CS332 Module 06: Lambda Calculus

Outcomes: By the end of these lectures you should be able to perform the following.

1. Identify the values, identifiers, operatos, and operands in an expression.
2. Identify the free and bound variables in an expression.
3. Identify the scope of any bindings in an expression.
4. Identify where accidental capture may occur in an expression.
5. Reduce an expression using alpha, α, and beta, β, reduction rules.
6. Recognize two expressions as structurally equivalent or not.

References:

1. A Tutorial Introduction to the Lambda Calculus, Raul Rojas

Notes:

1. Motivation:

   (a) The most immediate motivation for this topic in this course is to provide you with an indirect introduction to the *functional* programming paradigm. The next section of the course will introduce you to a functional programming language, and require that you create some simple programs in the language. All functional programming languages are built directly on the λ calculus. Without a basic understanding of the λ calculus, these programs are incomprehensible. This benefit goes beyond the functional language we will use in this course. Most modern programming languages support the functional paradigm to some degree. Python supports it fully, and Java has introduced some elements. Being a top notch software professional in your profession will likely require some knowledge of the functional paradigm.

   (b) A more abstract motivation for this topic is to provide you with a clearer idea of what computation is. As we discussed in the language topic, people nearly often mistake the usage of a thing with the underlying thing. Natural languages are *used* to communicate, and programming languages are *used* to instruct computers, but that's not what languages *are.* Languages, as we have seen, are simply sets of strings.

   Similarly, people typically confuse the *usage* of computation with the *nature* of computation. Computation may be used to find a numerical or Boolean result, but that's not what it is. Computation is mothing more than the *manipulation of symbols using well defined rules.* You'll see more why that is so as we go through the material. This realization has several benefits. First, it reminds us that computation is a *mindless process* - it has to be for computers to do it. Second, it expands the notion of computation beyond the mathematical notion that people typically have. For example, using the "Find and Replace" feature in a word processor is computation; it's a manipulation of symbols using well defined rules. This definition of computation allows us to expand the rigor

and formalsims we normally only associate with mathematics to any form of software. Last, sometimes when designing software, reminding yourself that you are attempting to construct a mindless, meaningless manipulation of symbols can be a clarifying thought. It sweeps away the clutter of the *use* of the software and reminds you to focus on the *actions* of the software.

(c) The last (and admittedly most difficult to sell) motivation is to remind you of one of the meta-lessons of the course regarding the nature of formal systems. All formal systems have a small number of things, a small number of operators to manipulate those things, and achieve complexity through iteration. Again, this can help with design. If you can identify the things in the system you are designing and the required basic operators it can clarify the design. Complexity will come later. Note: This is possibly most important in Object-Oriented design, as designers literally must decide what classes are needed, what they're attributes (things) are, and what basic actions (operators) they need to perfom.

2. Introduction: The λ calculus was developed by Alonzo Church and others in the 1930s. Church's goal was to establish a minimal system that would encompass all (mathematical) computation. The system he devised is simple, yet fully expressive of all computations. In that respect it is equivalent to the Turing Machine. Both systems were used to explore the limits of computation, and both systems came to the same result, hence the name Church-Turing Thesis. The Church-Turing Thesis, in it's most basic form, says that some things just aren't computable. More importantly, the things that are not computable in the Turing Machine system are not computable in the λ calculus, and vice versa. The two systems are equivalent, and arrived at the same conclusions independently.

   The λ calculus is composed of very few, highly abstract, elements. There are no primitive values, such as the number 0 or 1, though those values can be built within the system (Note: that's how abstract the λ calculus is!). The only components of the λ calculus are variable names (x, y, n, and so on), and expressions. Expressions will be represented as exp in these notes. There are very few rules as to how to make expressions. For example, an expression might be simply a sequence of variables separated by spaces. An expression might be, and often are, a sequence of expressions separated by spaces. Details will be provided in the discussion section.

   There are only two special, or non-trivial, forms of expressions, the *function* (sometimes called abstraction), and *application*. The details of those forms are discussed below. The function is a standalone expression, while application consists of a function followed by another expression – exp_1 exp_2. Typically, exp_1 is a function, while exp_2 serves as input to the function. It's called application because the function, exp_1, is being applied to the input, exp_2. The rules for how to perform the application are provided in the discussion below.

   The bottom line is that the λ calculus is a formal system that describes computation by applying functions to other expressions. Because it is highly abstract, it can be used to describe any computation applied to any domain. Because it is formal, it contains only a few things, and a few rules to manipulate those things. A word of warning before we dive into the details — the λ calculus is a notational mess! The expression we will look at are ugly, so it give yourself a little time to get comfortable with that.

3. Definitions and Notation:

(a) Variable: A *variable*, *var*, is simply a single character or string of characters.

(b) Function: A *function*, is a specific form of λ expression, having the form $\lambda\ x.\ exp$, where λ simply signifies the beginning ot the function, x is a variable, and exp is an expression. Note: Functions may also be referred to as *abstractions* or λ *functions*, though this course will use the term *function*.

(c) Bound Variable: A variable is considered *bound* if it occurs in a function to the left of the "dot." Given, $\lambda x.\ exp$, then x is a bound variable in the expression, exp.

(d) Free Variable: A variable is considered *free* if it occurs in an expression of a function, but is not listed to the left of the "dot." Given $\lambda\ x.\ exp$, any ocurence of y in exp (for example) is free. A free variable is not bound.

(e) Application: *Application* is defined as a sequence of two expressions, $exp_1\ exp_2$. Typically, exp_1 is a function being applied to exp_2.

(f) Substitution: Substitution, denoted $[x/value]$, is the act of replacing every occurence of the variable x with it's value, *value*. The value is yet another expression.

(g) Expression: An *expression* is recursively defined in BNF as follows:

$$exp \rightarrow var \mid \lambda var.\ exp \mid exp_1\ exp_2$$

In other words, a λ expression is either a single variable name, a function, or an application. (Note: Yes, this is enought to describe every possible computation, a fact that never ceases to amaze me.)

(h) α Reduction: An α reduction is the replacement of a bound variable name for another variable name. Examples provided in the discussion section.

(i) β Reduction: A β reduction is the substitution of every occurence of a bound variable within an expression with its value, and removal of that variable from the function as a bound variable. Examples provided in the discussion section.

4. Discussion

(a) Remember that the pure λ calculus does not contain numbers or mathematical operators, like plus or minus, as primitive elements. We're going to first *pretend* that it does for convenience. So the examples use numbers and math operators just as you're used to. Then we're going to *prove* that these things can be derived within the λ calculus, and are legitimately contained within the system. Lastly, the examples does use parenthesis to group terms for clarity.

(b) Writing λ expressions is not difficult. Here are some examples:

x (a variable)
$\lambda x.(x + 1)$ (a function)
$x\ y$ (an expression followed by an expression - application)
$\lambda x.(x+1)\ 4$ (application, where the function is exp_1 and 4 is exp_2)

(c) The previous function example used x as a variable. Did it have to? Is $\lambda x.(x + 1)$ the same as $\lambda y.(y + 1)$, or the same as $\lambda q.(q + 1)$ or $\lambda area.(area + 1)$? The answer is yes – symbols have no inherent meaning. It is the structure of the expression that is important, not the specific variable names used. This will become important later as we

start to manipulate expressions. Changing the variable names is allowed, but changing their structure is not allowed!

(d) Variables that occur within the expression portion of a function are either *bound* or *free*. Given

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

the variable x is bound, because it is listed next to the λ symbol, while y is free. This is straightforward for this simple example, but things start to get more complicated. Remember that the basic form of the function is $\lambda x.exp$, and the *expr* can be any other expression - even another function. Here's a more complicated example:

$$\lambda x.(x + \lambda y.(y + 12))$$

Here, both x and y are bound, but not in the same function. The y is bound at the inner function level only, and x is bound at the outer level. (Note: I warned you that the notation was going be ugly – we're just getting started!) Let's look at a λ expression that one should never, ever write:

$$\lambda x.(x + y + \lambda y.(y + 1))$$

Here, x is bound at the outer layer as before, but y occurs both bound, at the inner layer, and free at the outer layer. This is the reason for the discussion above about the importance of structure over specific variable names. The expression should properly be written as follows because the two y's really have nothing to do with each other:

$$\lambda x.(x + t + \lambda y.(y + 1))$$

Now the bindings are clear. Variable y is bound at the inner layer, x at the outer layer, and t is free. Here's a more common variation. You'll notice that the function only accepts one variable, $\lambda x.expr$. Two create functions of more than one variable, we need to nest functions, as in the previous example. This typically looks like:

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

Here, x, y, and p are all bound. This pattern shows up quite a bit, so a shorthand method to write it is:

$$\lambda xyp.(y + x - p)$$

(e) There are only three operations in the λ calculus: the α reduction, β reduction, and η reduction. This section discusses the β reduction, which is the rule used to apply a function to some input. The β reduction simply states that given the application $\lambda x.f(x)$ exp, you replace every occurence of x in $f(x)$ with exp, and remove the $\lambda x.$ portion of the expression. For example:

$\lambda x.x$ 42 reduces to 42, and

$\lambda x.(x+5)$ 16 reduces to 16 + 5 (which further reduces to 21, since we're pretending normal math rules work for now), and

$\lambda x.(x + y)$ 17 reduces to (17 + y), and

$\lambda x.x$ $\lambda z.(z + 2)$ reduces to $\lambda z.(z + 2)$.

This last is rather important. Notice that instead of having a numerical value as input to the function, we used another function as input to the function. This is because a function can be applied to any expression, even another function. Put another way, a function can accept anything as input, even another function. This is what gives such great descriptive power to the λ calculus, and to the functional programming paradigm that is based on the λ calculus.

(f) The α reduction simply allows one to rename the variables in any expression, as long as the form of the expression is not changed. This is allowed because, as we said earlier, the variables have no meaning. The following example provides a rationale as to why the α reduction is needed.

Are the two expressions $(x + y)$ and $(t + y)$ the same? Yes, because the variable names have no intrinsic meaning, and both expressions are adding two varaibles that are not related to each other. But what about $(y+y)$? This is not the same, because it is adding a value to itself, not two separate variables. The structure is different. Placing those expressions in functions points out the difference more clearly:

$\lambda xy.(x + y)$ will clearly add two numbers (using normal math rules), while
$\lambda xy.(y + y)$ clearly adds a number to itself.

These things are not the same. There are cases where careless use of a β reduction causes an accidental change in meaning. Given:

$\lambda xy.(x + y)\ t$, we get $\lambda y.(t + y)$.

In this expression, y is bound and t is free, and the expression adds two unrelated variables. But, ...

$\lambda xy.(x + y)\ y$ results in $\lambda y.(y + y)$

and the resulting expression has only one bound variable, y, and no free variables. The form has changed. This is called *accidental capture*, and the α reduction is used to prevent that. So, given

$\lambda xy.(x + y)\ y$, the first thing to do is use an α reduction to rename the second y to something else, and the perform the β reduction. It looks like:

$\lambda xy.(x + y)\ y$ becomes
$\lambda xy.(x + y)\ p$ using an α reduction to rename the free y to p,
$\lambda y.(p + y)$ using the β reduction.

It didn't matter what variable got renamed, as long as it avoided the accidental capture. The previous example could have been done as follows:

$\lambda xy.(x + y)\ y$ becomes
$\lambda xq.(x + q)\ y$ using an α reduction to rename the bound y to q,
$\lambda q.(y + q)$ using the β reduction.

The result is the same, because the form is the same.

(g) As a convenience, we will allow expressions to be named. This helps to reduce the notational clutter inherent in the system. For example, $\lambda x.x$ is clearly the idenity function – it takes in a value and produces the same value unchanged. It is convenient to give it a name, such as **I**. This is written as $\mathbf{I} \equiv \lambda x.x$. This will be important for the next discussion topic.

(h) Up till now we've pretended that the λ calculus contains numbers and basic math operations. This section shows how these things are derived from the basic operations. We start with an expression that we will call zero (0), and then create a *sucessor* function that adds one (1) to it. So, 0 is defined as a baseline, and 1 is defined as 0 as the successor to 0, and 2 is defined as the successor to 1, and so on. It looks like this:

$\mathbf{0} \equiv \lambda sz.z$ (the z is an intentional mnemonic for zero, and s for successor)
$\mathbf{1} \equiv \lambda sz.s(z)$
$\mathbf{2} \equiv \lambda sz.s(s(z))$
$\mathbf{3} \equiv \lambda sz.s(s(s(z)))$, and so on ....

This is an horrific way to write integers! But it works, as long as there is a successor function to create these patterns. And of course there is. The successor function, **S**, is defined as:

$$\mathbf{S} \equiv \lambda wyx.y(w\ y\ x)$$

Applying the successor function to zero, we get:

$\mathbf{S}\ \mathbf{0} \equiv \lambda wyx.y(w\ y\ x)\ \lambda sz.z$, which is then
$\mathbf{S}\ \mathbf{0} \equiv \lambda yx.y(\lambda sz.z\ y\ x)$, using β reduction, and again,
$\mathbf{S}\ \mathbf{0} \equiv \lambda yx.y(\lambda z.z\ x)$, using β reduction inside the parenthesis, and
$\mathbf{S}\ \mathbf{0} \equiv \lambda yx.y(x) \equiv \mathbf{1}$ (same form as $\lambda sz.s(z)$).

The transition from the second to the third line might look suspicious until you remember how the β reduction works. We applied $\lambda sz.z$ to y, and so had to replace all occurrences of s with y – but there were no occurrences, so the y just disappeared.

Applying **S** to **1**, we will get **2**:

$\mathbf{S}\ \mathbf{1} \equiv \lambda wyx.y(w\ y\ x)\ \lambda sz.s(z)$
$\mathbf{S}\ \mathbf{1} \equiv \lambda yx.y(\lambda sz.s(z)\ y\ x)$
$\mathbf{S}\ \mathbf{1} \equiv \lambda yx.y(\lambda z.y(z)\ x)$
$\mathbf{S}\ \mathbf{1} \equiv \lambda yx.y(y(x)) \equiv \mathbf{2}$.