- Lambda Calculus
  - A model of computation for functional languages, that relies on function composition
  - Everything in lambda calculus is a function and returns a function, or a lambda expression that declares a new variable (function)
  - Example 1
    - λa means we're declaring a function that takes one parameter, a
    - After the dot, we write the function that λa is defining, as far right as possible. In this case, to the closing parentheses, so a
    - If the parentheses didn't exist, though, the body would include b, so ab ○ Bound variables mean we can find the declaration of the variable within our expression. Unbound, or free, means we cannot, and it was declared outside of the expression
  - Example 1
    - Here, a is bound, because we see it declared in λa
    - b is free, though, since looking towards its left we cannot see its declaration
    - Same goes for if there were no parentheses, b is free, while a is bound
  - Let's look at making parentheses explicit.
  - Example 2
    - Lambda functions are left associative, meaning we do things left to right
    - That means for a b c, we first do (a b), then we apply c to its result ■ Answer: ((a b) c)
  - Example 3
    - Recall the expression extends as far right as possible, so λa extends all the way to the right, as does λb
    - We add parentheses around the entire lambda expression, as well as the body of the lambda expression
    - Answer: (λa.(λb.(a b)))
    - Note that this example shows a function that takes two variables, a and b ○ Example 4
    - The first λa extends all the way to the right
    - We see an associativity problem between a b and λa.a b, so we group a b together, and it takes in the next lambda expression
    - Answer: (λa.((a b) (λa.(a b))))
  - Now we'll look at identifying free and bound variables
  - Example 1
    - It might help to make the parentheses explicit for these examples
    - (λa.((a b) a))
    - Since the lambda expression extends all the way right, both a's are bound
    - b is free, though, because there is no lambda expression to the left that binds it
  - Example 2

- This is tricky because there are only a's and there is a lambda expression that takes a
- However, we have to consider the body of the lambda expression, which in this case, only contains 1 a bounded. The other two a's are not within the body of the lambda expression (not in the scope). One a is too far left, the other is on the right but outside the parentheses
- Thus, the first and last a's are free

○ Example 3
- Both a's are bound because the first λa extends all the way to the right, and includes any a's within contained lambda expressions
- The first b is also bound by the λb
- The second b is free though, because it is outside of the scope of λb ○

Alpha-conversions are used to avoid ambiguity by renaming variables that have the same name. If a variable is declared within our expression using lambda, we can just change the letter and change the uses

○ There can be multiple correct alpha-conversions

○ Example 1
- We need to identify ownership first
- The inner lambda expression owns a, so the outer lambda expression does not own any variables - it gets overshadowed
- We want to distinguish the first a from the second a parameter
- Two possible answers
  - λb.λa.a
  - λa.λb.b
- As long as the bindings stay the same, the conversion is correct

○ Example 2
- The inner a is different from the outer a - the outer a is free
- Since the outer a is free, we cannot rename it, because we would not be able to convert its declaration
- Same goes for b
- The only solution is thus (λc.c) a b, where c is any letter besides a or b ○

Example 3
- The last a is free, and the other 3 a's are bound to different lambda expressions
- (λd.(λc.(λb.b) c) d) a

○ Beta-Reductions apply the functions to our arguments, to evaluate them as much as possible. Again, it may help to make parentheses explicit

○ Example 1
- The explicit parentheses are (((λa. (a b)) x) b)
- First, we take x and we plug it into λa (so now, a = x)
- This evaluates to (x b) b, which is our answer
- We can only plug in when we have a lambda expression

○ Example 2

    - In this example, we will be passing an entire lambda expression into the first expression
    - We also see an example of currying: (λa. λb. λc. a b c) is equivalent to (λa. (λb. (λc. a b c))), so each lambda expression would return a new lambda expression with a variable input already
    - This is the benefit of lambda calculus, it captures higher order programming - passing in functions as parameters - and currying ■ We pass in the second lambda expression to the first one, but there are no a's being used in the first lambda expression, so we just output b as our answer
    - Answer: b
  - Example 3
    - We plug in the second expression into a in the first one, but notice this gives us the same expression as the original again
    - This means we are already done reducing, and we can't reduce further - it loops forever
    - Sometimes you may get an expression that grows as it "reduces"; this is also referred to as "cannot be reduced further"
    - Note that call-by-name and call-by-value will be important here sometimes, depending on which you do, the expression may or may not reduce indefinitely
  - Beta Normal Form is the state of a lambda expression where it cannot be beta reduced further. It just involves a chain of beta-reductions until it has reached a state of no more changes
  - Note that all of these have been either put into beta normal form, or cannot be reduced further
  - On the exam, you will likely get questions that involve applying beta reduction multiple times to get to beta normal form, instead of just one time.