Chapter 1

Lambda Calculus

1.1 Intro

According to Wikipedia¹, “Lambda calculus (also written as λ-calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.”

While correct, we will be treating it as a Turing complete language which is the basis of many functional languages. So we will go over how to create statements in this language, and how to evaluate them.

Let us begin by giving the grammar for the language:

\[ e \Rightarrow x \]
\[ \lambda x . e \]
\[ ee \]

\( x \) here is a variable. That’s it. A very small language grammar. Using this grammar the following statements are valid:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>y</td>
<td>( \lambda x . x )</td>
</tr>
<tr>
<td>xy</td>
<td>xx</td>
<td>( \lambda x . y . x )</td>
<td>( \lambda x . y . y x y )</td>
</tr>
</tbody>
</table>

1.2 Turing Complete

As we saw earlier, finite state machine can only solve certain types of problems. Different machines have different levels of computational power. Checking if a string is accepted by a regular expression requires a Finite State Machine. Checking if a string is accepted by a Context Free Languages requires a Push Down Automata (PDA)². Recursively enumerable languages need a Turing machine. A Turing Machine³, can solve any computable problem. Some problems, we know cannot be solved: like the halting problem, thus a Turing machine cannot solve this problem. But if we know the problem can be solved, then a Turing machine, can model it.

It is important to note the difference of syntax and semantics here. CFGs and regular expressions only describe the syntax of the language. Since they only describe sets of strings, they say nothing about what those strings can express (in this case, a language is just a set of strings that is allowed). What a language is capable of expressing (or its semantics) is a different question (in this case, a language is a way to express ideas). We will now pull away from the former way of speaking about languages, and start discussing the latter. One property the semantics of a language could have is Turing

---

1¹https://en.wikipedia.org/wiki/Lambda_calculus
2²We will not be covering PDAs in this course
3³Initially called a-machines or atomic machines
Completeness.

When we say a language is Turing complete, we mean that language can model all problems a universal Turing machine can. A Universal Turing Machines (UTM), is a Turing machines that can produce other Turing machines, where each machine solves a particular problem (a machine that AND can't be used to OR). What does a Turing complete language look like? Well, despite the fact that the syntax of Lambda Calculus can be represented as a CFG, the semantics of the language is enough to be Turing complete.

1.3 Lambda Calculus Semantics

As we saw in the intro section, the syntax of the language is very small. It's now time to figure out what do those lambda calculus sentences mean.

1.3.1 Variables

Let's start with the basic ‘x’ sentence. In languages like C, a variable by itself means whatever value is bound to that variable.

```plaintext
int x = 3;
x; //here x means 3
```

In lambda calculus, this meaning stays the same, however what the variable is bound too is sometimes not given.

```plaintext
y //y not defined, but understood to be a variable
/*
Analogous to the following C code
void foo(){
y; //not defined in this scope, but understood to be defined elsewhere.
}
*/
```

1.3.2 Function Definitions

The next type of sentence we is referred to as a lambda function. A lambda function takes the form $\lambda x . e$, where $x$ is a variable and $e$ is another lambda expression. As seen in the intro section, $\lambda y . x$ is a valid lambda expression, where $y$ is a variable and $x$ is another lambda expressions (in this case, a variable). This is called a function because it has the same semantics of a function we may see in other languages. Let's first break down it's structure though.

$\lambda y . x$

This is a lambda function where the $y$ can be thought of as the parameter to the function, and $x$ can be thought of as the body.

Since lambda functions are also lambda expressions, they can be placed as the body of another function: $\lambda x . \lambda y . y$. This one is more complicated, so let's look at it's structure:

$\lambda x . \lambda y . y$

In this case, the $\lambda x$ function has parameter ‘$x$’ and body of $\lambda y . y$. The $\lambda y$ function has parameter ‘$y$’ and body ‘$y$’.

Here is something analogous in other languages:
Each thing here is a function, that takes in one parameter, and then returns a function (where that returned function is the identity function).

As you can imagine, we can nest these even more: \( \lambda x . \lambda y . \lambda z . z \). This becomes important when we change the body of the nested function: \( \lambda x . \lambda y . x \) and start to do function application. More on this in a few sections.

### 1.3.3 Function Application

As the name suggests the last part is a way to call a function. However, this is also a bit of a misnomer, since sometimes we cannot apply a function. Let’s expand on this.

\((\lambda x . x)a\)

This is a lambda expression that consists of 2 sub-expressions: \((\lambda x . x)\) and \(a\). When we have this structure, we call the left sub-expression using the right sub-expression as its input. Specifically, \(a\) will be used in the \((\lambda x . x)\) function. Since this function is the identity function, we just get back \(a\).

\((\lambda x . x)a \Rightarrow a\)

This is equivalent to calling a function and getting the return value. Here are some analogous examples in some programming languages:

<table>
<thead>
<tr>
<th>For reference: lambda calculus: ((\lambda x . x))</th>
<th># Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda x . x))</td>
<td>(lambda x: x)(a)</td>
</tr>
<tr>
<td>((\lambda x . y))</td>
<td># Ruby</td>
</tr>
<tr>
<td>((\lambda x . y))</td>
<td>Proc.new{</td>
</tr>
<tr>
<td>(\lambda x . \lambda y . x)</td>
<td>// Java</td>
</tr>
<tr>
<td>(\lambda x . \lambda y . x)</td>
<td>interface Lambda{ Object run(Object i); }</td>
</tr>
<tr>
<td>(\lambda x . \lambda y . x)</td>
<td>public static Lambda foo(Object x){</td>
</tr>
<tr>
<td>(\lambda x . \lambda y . x)</td>
<td>return (Lambda)((y) -&gt; y);</td>
</tr>
<tr>
<td>(\lambda x . \lambda y . x)</td>
<td>}</td>
</tr>
</tbody>
</table>

Notice, that we have an argument, and it is being replaced in the body with whatever the input is. Thus when given something like \((\lambda x . y)a\), we get \(y\) back. To see this in more detail, consider the following:

```java
int foo(int i){
    return i;
}

foo(3) //returns 3, because 3 is our input, where ever we see x, replace with 3.
```

1. int bar(int i){
2.     return 4;
3. }
4. bar(3) //returns 4, because 3 is our input, but we don’t use x anywhere

To see more complicated examples, we first need to consider the following lambda expression:

\( a b \)
This lambda expression has two sub-expressions: \(a\) and \(b\). Here however, \(a\) is not a lambda function, so we have no idea how to call \(a\) with \(b\) as input. In this case, since we cannot call a function, we say this expression means exactly what it says: \(a b\).

To see this side by side:

\[
(\lambda x. x) a \Rightarrow a \quad (\lambda x. y) a \Rightarrow y \quad a b \Rightarrow a b
\]

### 1.4 Removing Ambiguity

Now that we have the basics down, let’s consider something more complicated:

\[
\lambda x. x \ y
\]

We can see this is an ambiguous statement. Either \(x y\) the body of the \(\lambda x\) function, or \(y\) is being applied to the \(\lambda x. x\) function.

Here is another example of an ambiguous expression:

\[
a \ b \ c
\]

\[
(a b c)
\]

It is initially unclear if we are calling \(a\) with the input \(b c\) or if we are calling \(a\) with input \(b\) and then calling \(c\) on the result of \(a b\).

To help remove ambiguity, there are some explicit rules that Lambda Calculus follows.

- Expressions are left associative
- The scope of a function goes until the end of the entire expression or until a (unmatched) parenthesis is reached

#### 1.4.1 Left Associative

When we say expressions are left associative, that means when given a series of expressions, for example three: \(e_1 e_2 e_3\), then we group the left two items together before grouping the next item: \((e_1 e_2) e_3\). This means that we take the right tree in the above example with \(a b c\). As you can imagine, we chain this rule with larger expressions. Thus: \(a b c d e f\) has implicit parenthesis: \(((a b) c) d) e f\).

For functions, this is important since it tells us what order we should apply things:

\[
(\lambda x. \lambda y. y) a b \\
\Rightarrow ((\lambda x. \lambda y. y) a) b \\
\Rightarrow (\lambda y. y) b \\
\Rightarrow b
\]

#### 1.4.2 Function Scope

The next important part is the scope of a lambda function. The body of a lambda function is whatever follows the \(\cdot\) symbol until the end of the entire expression is reached or a (unmatched) parenthesis is reached. It is important to note the
1.5. REDUCTION

The process of applying a function is called reducing. In particular, we call a function call a beta reduction (\(\beta\)-reduction). A single function call is a reduction. So the following is actually performing two reductions.

\[
(\lambda x.\lambda y.) a b \\
\Rightarrow (\lambda x.\lambda y.) a b \\
\Rightarrow (\lambda y.) b \quad \text{reduced the } x \text{ function} \\
\Rightarrow b \quad \text{reduced the } y \text{ function}
\]

When you cannot reduce any further, we say the expression is in beta normal form. So \(b\) is the result of reducing \((\lambda x.\lambda y.) a b\) to beta normal form.

When performing a beta reduction, consider there is the function being called, and the expression passed in to the function. For example: \((\lambda x.\lambda y.) x\ y\) can be beta reduced by calling the \(\lambda x.\ x\) function with the input \(y\).

What happens when the input can also be reduced? \((\lambda x.\lambda y.) (\lambda y.\ y\ y)\ z\) We have two options:

- We evaluate the argument first. That is, evaluate \((\lambda y.\ y\ y)\ z\) to \((z\ z)\) and pass that into \((\lambda x.\ a\ a)\)
- We shove \((\lambda y.\ y\ y)\ z\) into \((\lambda x.\ a\ a)\) first and get \((\lambda y.\ y\ y)\ a\)

The first is an example of eager evaluation (sometimes called call-by-value), where we evaluate the argument first before passing it into the function.

The second is an example of lazy evaluation (sometimes called call-by-name), where we only evaluate when we need to.

For example:

```python
1 def foo(x):
2     return "str"
3
4 foo(3+4)
5 # using lazy evaluation, 3 + 4 is never calculated
6 # using eager, 3 + 4 is calculated and then never used.
7 # in this case lazy does less work
8
9 # Try running print(foo((lambda x: print(x))(4))).
10 # Does python use eager or lazy evaluation?
```

1.6 Variable Semantics

Up until this point, we have been using simple functions, but typically more complicated or harder to read functions are used. For example:

\[
(\lambda x.\ (\lambda x.\ (\lambda y.\ x\ y))\ x)\ x
\]

To figure out how to reduce this problem, we need to discuss variables and their binding.
Variables fall under two categories: free or bound. A bound variable is one who’s value is dependent on a parameter of a lambda function.

\[(\lambda x. x \, a) \, b\]

In the above example, \(x\) is bound to the input parameter since they share the same name and \(x\) is in the body of the \(\lambda x\) function. \(a\) and \(b\) are then what are known as free variables, variables not dependent on the parameter. \(b\) is not bound because it falls outside the body of the function, and does not share the same name as the parameter. \(a\) is not bound because it does not share the same name as the parameter. Consider the following C program:

```
int a = 6; // free
int foo(int x){ // x is name of the parameter
  x == 5; // this x is bound to the parameter
  int y = 3; // free
}
int a = 6; // free
{
  int a = 5;
  printf("%d
",a); //prints 5 here since a is shadowed by the previous line
}
printf("%d
",a); //prints 6, now that the inner 5 is out of scope
```

In regards to the `foo` function, \(a\) and \(y\) are free since they are not bound to the parameter \(x\).

This becomes important when we nest functions:

\[(\lambda x. \lambda y. x \, a) \, b\]

Here, \(x\) is bound to the outer \(\lambda x\) parameter and the \(y\) is bound to the inner \(\lambda y\) parameter. This gets a tad confusing when we shadow the variable:

\[(\lambda x. \lambda x. x) \, a\]

In these cases, we know that \(x\) is bound, but to what? In lambda calculus (and most languages that I know), the \(x\) is bound to the inner \(\lambda\) function. This is because the parameter is being shadowed by the inner function. Consider the following C code:

```
int a = 6; // free
{
  int a = 5;
  printf("%d
",a); //prints 5 here since a is shadowed by the previous line
}
printf("%d
",a); //prints 6, now that the inner 5 is out of scope
```

Thus, we can now beta reduce complicated functions if we keep this is mind:

\[(\lambda x. (\lambda x. x \, x) \, x) \, a\]

\(\Rightarrow\) \((\lambda x. x \, x) \, a\)

\(\Rightarrow\) \(a \, a\)

Here, the right most \(x\) is bound to the left most \(\lambda x\) where as the inner two \(x\)’s are bound to the inner \(\lambda\) function.

\[(\lambda x. (\lambda [x][x][x] \, x)\]

Just reiterated, where all boxed variables are related and all underlined variables are related. Here’s one more for practice:

\[(\lambda y. (\lambda x. x \, x) \, y \, x) \, a\]

\(\Rightarrow\) \((\lambda x. x \, x) \, a \, x\)

\(\Rightarrow\) \((a \, a) \, x\)

To help make things more readable, there is this concept of alpha equivalence (\(\alpha\)-equivalence). Alpha equivalence should not change the meaning of the initial statement, but rather make sure things are more readable, or to preserve the semantics of the initial statement.

To do so, an alpha-conversion (\(\alpha\)-conversion) is the process of renaming all the variables that are bound together along with the bounded parameter to a different name.

\[(\lambda x. x) \, a \Rightarrow (\lambda y. y) \, a\]

Again, this does not change the semantics of the expression, but makes things more readable. Consider the C code:
There is no difference between `foo` and `bar`. These two functions are $\alpha$-equivalent. So let's consider our initial statement and alpha-convert it to be more readable:

$$(\lambda x. (\lambda y. x y)) x \Rightarrow (\lambda a. (\lambda b. (\lambda y. b y)) a) x$$

It is important to note that you cannot convert free variables. You can only convert bound variables. Thus $(\lambda x. x) x$ is not alpha equivalent to $(\lambda x. x) a$.

For the most part, this just helps with readability, but sometimes it is important to alpha convert to keep the semantics. Consider the following incorrect reduction:

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

This now reduces to the identity function, but this is incorrect since the initial outer $x$ was free and now becomes bound. To keep the semantics, free variables cannot become bound, and bound variables cannot become free. To make this correct, we must alpha convert:

$$(\lambda y. (\lambda x. y)) x \Rightarrow (\lambda y. (\lambda a. y)) x \Rightarrow (\lambda a. x)$$