Lambda calculus

Now that we've covered the basic notation of programming languages (grammars, logic rules, operational semantics), our goal is to slowly build up a theory of each feature of a Church programming language. That means we'll formally describe the syntax of each feature (by extending a grammar) and also its semantics (by defining new small step rules).

Today, we will look at two core ideas in programming languages: functions, and variables. This is both for pedagogic reasons—understanding these ideas lays the foundation for building a richer programming languages—but also for historical ones, since these two features are the basis of the *lambda calculus*, the original Church language.

Examples

First, I'll show you what the lambda calculus looks like by example, and then we can work through its formal syntax/semantics. The lambda calculus is a programming language with three features: functions, function application, and variables. That's it! No numbers, strings, for loops, modules, and so on. Here's an example function.

$$\lambda\, x\, .\, x$$

The first symbol λ is the greek letter "lambda" (hence the name), which means "function". The first x is a variable which is the argument to the function, followed by a dot, then the body of the function, which in this case just returns the input x. In Javascript, this would equivalently be `function(x) { return x; }`. Functions can be nested, e.g.

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

You can read this as: a function that takes an input x, then returns a function that takes an input y, then calls the function x with the argument y (written as `x(y)` in most languages). In Javascript, this would be:

```
function(x) {
  return function(y) {
    return x(y);
  }
}
```

In the lambda calculus, all functions have one input and one output. For example:

$$\begin{aligned} & (\lambda\, x\, .\, \lambda\, y\, .\, x\ y)\ (\lambda\, z\, .\, z)\ w \\ \mapsto\ & (\lambda\, y\, .\, (\lambda\, z\, .\, z)\ y)\ w \\ \mapsto\ & (\lambda\, z\, .\, z)\ w \\ \mapsto\ & w \end{aligned}$$

Here, calling a function is equivalent to replacing every instance of the argument variable with the argument expression, e.g. replacing x with $\lambda\, z\, .\, z$ in the example above. A few notes on syntactic convention:

1. Function application is *left-associative*, e.g. $x\ y\ z$ is equivalent to as $((x\ y)\ z)$, not $(x\ (y\ z))$.
2. Function application has higher precedence than function definition, e.g. $\lambda\, x\, .\, \lambda\, y\, .\, x\ y$ is equivalent to $\lambda\, x\, .\, (\lambda\, y\, .\, (x\ y))$, not $\lambda\, x\, .\, (\lambda\, y\, .\, x)\ y$.

Formal specification

Just like we did for the arithmetic language, now we'll use the PL metalanguage to formally specify the syntax and semantics of the lambda calculus. First, the grammar:

$$
\begin{array}{rcll}
\textsf{Expression } e ::= & x & & \text{variable} \\
& \mid & \lambda\, x\, .\, e & \text{function definition} \\
& \mid & e_{\textsf{lam}}\ e_{\textsf{arg}} & \text{function application}
\end{array}
$$

That's it! The lambda calculus grammar consists of only three things. Carefully note that here, x is a *metavariable for variables*[1]. Our operational semantics are similarly simple:

$$
\frac{}{\lambda\, x\, .\, e\ \textsf{val}}\ \text{(D-Lam)} \qquad \frac{e_{\textsf{lam}} \mapsto e'_{\textsf{lam}}}{e_{\textsf{lam}}\ e_{\textsf{arg}} \mapsto e'_{\textsf{lam}}\ e_{\textsf{arg}}}\ \text{(D-App-Step)} \qquad \frac{}{(\lambda\, x\, .\, e_{\textsf{lam}})\ e_{\textsf{arg}} \mapsto [x \to e_{\textsf{arg}}]\ e_{\textsf{lam}}}\ \text{(D-App-Sub)}
$$

The first rule lays down the iron law of functional programming: **functions are values** (at my alma mater, we had jackets enshrining this ideal). An expression of the form $\lambda\, x\, .\, e$ cannot be evaluated any further until it has an argument applied to it. The second rule says: step the left expression $e_{\textsf{lam}}$ until it becomes a function.

The last rule introduces a new construct: substitution. The metasyntax $[x \to e_{\textsf{arg}}]\ e_{\textsf{lam}}$ means "substitute all instances of x in $e_{\textsf{lam}}$ with $e_{\textsf{arg}}$".

Substitution

To understand how substitution works, we need to understand the essence of variables in the lambda calculus. The basic intuition can be summarized as, given the following expression:

$$\lambda\, x\, .\, \lambda\, x\, .\, x$$

Which argument x does the innermost x refer to? Does $(\lambda\, x\, .\, \lambda\, x\, .\, x)\ y\ z$ evaluate to y or z? The short answer is z, and the reason is because substitution uses the rules of *lexical scoping*.

Free vs. bound variables

When a variable is used in an expression under a function with a variable of the same name, then the variable is *bound*. For example, in $\lambda\, x\, .\, x$, the inner variable x is bound to the function argument x. In contrast, if a variable is not bound, then it is *free*, e.g. y in $\lambda\, x\, .\, y$.

Suppose we had a function $FV : \textsf{Expression} \to \{\textsf{Variable}\}$ that returns a set of all the free variables in an expression. It would look like:

$$
\begin{array}{ll}
FV(x) & = \{x\} \\
FV(x\ y) & = \{x, y\} \\
FV(\lambda\, x\, .\, x) & = \{\} \\
FV(\lambda\, x\, .\, x\ y) & = \{y\} \\
FV((\lambda\, x\, .\, x\ y)\ x) & = \{x, y\}
\end{array}
$$

A variable usage binds to the closest argument in its enclosing functions. So in $\lambda\, x\, .\, \lambda\, x\, .\, x$, we would say the inner argument x "shadows" the outer argument x. Hence, the function is logically equivalent to:

$$\lambda\, x\, .\, \lambda\, y\, .\, y$$

This idea is called "alpha-equivalence", where two functions are equivalent up to a renaming of bound variables, written as:

$$\lambda\, x\, .\, \lambda\, x\, .\, x \equiv_\alpha \lambda\, x\, .\, \lambda\, y\, .\, y$$

Although the functions are not literally the same syntax, they are *effectively* the same function since they will evaluate the same way.

Substitution is replacing free variables

The substitution operation $[x \to e_2]\ e_1$ replaces all *free variables* x in e_1 with e_2. Here's a few examples:

$$
\begin{aligned}
&[x \to y]\ x && = y \\
&[x \to y]\ z && = z \\
&[x \to y]\ \lambda\, z\ .\ x && = \lambda\, z\ .\ y \\
&[x \to y]\ \lambda\, y\ .\ x\ y && = \lambda\, y'\ .\ y\ y' \\
&[x \to y]\ x\ (\lambda\, x\ .\ x) && = y\ (\lambda\, x\ .\ x)
\end{aligned}
$$

Read each example closely, as it exhibits a different kind of behavior. In particular, note that in the fourth example, the function argument had to be renamed to preserve the proper semantics. If this seems a little complicated, it is—name resolution is a hard (and underappreciated!) problem. Alternative variable schemes like de Bruijn indices have been proposed to avoid these kinds of issues.

Finally, this brings us back to function application. Recall the rule from above:

$$
\frac{}{(\lambda\, x\ .\ e_{\mathsf{lam}})\ e_{\mathsf{arg}} \mapsto [x \to e_{\mathsf{arg}}]\ e_{\mathsf{lam}}}\ \text{(D-App-Sub)}
$$

We can now read this as: to call a function $\lambda\, x\ .\ e_{\mathsf{lam}}$ with argument e_{arg}, replace all free variables x in e_{lam} with e_{arg}. With that, we have a full definition of the semantics for the lambda calculus. Huzzah!

Stuck states

Our semantics still have some odd edge cases we need to address. Remember that our goal for operational semantics is to reduce expressions until they become values. Last time, in the context of our arithmetic language, every expression in the language could be reduced to a value, which we proved with the totality theorem. However, consider a lambda calculus expression consisting of a single free variable, e.g. x. Is this a value? No, only functions are values. Is this reducible? No, we never defined a semantics for plain free variables.

Instead, we would say the expression x is in a "stuck state." It is neither a value, nor do any semantics apply. This is simply the way of life in the lambda calculus. It's easy to accidentally produce expressions that will end up in a stuck state, like

$$
(\lambda\, x\ .\ x)\ z
$$

This is very similar to the notion of undefined behavior in languages like C, except instead of choosing continue execution with arbitrary semantics (e.g. we could say "replace every free variable in a stuck state with $(\lambda\, x\ .\ x)$"), we choose to throw our hands in the air and say "can't do anything else, not my problem."

> A interesting question: can we identify expressions that are going to be stuck ahead of time, and flag them before they ever run? The short answer is no, and the long answer is "because of the Halting Problem." You may ask, but isn't the Halting Problem defined over Turing machines, not the lambda calculus? Turns out they're computationally equivalent. You'll work through why on the homework.

The point of solving this problem, of avoiding stuck states, is the purpose of type systems, which we will explore next lecture). A type system is a tool for constructing proofs that a program will run "safely" (to be defined), and we can define these proofs *without ever running the program.*

Distilling abstraction

So why are we talking about the lambda calculus, anyway? It's obviously not a language anyone would seriously use in a real world context. If we're supposed to bridge our theory to real world systems, this doesn't seem like a strong start.

The key idea is that the lambda calculus is an aggressively simple model of computation that is still surprisingly powerful. Just take a look at the formal definition of a Turing machine. They have a head, a tape, a state machine, an alphabet, you name it. That's a lot of complexity when you could just use functions! This simplicity lends two benefits:

1. **Distilling the essence of functions and variables.**

A function is a critical computational concept in any programming language, since it is the basis of abstraction: taking something concrete, like $1+2$, and making it abstract, like $\lambda\, n\, .\ 1+n$. Just as in standard mathematics, a function in the lambda calculus is beautifully simple: one input, one output, and a function call just replaces its argument variable with its argument expression.

However, in practice, programming languages often weight down their functions with extra "features." Most languages have some baked-in idea that a function takes multiple arguments, and that a function can optionally return a value, like:

```
void f(int a, int b) { .. }
```

Some languages then have "multiple returns" as a feature, or named parameters, or default parameters. Nearly every language has recursion built-in to functions. By contrast, in the lambda calculus, each of these ideas is embodied by extending the language with a *new* and *orthogonal* concept[2]. For example, consider multiple function arguments. If we extend our lambda calculus with a concept of *tuples*, then we could just as easily write something like (reversing a pair):

$$\lambda\, (a,b)\, .\ (b,a)$$

By separating tuples from functions, we've done three things.

1. Increased the expressive power of our language. We can use tuples anywhere, not just functions!
2. Increased the clarity of our language. Rather than have certain features only work with certain other ones, tuples can be used uniformly/consistently across the language, not just with functions.
3. Minimally increased complexity of analysis. If we had some previous code analyzing functions, then that code doesn't need to change to consider a function with multiple arguments, it just needs to separately understand the idea of tuples.

Consider as well the same case for variables. In a standard Turing languages, variables always do double-duty: they are both placeholders for arbitrary values, and they are mutable slots in persistent storage that change throughout the program. In the lambda calculus, variables are only the former: they represent an unknown value, but always a *single* value that will never change.

Some Church languages include the notion of a reference that allows the use of persistent mutable slots, but this is necessarily orthogonal to variables. The two features can be understood and analyzed in isolation.

2. **Providing a basis for program analysis.**

When trying to define a new kind of programming language feature, it is convenient to place it in the context of the simplest language possible. For example, if I had a great idea on a new kind of communication protocol, I could define it in C. But to then analyze my idea and prove something like "this protocol will never segfault" would require understanding my idea's interaction with everything in the C standard—recall from the first lecture why this is terrifying.

Instead, the PL theory community mostly uses the lambda calculus as a basis for research. I randomly selected a few preprints from the latest proceedings of POPL, the premier PL theory conference, and you can see the lambda calculus underpinnings: section 3.1, section 2, section 3.1.

In this class, we will look at some basic PL theory extending from the lambda calculus. This should hopefully equip you to more effectively dive deeper into academic PL if you should so choose.

---

1. You may notice that we did not define a grammar rule for Variable x—in general, we're assuming variables syntactically follow the same conventions as normal mathematics. Here, they will always be one letter and always italicized. ↩
2. "Extending" the language doesn't necessarily mean changing the grammar or the semantics, this could just be adding new concepts as libraries within the language. You'll see this in the context of the lambda calculus on Assignment 1. ↩