1.1. Instead of this definition, your job is to provide an alternate implementation of **mult** that only uses $\{m, n, \text{plus}, Z\}$, as well as parentheses. A few hints.

1. Fill in the template: **mult** $= \lambda\, m\, .\, \lambda\, n\, .$ λf.λx. m (plus n) Z
2. Remember that although m and n represent numbers, they are still functions that you can use however you like.
3. Use the interpreter to help you experiment!
4. A key idea here is **partial evaluation**. In this style of programming, if you give a function less than the expected number of arguments, it returns a function that waits on the remaining ones. For example, **plustwo** $=$ **plus** 2 is a function that adds 2 to a number, e.g. **plustwo** $3 = 5$.

Put your answer in the `assign2/program/mult.lam` file.

1.2. Again, see the Wiki page for more explanation and examples. In this section, your challenge is to define a function **even** that takes as input a number, and returns true if it's even, and false otherwise. You can only use the input number and the $\{$**true, false, not, if**$\}$ constructs (no recursion!). Some hints:

1. Fill in the template: **even** $= \lambda\, n\, .$ n not true
2. We can inductively define the **even** function over the natural numbers as: **even** $Z =$ **true** $\mid$ **even** $S(n) =$ **not** $($**even** $n)$

Put your answer in the `assign2/program/even.lam` file.

$$\text{fix} = \lambda\, f\, .\, (\lambda\, x\, .\, f\, (x\, x))\, (\lambda\, x\, .\, f\, (x\, x))$$

1.3. The **fix** function has the curious property of a fixpoint, which is that $\text{fix}\ f = f\ (\text{fix}\ f)$ for all f.

fix g ↦ g (fix g)

g: λf. e

So g (fix g)

→ [f→fix g] e

so g should use f as if f can refer to the recfunc itself.

g: λf.λn. iszero(n) 0 (plus n (f (pred n)))

fix g 3

→ g (fix g) 3

→ iszero(3) 0 (plus 3 (fix g 2))

→ plus 3 (fix g 2)

→ plus 3 (g (fix g) 2)

→ plus 3 (iszero(2) 0 plus 2 (fix g 1)

→ plus 3 (plus 2 (fix g 1))

→ plus 3 (plus 2 (g (fix g 1)))

→ plus 3 (plus 2 (iszero(1) 0 plus 1 (fix g 0))

→ plus 3 (plus 2 (plus 1 (fix g 0))))

→ plus 3 (plus 2 (plus 1 (iszero (0) 1 (fix g 0))

→ plus 3 (plus 2 (plus 1 0)))

= plus 3 (plus 2 1)

= plus 3 3

= 6

If you wanted to, how would you complete the proof of the equivalence of the lambda calculus and Turing machines? Once you have numbers, booleans, and a fixpoint, it's only a short jump to simulating a Turing machine inside the lambda calculus. You would need a notion of lists and tuples (and a way of getting input), and then you can define a function:

```
step : (number /*state*/, list /*tape*/) -> number /*input*/ -> (number /*state*/, list /*tape*/)
s0 = pair 0 empty
tm = fix (fun tm -> fun s -> tm (step s (get_input ())))
```

Out of respect for your time, I'll leave this as an optional exercise for the reader.

The first option, picking `x = 1`, is called *lexical* scoping. The intuition behind lexical scoping is that the program structure defines a series of "scopes" that we can infer just by reading the program, without actually running the code.

```
// The {N, ...} syntax indicates which stack of scopes are "active" a given
```

As you may have inferred at this point, dynamic scoping looks at the runtime call stack to find the most recent declaration of `x` and uses that value. Here, the binding `x = 4` in `f` is more recent than the binding `x = 1` in the top-level scope.

In both cases, dynamic or lexical, **the core algorithm is the same**: given a stack of scopes, walk up the stack to find the most recent name declaration. The only difference is whether the the scope stack is determined before or during the program's execution.

Using the above semantics, we can produce a formal proof to justify each step in the provided example trace. Below, we have provided you with the first step in the proof—your task is to fill in the proof for the remaining three steps.

When was {x→D} added?

$$\text{Step 1:}\quad \cfrac{\cfrac{\cfrac{\cfrac{}{\lambda\,\_\,.\,x\ \text{val}}\ \text{(D-Lam)}}{\{x \to D\} \vdash (\lambda\, x\, .\, \lambda\, \_\, .\, x)\ L \mapsto \lambda\, \_\, .\, x}\ \text{(D-App-Done)}}{\{x \to D\} \vdash (\lambda\, x\, .\, \lambda\, \_\, .\, x)\ L * \mapsto (\lambda\, \_\, .\, x) *}\ \text{(D-App-Lam)}}{\varnothing \vdash (\lambda\, x\, .\, (\lambda\, x\, .\, \lambda\, \_\, .\, x)\ L *)\ D \mapsto (\lambda\, x\, .\, (\lambda\, \_\, .\, x) *)\ D}\ \text{(D-App-Body)}$$

observe: λ_.x val

observe: λx.e L while e not val. zoom in to e

observe: (e L)* while (e L) not val. zoom in to (e L)

observe: λx.e D while e not val, zoom in to e

Put your answer in part 2 of the written solution.

2.1. step2:

$$\cfrac{\cfrac{\cfrac{x \to D \in \{x \to D\ -\ *\}}{\{x \to D, \_ \to *\} \vdash x \mapsto D}}{\{x \to D\} \vdash (\lambda\,\_\,.\,x)* \mapsto (\lambda\,\_\,.\,)*}}{\varnothing \vdash (\lambda x.(\lambda\,\_\,.\,x)*)D \mapsto (\lambda x.(\lambda\,\_\,.\,D)*)D}$$

observe form: x can be substitute.

observe form: λ_.x * while x not val

observe form: λx.e D while e not val

step3:

$$\cfrac{\cfrac{D\ \text{val}}{\{x \to D\} \vdash (\lambda\,\_\,.\,D)* \to D}}{\varnothing \vdash (\lambda x.(\lambda\,\_\,.\,D)*)D \mapsto (\lambda x.D)D}$$

step4

$$\cfrac{D\ \text{val}}{\varnothing \vdash (\lambda x.D)D \mapsto D.}$$

(λf.(λx.f _)D)(λ_.x)

↦ (λf.(λx.λ_.x _)D)(λ_.x)

↦ (λf.(λx.λ_.D _)D)(λ_.x)

↦ (λf.(λx.D)D)(λ_.x)

↦ (λf.D)(λ_.x)

↦ D.

2.2.

$$\cfrac{}{\lambda x.e\ \text{val}}\qquad \cfrac{x \to e \in \Gamma}{\Gamma \vdash x \mapsto e}\qquad \cfrac{\Gamma \vdash e_{lam} \mapsto e'_{lam}}{\Gamma \vdash e_{lam}\ e_{arg} \mapsto e'_{lam}\ e_{arg}}$$

$$\cfrac{e_{body}\ \text{val}}{\Gamma \vdash \text{let}\ x = e_{var}\ \text{in}\ e_{body} \vdash e_{body}}$$

$$\cfrac{\Gamma, x \to e_{var} \vdash e_{body} \mapsto e'_{body}}{\Gamma \vdash \text{let}\ x = e_{var}\ \text{in}\ e_{body} \vdash \text{let}\ x = e_{var}\ \text{in}\ e'_{body}}$$

Note: it seems that let binding is just another form of function apply

3. The first extension adds `let` statements to the language, which emulates traditional (immutable) variable binding in Turing languages.

$$\cfrac{\Gamma, x : \tau_{\text{var}} \vdash e_{\text{body}} : \tau_{\text{body}}}{\Gamma \vdash (\text{let}\ x : \tau_{\text{var}} = e_{\text{var}}\ \text{in}\ e_{\text{body}}) : \tau_{\text{body}}}\ \text{(T-Let)}\qquad \cfrac{}{\text{let}\ x : \tau = e_{\text{var}}\ \text{in}\ e_{\text{body}} \mapsto [x \to e_{\text{var}}]\ e_{\text{body}}}\ \text{(D-Let)}$$

For example, this would allow us to write:

$$\text{let}\ x : \text{num} = 3\ \text{in}\ x + 2 \mapsto 3 + 2 \mapsto 5$$

I think this is unsafe, since we can step without checking whether $e_{var} : \tau_{var}$

counterexample: let x: num = true in x+1

$$\cfrac{\Gamma, x : \text{num} \vdash x+1 : \text{num}}{\Gamma \vdash (\text{let}\ x : \text{num} = \text{true in}\ x+1) : \text{num}}$$

but if we eval it.

let x : num = true in x+1 ↦ true+1 stuck!

The second extension adds a recursor (`rec`) to the language. A recursor is like a for loop (or more accurately a "fold") that runs an expression from 0 to n:

$$\cfrac{\Gamma \vdash e_{\text{arg}} : \text{num} \quad \Gamma \vdash e_{\text{base}} : \tau \quad \Gamma, x_{\text{num}} : \text{num}, x_{\text{acc}} : \tau \vdash e_{\text{acc}} : \tau}{\Gamma \vdash \text{rec}(e_{\text{base}};\ x_{\text{num}}.\,x_{\text{acc}}.\,e_{\text{acc}})(e_{\text{arg}}) : \tau}\ \text{(T-Rec)}$$

$$\cfrac{e \mapsto e'}{\text{rec}(e_{\text{base}};\ x_{\text{num}}.\,x_{\text{acc}}.\,e_{\text{acc}})(e) \mapsto \text{rec}(e_{\text{base}};\ x_{\text{num}}.\,x_{\text{acc}}.\,e_{\text{acc}})(e')}\ \text{(D-Rec-Step)}$$

$$\cfrac{}{\text{rec}(e_{\text{base}};\ x_{\text{num}}.\,x_{\text{acc}}.\,e_{\text{acc}})(0) \mapsto e_{\text{base}}}\ \text{(D-Rec-Base)}$$

$$\cfrac{n > 0}{\text{rec}(e_{\text{base}};\ x_{\text{num}}.\,x_{\text{acc}}.\,e_{\text{acc}})(n) \mapsto [x_{\text{num}} \to n, x_{\text{acc}} \to \text{rec}(e_{\text{base}};\ x_{\text{num}}.\,x_{\text{acc}}.\,e_{\text{acc}})(n-1)]\ e_{\text{acc}}}\ \text{(D-Rec-Dec)}$$

Here's a few examples of using a recursor:

$$\text{rec}(0;\ x.\,y.\,x + y)(n) = 0 + 1 + \ldots + n$$
$$\text{rec}(1;\ x.\,y.\,x * y)(n) = 1 * 1 * 2 * \ldots * n$$

Put your answer in part 3 of the written solution.

So next, we'll try to prove that this is type-safe

Progress: if e:τ the either e val or exists e' e↦e'

if rec (e_{base}; x_{num}.x_{acc}.e_{acc})(e_{arg}) : τ

then it could not be a val, so there exists e', e↦e'

by inversion e_{arg} : num

① e_{arg} val, denoted as n

a. n=0 apply D-Rec-Base to step

b. n≥1 apply D-Rec-Dec to step,

② e_{arg} is not val, by IH, exists e'_{arg} e_{arg} ↦ e'_{arg}, apply D-Rec-Step to step

So, there always exists e' so that e↦e'

Preservation: if e:τ, e↦e', then e':τ

Since Γ ⊢ rec(e_{base}; x_{num}.x_{acc}.e_{acc})(e_{arg}) : τ by inversion,

Γ ⊢ e_{base} : τ, Γ ⊢ e_{arg} : num. Γ, x_{num} : num, x_{acc} : τ ⊢ e_{acc} : τ

① e↦e' with D-Rec-Step, by IH e_{arg} : num ↦ e'_{arg} : num so e':τ still holds

② e↦e' with D-Rec-Base, by IH e_{base} : τ ↦ e'_{base} : τ so e':τ still holds

③ e↦e' with D-Rec-Dec, Since Γ, x_{num} : num, x_{acc} : τ ⊢ e_{acc} : τ e' = [...] e_{acc} : τ

So preservation holds for the rec-extension.