$. First, per induction hypothesis, the proof recursively shows us how to construct a derivation tree $\mathbf{T_2}$ for $l_1 \sqsubseteq l_3'$ (using $\mathbf{T_1}$ and the derivation tree for $l_1 \sqsubseteq l_2$). Then we can apply the rule SL:Drop to get a derivation tree for $l_1 \sqsubseteq l_3$:

$$\frac{\dfrac{\mathbf{T_2}}{l_1 \sqsubseteq l_3'}}{l_1 \sqsubseteq \texttt{cons}(\, x \,;\, l_3' \,)}\ \text{SL:Drop}$$

Task 2.3 (10 pts). Consider the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\texttt{nil} \sqsubseteq \texttt{nil}}\ \text{SL:Nil}}{\texttt{nil} \sqsubseteq \texttt{cons}(\, 2 \,;\, \texttt{nil} \,)}\ \text{SL:Drop}}{\texttt{cons}(\, 1 \,;\, \texttt{nil} \,) \sqsubseteq \texttt{cons}(\, 1 \,;\, \texttt{cons}(\, 2 \,;\, \texttt{nil} \,) \,)}\ \text{SL:Keep}$$

Recall our earlier derivation tree, as well:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{nil} \sqsubseteq \texttt{nil}}\,\textsc{SL:Nil}}{\texttt{cons}(2;\texttt{nil}) \sqsubseteq \texttt{cons}(2;\texttt{nil})}\,\textsc{SL:Keep}}{\texttt{cons}(2;\texttt{nil}) \sqsubseteq \texttt{cons}(1;\texttt{cons}(2;\texttt{nil}))}\,\textsc{SL:Drop}}{\texttt{cons}(2;\texttt{nil}) \sqsubseteq \texttt{cons}(3;\texttt{cons}(1;\texttt{cons}(2;\texttt{nil})))}\,\textsc{SL:Drop}}{\texttt{cons}(2;\texttt{nil}) \sqsubseteq \texttt{cons}(5;\texttt{cons}(3;\texttt{cons}(1;\texttt{cons}(2;\texttt{nil}))))}\,\textsc{SL:Drop}}{\texttt{cons}(1;\texttt{cons}(2;\texttt{nil})) \sqsubseteq \texttt{cons}(1;\texttt{cons}(5;\texttt{cons}(3;\texttt{cons}(1;\texttt{cons}(2;\texttt{nil})))))}\,\textsc{SL:Keep}$$

We abbreviate:

$$\begin{aligned} l_1 &= \texttt{cons}(1;\texttt{nil}) \\ l_2 &= \texttt{cons}(1;\texttt{cons}(2;\texttt{nil})) \\ l_3 &= \texttt{cons}(1;\texttt{cons}(5;\texttt{cons}(3;\texttt{cons}(1;\texttt{cons}(2;\texttt{nil}))))) \end{aligned}$$

Combined, these trees show that $l_1 \sqsubseteq l_2$ and $l_2 \sqsubseteq l_3$. Using these two derivations, what derivation of $l_1 \sqsubseteq l_3$ does our proof of Theorem 2.2 find? Briefly (via a few sentences) explain your reasoning.

Hint. There are two different ways to derive $l_1 \sqsubseteq l_3$; be careful to use the derivation of $l_2 \sqsubseteq l_3$ above, rather than the one you constructed in a previous task.

2.2 List Containment

Now, consider another judgment, $\boxed{x \in l}$, expressing that x is an element of list l:

$$\dfrac{}{x \in \texttt{cons}(x;l)}\,\textsc{In:Here} \qquad \dfrac{x \in l}{x \in \texttt{cons}(y;l)}\,\textsc{In:There}$$

The following theorem relates the sublist judgment to the containment judgment:

Theorem 2.3. *If $l_1 \sqsubseteq l_2$ and $x \in l_1$, then $x \in l_2$.*

Task 2.4 (25 pts). Prove Theorem 2.3 by rule induction.

Hint. Go by rule induction on $l_1 \sqsubseteq l_2$. Some cases may be vacuous. In some cases, you will need to use an inner rule induction on $x \in l_1$.

3 Binding and Scope

3.1 Free Variables

Recall the definition of the judgment $x \in \mathsf{FV}(M)$ from the λ-calculus supplement. We could define $x \notin \mathsf{FV}(M)$ if not $x \in \mathsf{FV}(M)$. However, we can also define the judgment $x \notin \mathsf{FV}(M)$ inductively without using $x \in \mathsf{FV}(M)$.

Task 3.1 (20 pts). Recall the definition of the judgment $x \in \mathsf{FV}(M)$ from the **PFPL** Supplement.

1. Give an inductive definition of the judgment $x \notin \mathsf{FV}(M)$ by exhibiting a selection of rules, stating positively that x is not-free in M.
   Be careful!
   Do *not* define the judgment $x \notin \mathsf{FV}(M)$ to be the logical negation of $x \in \mathsf{FV}(M)$, as they are two separately defined judgments. As a matter of fact, you're **not allowed** to refer to the judgment $x \in \mathsf{FV}(M)$ in your rules. This idea of logical complementation accurately specifies the behavior of the judgment, but doesn't inform its implementation.

   As such, restated in terms of this semantic specification, define a judgment $x \notin \mathsf{FV}(M)$ such that:

   (a) For all x, M, $x \notin \mathsf{FV}(M)$ is derivable if $x \in \mathsf{FV}(M)$ is not derivable

   (b) For all x, M, $x \notin \mathsf{FV}(M)$ is not derivable if $x \in \mathsf{FV}(M)$ is derivable

2. Prove that if $x\,\mathsf{var}$ and $M\,\mathsf{tm}$ either $x \in \mathsf{FV}(M)$ or $x \notin \mathsf{FV}(M)$. In other words, prove that **at least** one of these two statements is true. You are not required to prove that **exactly** one of them is true, although your definition of the judgment $x \notin \mathsf{FV}(M)$ should satisfy that. Proceed by rule induction on $M\,\mathsf{tm}$, making use of the fact that if $x\,\mathsf{var}$ and $y\,\mathsf{var}$, then either $x = y$ or $x \neq y$.

3.2 Substitution

Recall the definition of $\{M/x\}N = P$ from the **PFPL** Supplement. Unfortunately, $\{M/x\}N = P$ is undefined for certain terms M and N, where the substitution could capture variables. To avoid this issue, we have introduced α-equivalence to make substitution a total operation that is defined for all terms.

Theorem 3.1. *For all $M\,\mathsf{tm}$, $x\,\mathsf{var}$, and $N\,\mathsf{tm}$, there exists $N'\,\mathsf{tm}$ and $P'\,\mathsf{tm}$ such that $N' \equiv_\alpha N$ and $\{M/x\}N' = P'$.*[7]

Task 3.2 (10 pts). Give some M, N, N', and P' such that:

1. there does not exist a P such that $\{M/x\}N = P$

[7] Moreover, it is the case that if $N'' \equiv_\alpha N'$ and $\{M/x\}N'' = P''$, then $P'' \equiv_\alpha P'$.

2. $N' \equiv_\alpha N$
3. $\{M/x\}N' = P'$

Briefly explain why these properties hold for your solution.

4 Implementing the λ-Calculus

We have observed that it is convenient to work up to α-equivalence, ignoring the names of bound variables. In this section, we will build a tool in Standard ML for representing α-equivalence classes of λ-terms.[8]

The tool you will build can be generalized to support arbitrary *abstract binding trees* (ABTs) as defined in **PFPL**, Chapter 1. By implementing a language as simple as the untyped λ-calculus (abbreviated as ULC in the code), you will grapple with fundamental ideas; then, on the rest of the assignments, we will provide you with analogous ABT structures for more complex languages.

4.1 Variables

A variable in a λ-term is a placeholder for a λ-term, which can be plugged in for the variable using substitution. To work with variables, we introduce the `TEMP` signature, partially reproduced below. Here, "temp" stands for "temporary", since the exact identity of a variable is only kept until the variable is bound, as we will see in the next section.

`var/temp.sig`

```
signature TEMP =
sig
  type t

  (* Creates a new, globally unique temp. *)
  val new: string -> t

  (* Tests whether two temps are equal. *)
  val equal: t * t -> bool

  (* Provides a string representation of the globally unique temp.
     *)
  val toString: t -> string
  structure Dict: DICT where type key = t
end
```


Values of type `t` represent variables. A key idea is the notion of a *new* or *fresh variable* – one that has not been seen before in some given context. There are several ways to guarantee freshness relative to a context, but one simple way is to just ignore the context and make sure every freshly generated variable is globally unique. This is achieved by using global state behind the scenes. As regards your ABT implementation, this shows up in the above signature as `val new : string -> t`, where the `string` value being passed in is entirely up to you (the user), as it's primarily there for human readability.

[8] If you are ambitious and have the time, it is a very good exercise to try to implement abstract binding trees yourself, without the benefit of any theoretical understanding or the techniques suggested here, and watch yourself suffer and fail miserably.

For example, if `Var : TEMP`, then `Var.toString (Var.new "x")` might give `"x1"`, whereas running it again would give `"x2"` and running with `"y"` would give `"y3"`.

You will only need to work with an abstract structure ascribing to the signature.

4.2 α-Equivalence Classes of λ-Terms

Now, we introduce the signature for λ-terms. This is a version of an interface that has been developed here at CMU over numerous compiler development efforts, designed to help users avoid common errors.

`lang-ulc/ulc.abt.sig`

```
signature ULC =
sig
  structure Term:
  sig
    type termVar
    type term
    type t = term

    structure Var: TEMP where type t = termVar

    datatype view = Var of termVar | Lam of termVar * term | Ap of
       term * term

    val Var': termVar -> term
    val Lam': termVar * term -> term
    val Ap': term * term -> term

    val into: view -> term
    val out: term -> view
    val aequiv: term * term -> bool
    val toString: term -> string

    val subst: term -> termVar -> term -> term
  end
end
```


First, observe that `ULC` contains a single sub-structure, `Term`. In languages with more than one sort (e.g., a sort for expressions and a sort for types), we will have one sub-structure per sort.

4.2.1 Variables

In the `Term` structure, we have `structure Var : TEMP`; this will provide us with all the necessary operations on variables. We also have `type termVar`, synonymous with `Var.t`.

4.2.2 Terms and Views

Next, we introduce `type term` and `type t`, which are synonymous.[9] Values of type `Term.t` will be (α-equivalence classes of) λ-terms. Importantly, `type t` is *abstract*: its representation will be internal to the structure.

Access to the representation type `t` is mediated by a *view*, which is a *non-recursive* datatype `view` with a pair of functions `into` and `out`. The idea behind a view is that it is a type whose values represent a one-step unfolding of a value of the abstract type `t`. The function `out` does this one-step unfolding of a λ-term to create a view, and `into` puts a view back together into a λ-term.

The function `out` has type `term -> view`, taking a genuine λ-term and makes it visible to the user. Let `m : term`; we describe the behavior of `out m` as follows:

1. Suppose `m` represents x, where x is a variable. Then, `out m` would return `Var x`, where the value `x : Variable.t` represents the variable x.
2. Suppose `m` represents the lambda abstraction $\lambda(x\,.\,M')$. Then, `out m` would return `Lam (y, m')`, where `y` represents a fresh variable y and `m'` represents the λ-term $\{y/x\}M$, which is well-defined because y is (globally) fresh.

   In other words, the `out` operation renames the variable bound by an abstraction when unpacking it! This is one of the essential features of this interface, and implementing this correctly will save you many hours of tracking down bugs involving binding and scope.
3. Suppose `m` represents $\mathtt{ap}(M_1\,;M_2)$. Then, `out m` should return `Ap (m1, m2)`, where `m1` represents the λ-term M_1 and `m2` represents the λ-term M_2. Observe that `m1` and `m2` are of type `term`, *not* of type `view`. This is what we mean by "one-step unfolding".

The function `into` has type `view -> term`, folding a one-step unfolding (that is, a view) back into a λ-term.

1. Suppose `x : Variable.t` represents the variable x. Then, `into (Var x)` will return an internal form representing the variable term x.
2. Suppose `x : Variable.t` represents the variable x and `m : term` represents the λ-term M. Then, `into (Lam (x, m))` will return an internal form representing the λ-term $\lambda(x\,.\,M)$.
3. Suppose `m1 : term` represents the λ-term M_1 and `m2` represents the λ-term M_2. Then, `into (Ap (m1, m2))` will return an internal form representing the λ-term $\mathtt{ap}(M_1\,;M_2)$.

For convenience, we provide functions which simply compose constructors with `into`:

```
val Var' = into o Var
val Lam' = into o Lam
val Ap' = into o Ap
```

[9] In SML, it is common practice to define `type foo` and `type t = foo` in `structure Foo`, so the signature can use `foo` while clients of the structure may use `Foo.t`.

Task 4.1 (5 pts). Suppose `v` is a value of type `view`. Will `out (into v)` be *equal to* `v`? (Here, equality does *not* mean α-equivalence.) If so, explain why; if not, give a simple counterexample. Think carefully about what effect the renaming `out` can perform before answering this question.

4.2.3 Auxiliary Functions

We can test α-equivalence with the `aequiv` function, which returns `true` if its two arguments are α-equivalent, and `false` otherwise.

Additionally, we can use the `subst` function to substitute λ-terms in for free variables. Suppose `m1` represents M_1, `x` represents x, and `m` represents M. Then, `subst m1 x m` will return $\{M_1/x\}M$, which by Theorem 3.1 must be defined since `m` represents an α-equivalence class of terms.

Finally, we have a convenience function `toString : term -> string` for producing a string representation of a term. Note that `toString` uses `out`, so the variables in the string representation will be different on each call of `toString`.

4.3 Implementing ABTs

One of the inconveniences of a naive representation of λ-terms is that α-equivalent terms can have multiple representations, so implementing `aequiv` and other helper functions becomes tricky. We will look at a more sophisticated representation, called *locally nameless form* or *de Bruijn indexed* form, which avoids this problem, so that each α-equivalence class is represented with a single data value, and thus α-equivalence can be tested for with a simple structural traversal.

The main idea is to observe that variables in a λ-term serve two roles.

1. First, they can appear free — that is, they can be a *name* for a yet-to-be-determined term.
2. Second, they can appear bound, in which case the variable occurrences simply *refer back* to the location of the λ abstraction.

For example, consider the following λ-term:

$$\lambda(\, x \,.\, \lambda(\, y \,.\, \mathtt{ap}(\, x \,;\, \mathtt{ap}(\, y \,;\, z \,)\,)\,)\,)$$

The use of x refers back to the variable from the outer λ abstraction; similarly, the use of y refers back to the variable from the inner λ-abstraction. However, z is a free variable. The only reason we need the names of the bound variables x and y is to distinguish one abstraction site from another; the names themselves are irrelevant from a semantic point-of-view. (This is just another way of saying that we want to identify terms up to α-equivalence, of course.)

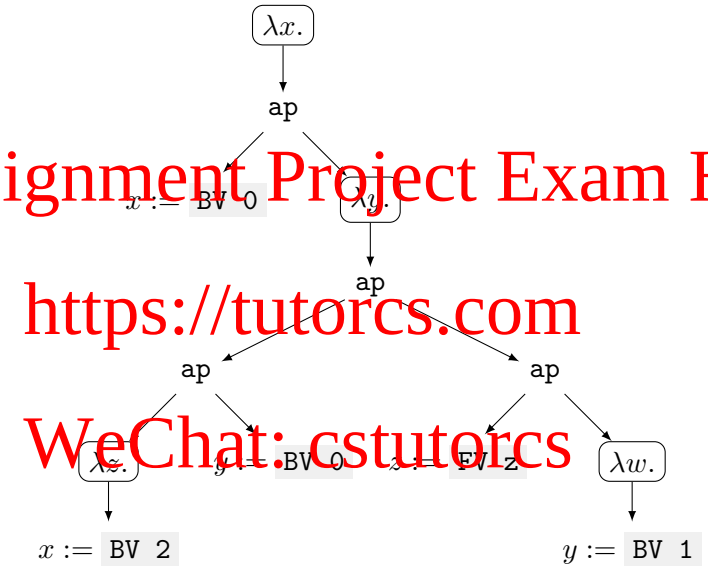
In a locally nameless representation, we distinguish these two roles in our data structure, in order to make α-equivalence implementable as structural equality on terms. The trick in this representation is to exploit a structural invariant of λ-terms: in any λ abstraction $\lambda(\, x \,.\, M\,)$, the only occurrences of x are within M, up to α-equivalence.

As a result of this fact, we have a unique path "upward" from each bound variable to the abstractor that bound it. We can represent this path as a number that tells us how many binders we have to hop over before we reach the one we're interested in. Thus, we define `type term` as follows:

```
datatype term
  = BV of int
  | FV of Variable.t
  | LAM of term
  | AP of term * term
```

Here, `BV` stands for "bound variable" and stores the (non-negative) number of binders between the variable and the binding site, and `FV` stands for "free variable" and stores a genuine variable. Notice the difference between `datatype term` (recursive) and `datatype view` (non-recursive).

Consider the following diagram of $\lambda x.x(\,\lambda y.(\,\lambda z.x\,)(\,y\,)(\,z(\,\lambda w.y\,)\,)\,)$.



We've put a box around every λ abstraction and labeled each bound variable with its bound variable number. We can calculate the bound variable number by looking at each path from an abstraction to its use sites and count the number of abstractions crossed along the way:

Path	Variable #
$\boxed{\lambda x.} \to \texttt{ap} \to x$	0
$\boxed{\lambda x.} \to \texttt{ap} \to \boxed{\lambda y.} \to \texttt{ap} \to \texttt{ap} \to \boxed{\lambda z.} \to x$	2
$\boxed{\lambda y.} \to \texttt{ap} \to \texttt{ap} \to y$	0
$\boxed{\lambda y.} \to \texttt{ap} \to \texttt{ap} \to \boxed{\lambda w.} \to y$	1

An important fact to notice about these paths is that even for the same binder, each occurrence of its bound variable can have a *different* bound variable number, depending on the number of abstractions we crossed over to reach that variable occurrence.

Task 4.2 (40 pts). Implement the rest of `structure ULC` in `lang-ulc/ulc.abt.sml` using a

locally nameless representation. Specifically, you should implement `into`, `out`, `aequiv`, and `subst` according to the specifications above.

Hint. When implementing this structure, you will find it helpful to define two recursive helper functions `bind` (for `into`) and `unbind` (for `out`).

- `bind` should take a variable `x : Variable.t`, some `i : int`, and a term `m : term`, returning an updated version of `m` with all (free) occurrences of `x` bound, assuming the abstraction site is `i` layers away.
- `unbind` should take a variable `x : Variable.t`, some `i : int`, and a term `m : term`, returning an updated version of `m` with bound variable `i` layers away replaced with free variable `x`.

Hint. Note that one of the invariants of the locally nameless representation is that the case `BV n` only occurs once you've gone beneath a binder. So, it can't happen at the top level of a term.

Hint. Your implementations of `aequiv` and `subst` should be fairly simple, since they need not worry about variable capture due to the locally nameless representation.

Testing

To test your code, you can load `lang-ulc/sources.cm`:


```
smlnj -m lang-ulc/sources.cm
- structure Term = ULC.Term;
- val x = Term.Var.new "x";
val x = - : Term.Var.t
- val i = Term.Lam' (x, Term.Var' x);
val i = - : ?.ULC.term
- Term.toString i;
val it = "(Lam (x2, x2))" : string
- Term.toString i;
val it = "(Lam (x3, x3))" : string
- Term.aequiv (i, i);
val it = true : bool
- Term.aequiv (i, Term.Lam' (x, Term.Var' (Term.Var.new "x")));
val it = false : bool
- val y = Term.Var.new "y";
val y = - : Term.Var.t
- Term.toString (Term.subst (Term.Lam' (x, Term.Var' y)) x
    (Term.Ap' (Term.Var' x, Term.Var' x)));
val it = "(Ap ((Lam (x6, y5)), (Lam (x7, y5))))" : string
```

5 β-Equivalence: Calculation in the λ-Calculus

In the λ-calculus, β-equivalence tells us how to *compute*! In this section, you will code in the λ-calculus and implement a simple interpreter based on β-equivalence. Note that you may attempt the following subsections in either order, depending on your preference.

5.1 Church Encodings

At first glance, it would seem that the λ-calculus is too simple to be useful. While it has (higher-order) functions, what about data structures like booleans? Surprisingly, booleans (and all other data structures!) need not be included as primitives, since as we saw in lecture, they can be defined inside the λ-calculus.

5.1.1 Booleans

Consider the following encoding of booleans:

$$\begin{aligned}
\texttt{true} &\triangleq \lambda t.\lambda f.t \\
\texttt{false} &\triangleq \lambda t.\lambda f.f \\
\texttt{if}(M\,;M_1\,;M_0) &\triangleq M(M_1)(M_0)
\end{aligned}$$

Here, $\texttt{if}(\,M\,;M_1\,;M_0\,)$ is the ABT form of the concrete syntax $\texttt{if}\,M\,\texttt{then}\,M_1\,\texttt{else}\,M_0$. Observe that the booleans themselves are "active" data: given a "then" branch M_1 and an "else" branch M_0, the boolean itself is an algorithm that picks between them.

How do we argue this encoding is correct? Our specification of booleans should require that:

$$\begin{aligned}
\texttt{if}(\texttt{true}\,;M_1\,;M_0) &\equiv_\beta M_1 \\
\texttt{if}(\texttt{false}\,;M_1\,;M_0) &\equiv_\beta M_0
\end{aligned}$$

We can validate these laws using the axioms of $\equiv_\beta$, stated in the **PFPL** Supplement; we give the following derivations as a chain of β-equivalences, since $\equiv_\beta$ is postulated to be reflexive and transitive.

$$\begin{aligned}
\texttt{if}(\texttt{true}\,;M_1\,;M_0) &\triangleq \texttt{true}(M_1)(M_0) \\
&\triangleq (\lambda t.\lambda f.t)(M_1)(M_0) \\
&\equiv_\beta (\lambda f.M_1)(M_0) \\
&\equiv_\beta M_1 \\
\texttt{if}(\texttt{false}\,;M_1\,;M_0) &\triangleq \texttt{false}(M_1)(M_0) \\
&\triangleq (\lambda t.\lambda f.f)(M_1)(M_0) \\
&\equiv_\beta (\lambda f.f)(M_0) \\
&\equiv_\beta M_0
\end{aligned}$$

5.1.2 Pairs

We can also define pairs in the λ-calculus:

$$\langle M_1, M_2 \rangle \triangleq \lambda k.k(\,M_1\,)(\,M_2\,)$$
$$M \cdot \texttt{l} \triangleq M(\,\lambda x_1.\lambda x_2.x_1\,)$$
$$M \cdot \texttt{r} \triangleq M(\,\lambda x_1.\lambda x_2.x_2\,)$$

Here, $M \cdot \texttt{l}/M \cdot \texttt{r}$ mean "get the left/right component of pair M".

To validate this encoding, we specify that the following equivalences should hold:

$$\langle M_1, M_2 \rangle \cdot \texttt{l} \equiv_\beta M_1$$
$$\langle M_1, M_2 \rangle \cdot \texttt{r} \equiv_\beta M_2$$

Task 5.1 (10 pts). Validate the encoding of pairs by giving a derivation of the given equivalences via chains of β-equivalences. Please make sure to justify each β-equivalence step by citing the corresponding rule (as presented in the λ-calculus supplement).

5.1.3 Options

Now, it's your turn!

Task 5.2 (15 pts). In `tests/option.lc`, define the following operations:

$$\texttt{none} \triangleq \ldots$$
$$\texttt{some}(\,M\,) \triangleq \ldots$$
$$\texttt{case}(\,M\,;M_0\,;M_1\,) \triangleq \ldots$$

Here, $\texttt{case}(\,M\,;M_0\,;M_1\,)$ is like the SML code `case M of NONE => M0 | SOME x => M1 x`. These operations should satisfy the following laws, although you need not prove it:

$$\texttt{case}(\,\texttt{none}\,;M_0\,;M_1\,) \equiv_\beta M_0$$
$$\texttt{case}(\,\texttt{some}(\,M\,)\,;M_0\,;M_1\,) \equiv_\beta M_1(\,M\,)$$

Hint. Options are very similar to booleans, except one of the forms contains data M.

You can find the encodings of booleans and pairs in `tests/bool.lc` and `tests/pair.lc` for reference. Feel free to write a `λ` or a `\` for a λ-abstraction; as described in Section 5.3, they are interchangeable. For information on testing your solution, consult Section 5.3.

5.2 Infinite Jest

A λ-term N is called a *β-normal form* if there is no N' such that $N \succ_\beta N'$; we write this as $N \not\succ_\beta$. Somewhat surprisingly, not all λ-terms have normal forms.

Task 5.3 (10 pts). Show that there is a λ-term M *without* a normal form. Concretely, give some M such that there is no N such that $M \succ_\beta N$ and $N \not\succ_\beta$. Define this term M as `nonormal` in `tests/no-normal.lc` and briefly explain in a comment why it doesn't have a normal form.

Hint. Consider a situation in which a chosen λ-term is *applied to itself* in such a way that the self-application reduces to itself, and can thus be so-reduced forever.

5.3 β-Normalization

In this section, you will implement an algorithm for β-normalizing λ-terms, using the infrastructure you developed in Section 4.

Task 5.4 (30 pts). Implement normalization for the λ-calculus, as defined in the **PFPL** Supplement, in the file `lang-ulc/normalize-ulc.fun`. In particular, if `m : ULC.Term.t` represents M, evaluating `norm m` should attempt to find some `n` representing N such that M norm N.

Hint. You will almost certainly want to implement a helper function `whnf` to perform head normalization.

Running Your Code

To test your normalizer, you can use our custom λ-calculus REPL, which supports several commands useful for interacting with the λ-calculus:

```
smlnj -m top/top.cm
- TopLevel.repl ();
->def i = λx.x;
->def s = λx.λy.λz.x z (y z);
->print norm s i i;
λz. z z
->assert! i i = i;
```

See the `tests/tests.lc` file for more details and examples. To return to the SML REPL from the λ-calculus REPL, hit Ctrl-C.

You can evaluate a file using `TopLevel.evalFile "<filename>.lc";`.

Using the Heap Image If you have not yet finished your implementation of `structure ULC` from Section 4, you may use the distributed heap image as described in Appendix B.1 to test with a reference implementation of `structure ULC`.

```
smlnj @SMLload refsol
- use "lang-ulc/normalize-ulc.fun";
- use "top/top.sml";
```

Note that you *must* reload the `top/top.sml` file every time you refresh the normalizer.

Unicode Note that by default, the REPL will pretty-print a Unicode `λ` symbol, and entering `λ` symbols into the REPL should also be supported alongside the `\` character. If your terminal emulator is not sufficiently advanced to display the symbol, you may switch to ASCII-only mode by replacing `smlnj` with `smlnj -Dascii=1`. You may also use the `tests/*_ascii.lc` files if your text editor has issues with the symbol.

A Submitting Your Solutions

A.1 Written Solutions

First, write your solutions and build your written solution PDF. We require that you use the provided LaTeX template in the `written/` directory to at least organize your solutions. However, if your handwriting is neat and clear, you are welcome to include images of handwritten work in the LaTeX template.

If you are not familiar with LaTeX, there are lots of great introductory resources online; you may also find it useful to look at the `written/assn0.tex` file to understand how we typeset parts of the writeup. You are encouraged to ask for help on Piazza, as well!

Here are some tips for using LaTeX:

- You can upload the contents of the `written/` folder to a private Overleaf project if you wish to write and build your solutions online.
- If you wish to develop locally, you may find it productive to use a LaTeX editor, such as the TeX Workshop extension for VS Code.
- To build from the terminal, run `make hw00.pdf` in the `written/` directory to build your solutions.

  If there were LaTeX issues with the `make` command, you are likely missing some `.sty` files.

  – If you are using your own Ubuntu computer, executing the following should be sufficient:

  ```
  sudo apt update && sudo apt install texlive-latex-extra texlive-science
  ```

  – If you wish to use the Andrew machines, these packages are on the available Ubuntu machines.

  ```
  ssh andrewid@ubuntu.andrew.cmu.edu
  ```


Submit your PDF to Gradescope, matching the pages as indicated.

A.2 Code Solutions

Build a code submission zipfile by running `make`. Then, submit the produced `handin.zip` to Gradescope.

Gradescope will be set up to run some test cases on your submission. However, a score will not be assigned until after the submission deadline, and the tests are not exhaustive, so passing many (or all) of the public tests **does not** necessarily guarantee a great score (and vice versa).

For example, it is possible to pass 80% of test cases and get a score less than, equal to, or greater than 80%; all code is graded by hand. If the remaining 20% of cases would pass if not for a simple error, your score may be higher than an 80%. However, if you pass 80% of the tests but violated given constraints, your score may be lower than an 80%.

You must instead check and test your code carefully! This policy will be maintained throughout the semester, and you will be referred back to it.

B Directory Structure

In the assignment distribution, you will find the following files and directories:

- `Makefile`, containing scripts for submitting your solutions.
- `written/` contains the LaTeX sources (in `assn0.tex`) and template (in `hw00.tex`) for this assignment.
- `cmlib/` [10], a standard library for Standard ML that extends the SML/NJ built-in library. Though you are unlikely to need it for this assignment, feel free to peruse its signatures (and structures) to see its capabilities.
- `pprint/` and `unicode/` contain utility libraries.
- `var/` contains primitives for variables, as described in Section 4.1.
- `lang-ulc/` contains a (partial, to be completed on this assignment) implementation of the untyped λ-calculus, including a parser, ABT, and normalizer.
- `top/` includes top-level utilities for the assignment, such as a REPL and a tool to evaluate a file.
- `tests/` contains tests.

B.1 Reference Solution Heap Images

With each assignment, we distribute "heap images" containing compiled reference solutions of all coding tasks. You can download the relevant heap images from Canvas depending on your operating system:

- `refsol.x86-linux` is for x86 Linux machines, such as the Andrew servers.
- `refsol.x86-darwin` is an equivalent image for macOS.

Make sure the heap image you download matches your SML/NJ version, as well. If you would like images for another platform or version, please let us know.

Once you download your heap image, move it to the assignment root directory (i.e., at the same level as `Makefile`). You can use the heap image to get a better understanding of the intended behavior of the code; by running the following in the assignment root directory, you will be placed in an SML/NJ REPL session where the reference solution has been loaded.

```
smlnj @SMLload refsol
```

Via the `use` command, you may also bring in parts of your solution to test with other components of the reference solution.

[10] Available on GitHub at `https://github.com/standardml/cmlib`.