1 Warmup

- Is the program x = 5 in the grammar?
  
  If so, does it typecheck?

Solution: Yes, it is in the grammar (derivable from $e_1 = e_2$); no, it does not typecheck. There is only one rule for assignment, and the rule has $x \rightarrow^{w} \tau \in \Gamma$ as a precondition. The program $x = 5$ does not satisfy this precondition because $x$ is not already in the type environment $\Gamma$. How can we get $x$ into the type environment, you ask? Looking at the typing rules, the only ones that permit adding a binding to the type environment are the four rules for functions and the rule for match. Thus, if we want to assign to some variable $x$, we can only do so inside of a function body or match statement.
• Is the program \((x : \text{Int}) \Rightarrow (x = 5)\) in the grammar?
  If so, does it typecheck?

  \textbf{Solution:} Yes, it is in the grammar; no, it does not typecheck. The reason is almost the same as before; the rule for
  assignment has \(x \rightarrow^w \tau \in \Gamma\) as a preconditon. Our program will not satisfy this preconditon because (looking at the
  rule for annotation-free functions) will have \(x \rightarrow \tau \in G\), but not \(x \rightarrow^w \tau \in \Gamma\) (that is, \(x\) is immutable).
  The story is
  the same for \((\text{name } x : \text{Int}) \Rightarrow (x = 5)\) and for the \textbf{match} statement. Thus, if we want to assign to a variable
  \(x\), we must do so inside a function with the \texttt{var} or \texttt{ref} annotation on the function parameter.

2 Call-By-Wha?!?!

• Walk through the evaluation steps for \(((\text{var } x : \text{Ptr}[\text{Int}]) \Rightarrow (x = \text{new}(3); *x))\)\((\text{new}(5))\). How many addresses
  have been allocated and assignment by the end of evaluation? Why?

  \textbf{Solution:} We use \(\circ\) to hold our memory state. Items beginning with @ are addresses.

  \begin{verbatim}
  < {}, ((var x : Int) => (x + 3))*(new(5)) >
  < {00 -> 5}, ((var x : Int) => (x + 3))*(00) >
  < {00 -> 5}, ((var x : Int) => (x + 3))(5) >
  < {00 -> 5, 01 -> 5}, [*01 / x](x + 3)> >
  < {00 -> 5, 01 -> 5}, (*01 + 3) >
  < {00 -> 5, 01 -> 5}, (5 + 3) >
  < {00 -> 5, 01 -> 5}, (8) >
  \end{verbatim}
We have allocated and assigned two memory addresses. This is because function calls with the `var` annotation allocate a new address and assign the parameter to it. It’s like putting a “local variable” for the function parameter on the heap.

- Walk through the evaluation steps for `((ref x : Int) => (x = 3))(*new(5))`. How many addresses have been allocated and assignment by the end of evaluation? Why?

Solution:

```
< {}, ((ref x : Int) => (x = 3))(*new(5))>
< {00 -> 5}, (ref x : Int) => (x = 3))(*(@0)) >
< {00 -> 5}, [*(@0)/x](x = 3)) >
< {00 -> 5}, (*(@0) = 3) >
< {00 -> 3}, () >
```

We have assigned a single memory address. This is because function calls with the `ref` annotation are pass by reference; they “reuse” the reference they are given rather than allocating a new address for the variable.

- Write a function using the `var` annotation that takes a pointer to some integer as a parameter and reassigns the pointer to the integer value 3 (in other words, rewrite the previous program using `var` instead of `ref`). Apply this function to a `Ptr[Int]` and walk through the evaluation steps.

Solution:

```
< {}, ((var x : Ptr[Int]) => (*x = 3))(new(5)) >
```
A potential problem with this language is that using the construct `if (e_1) e_2 else e_3` in a function can behave unexpectedly if we choose the wrong kind of function parameter annotation. For example, say we wrote the following program:

```
(((((tst : Boolean) =>
   ((s0 : Int) => ((s1 : Int) => if (b) s0 else s1))) (true)) (print(5); 7)) (print(6); 8))
```

What we would like is to have the program print only 5 and evaluate to 7, but currently it is printing both 5 and 6 before evaluating to 7. What is the problem? How can we fix it?

**Solution:** The problem is that we have chosen the “default” empty annotation for our function parameters. The evaluation rule for this parameter annotation insists that the parameter passed to function is evaluated before calling the function. This causes us to print both 5 and 6, since both are arguments to functions that must be evaluated before they are passed. An easy fix to this problem is changing our parameter passing to call-by-name:

```
((((tst : Boolean) => ((s0 : Int) =>
   ((s1 : Int) => if (b) s0 else s1))) (true)) (print(5); 7)) (print(6); 8))
```

The rule for call-by-name specifies that we do not evaluate a function parameter before passing it. This means that we will end up with

```
if (true) (print(5); 7) else (print(6); 8)
```

which we can evaluate to `print(5); 7` and then 7 by using the rules for `if`. This way, we never evaluate `print(6); 8`, and our program behaves as desired.