Lambda Calculus
The λ-calculus is a mathematical language of lambda terms bound by a set of transformation rules. The λ-calculus notation was introduced in the 1930s by Alonzo Church.

Just like programming languages, the λ-calculus has rules for what is a valid syntax:

**Variables:** A variable (such as $x$) is valid term in the λ-calculus.

**Abstractions:** If $t$ is a term and $x$ is a variable, then the term $(\lambda x.t)$ is a lambda abstraction.

**Applications:** If $t$ and $s$ are terms, then $(ts)$ is the application term of $t$ onto $s$. 
Anonymous Functions

Similar to how (\ x \rightarrow \ t) defines an anonymous function in Haskell, lambda abstractions define anonymous functions in the \( \lambda \)-calculus.

A lambda abstraction which takes an \( x \) and returns a \( t \) is written as so:

\[ (\lambda x.t) \]

**Example**

Suppose in mathematics we define a function \( f(x) = x + 2 \). This could be written as \( (\lambda x.x + 2) \) in the \( \lambda \)-calculus\(^1\). Of course, this function is anonymous and not bound to the name \( f \).

\(^1\) Of course, we haven’t said that either \( + \) nor \( 2 \) is valid in lambda calculus yet. We will get to that...
In the $\lambda$-calculus, functions are not only first class, they are the only class of objects. In other words, all data in the $\lambda$-calculus are represented as functions.
Functions in the $\lambda$-calculus may only take one argument, so currying is typically used to write functions with multiple arguments. For example, the function $f(x, y) = x + y$ might be written anonymously as:

$$(\lambda x. (\lambda y. x + y))$$

Further, function application is left-associative, so $(fxy)$ means $((fx)y)$. 
The \( \lambda \) operator (which creates lambda abstractions) binds a variable to wherever it occurs in the expression.

- Variables which are bound in an expression are called **bound variables**
- Variables which are not bound in an expression are called **free variables**

**Example**

With your learning group, identify the free and bound variables in this expression:

\[
(\lambda x.(\lambda y.zy)(zx))
\]
**Transformations**

- **α-conversion**: Allows variables to be renamed to non-colliding names. For example, $(\lambda x.x)$ is $\alpha$-equivalent to $(\lambda y.y)$.

- **β-reduction**: Allows functions to be applied. For example, $((\lambda x.x^2)8)$ is $\beta$-equivalent to $64$.

- **η-conversion**: Allows functions with the same external properties to be substituted. For example, $(\lambda x.(fx))$ is $\eta$-equivalent to $f$ if $x$ does not appear in $f$. 
Examples

With your learning group, identify the transformation used in each of the following expressions, or state they are not equivalent. Turn in your answers on a sheet of paper with all of your names at the end of class for learning group participation credit for today.

1. $(\lambda x.(\lambda x.x)) \rightarrow (\lambda y.(\lambda y.y))$
2. $(\lambda x.(\lambda x.x)) \rightarrow (\lambda y.(\lambda x.x))$
3. $(\lambda x.(\lambda x.x)) \rightarrow (\lambda y.(\lambda x.y))$
4. $(\lambda x.(\lambda y.x)) \rightarrow (\lambda y.(\lambda y.y))$
5. $((\lambda x.x)(\lambda y.y)) \rightarrow (\lambda y.y)$
6. $(\lambda x.((\lambda y.y)x)) \rightarrow (\lambda y.y)$
Since all data in the $\lambda$-calculus must be a function, we use a clever convention of functions (called **Church numerals**) to define numbers:

0: $$(\lambda f.(\lambda x.x))$$
1: $$(\lambda f.(\lambda x.(fx)))$$
2: $$(\lambda f.(\lambda x.(ffx)))$$
3: $$(\lambda f.(\lambda x.(fffx)))$$

... and so on. In fact, the successor to any number $n$ can be written as:

$$\lambda f.\lambda x.f(nfx)$$

**Notice this**

Defining numbers as functions in this way allows us to apply a Church numeral $n$ to a function to get a new function that applies the original function $n$ times.
Shorthand Notations

While it’s not a defined part of the $\lambda$-calculus, we define common shorthands for some features:

- 0, 1, 2, ... are shorthand for their corresponding Church numerals
- $\text{SUCC} = \lambda n. \lambda f. \lambda x. f(nfx)$

**Note**

The notation "=" above is not a part of the $\lambda$-calculus. I’m using it for saying "is shorthand for".
Adding $m$ to $n$ can be thought of as taking the successor to $n$, $m$ times. Using our shorthand SUCC, this can be written as:

$$ADD = \lambda m.\lambda n.(m \text{ SUCC } n)$$

Similarly, multiplying $m$ by $n$ can be thought of as repeating ADD $n$, $m$ times and then applying it to 0, this can be written as:

$$MULT = \lambda m.\lambda n.(m(\text{ADD } n)0)$$
Boolean Logic

We use the following convention for true and false:

$$\text{TRUE} = \lambda x.\lambda y.x$$
$$\text{FALSE} = \lambda x.\lambda y.y \quad \text{(Church numeral zero)}$$

From here, we can define some common boolean operators:

$$\text{AND} = \lambda p.\lambda q.pqp$$
$$\text{OR} = \lambda p.\lambda q.ppq$$
$$\text{NOT} = \lambda p.p \text{ FALSE TRUE}$$
$$\text{IF} = \lambda p.\lambda a.\lambda b.pab$$
   \( \text{(returns } a \text{ if the predicate is TRUE, } b \text{ otherwise)} \)
By convention, we will represent a cons cell as a function that applies its argument to the CAR and CDR of the cons cell. This leads to the shorthand:

\[
\text{CONS} = \lambda x.\lambda y.\lambda f.fyx \\
\text{CAR} = \lambda c.c\,\text{TRUE} \\
\text{CDR} = \lambda c.c\,\text{FALSE} \\
\text{NIL} = \lambda x.\text{TRUE}
\]

Using this, we can define lists:

\[
(\text{CONS} \, 1 \, (\text{CONS} \, 2 \, (\text{CONS} \, 3 \, \text{NIL})))
\]
Subtraction is hard, but doable. Check out the Wikipedia page on Church Numerals for more info.

For recursion, we need to reference ourselves in a lambda abstraction. This is done using a Y-combinator.

From there, we can use the $\lambda$-calculus to compute the solution to any problem that a Turing machine can.

More on all of this in CSCI-561 (Theory of Computation).

Many functional programming languages (e.g., Haskell, Lisp) are just practical implementations of the $\lambda$-calculus.
Monads (not a quiz or exam topic)
Monads are a class of functions that compose other functions together in a certain way. A type with a monadic structure defines what it means to chain operations. A monadic type consists of a type constructor and two operations:

- **Return**: Takes a plain value, and uses the constructor to place the value in a monadic container, creating a monadic value.
- **Bind**: Does the reverse: takes a monadic container and passes it to the next function.

Remember that silly function in Haskell (>>=) that chained IO statements together?
In Haskell, when you write a list comprehension:

\[ \{ x \times 2 \mid x \leftarrow [1\ldots10], \text{odd } x \} \]

In Haskell, the `do` block used for IO:

```haskell
main = do
    putStrLn "What is your name?"
    name <- getLine
    putStrLn $ "Nice to meet you " ++ name
```
Monads essentially are a hidden data structure that passes around state for you.

Many common imperative PL concepts can be defined in terms of a monadic structure, such as random number generators, input/output, variable assignment, ...

Monads can be created in any language that supports anonymous functions and closures.
From https://xkcd.com/1957/:

CVE-2018-????: Haskell isn’t side-effect free after all, the effects are all just concentrated on this one computer in Missouri that nobody has checked in on in a while.