1 Even more ML

Q: How do I build a AST from ML code?

Building a AST from ML code is similar to lambda calculus. The main difference is that ML has more built-in operations, but if you loosely translate the ML code to lambda calculus then you should arrive at the right tree.

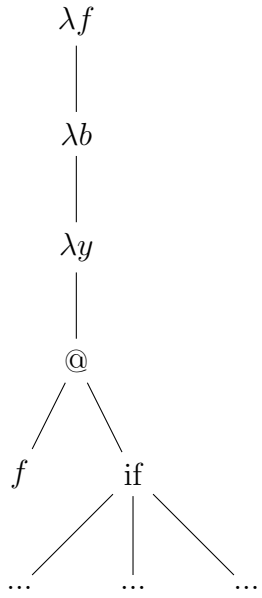
For example, take the following ML function:

```
1 func foo f b y = f (if b then y else 2)
```

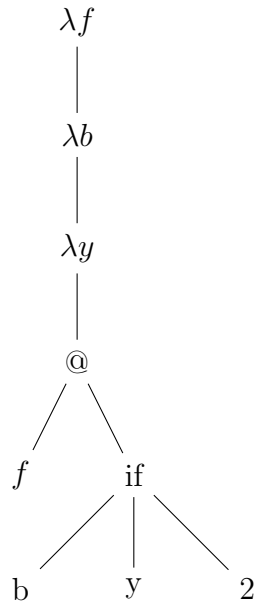
Remember that the syntax `func foo f b y` represents a curried function named foo, that takes in three arguments: f, b, and y. The top level of this program is a function (foo) so we start our tree with the representation of a function, a λ associated with the first argument:

Since the function is curried, each one-parameter function returns another function (with only one parameter).

Now we fill out the function body. The top level statement is applying `f` to an if-expression:



The three children from the if node in our tree are: the conditional expression, the then branch, and the else branch. We can then complete the tree by filling out these nodes.



A note about notation. You don't need to follow this notation exactly. As long as your tree makes sense and the representation is clear you will be okay on an exam.

2 Type Checking

Q: What is the purpose of a typechecker?

A type checker statically analyzes code for potential errors. These are known as type errors. Think of errors where you are attempting to add a boolean to and integer. Without the typechecker you would only discover this error at runtime, when the results could have real world consequences. With a typechecker that class of errors will be caught at compile time so they won't occur when the program is running.

Q: What is the role of the environment in the typechecker?

The environment is a mapping of variable names to types. Consider the expression `a + 4` in a language where the only numeric type is `int`. The type of `a` must be an int because it doesn't make sense to add a boolean to 4. However `a` may already be bound to a boolean. If the previous expression was inside a function:

```
1 func foo a : bool = a + 4
```

Now we know that `a` is a bool (because of the type annotation). When type checking the body of the function the type environment is $\Gamma, a : bool$, where Γ is the rest of the environment and $a : bool$ is a mapping of a to bool. Now the typechecker can determine the type of `a` by looking at the environment and see that the code has a type error.

Q: What is the general procedure for type inference?

1. Draw out the AST
2. Label each node with a potential type (can be any label as long as they don't collide with other labels, such as parameter names)
3. Write out all the constraints shown by the tree
4. Use the written constraints to determine as much about the types as possible.

For step 4 there is no set algorithm for determining the constraints so you will need to analyze the information you wrote down in step 3 in any way you want to come to your final program type.