CIS 425 - Week 2 - Lecture 3

Lindsay Luallen & Bethany Van Meter

8 October 2019

1 Lambda Calculus & JavaScript

Lambda calculus is the core of JavaScript. This core can be described by the following grammar:

$$M ::= x \mid (function(x)\{return\ M;\}) \mid M(M)$$

which reads as :

M is either a variable x or a function or M applied to M

Note the following:

- The only nonterminal is M.
- x is the representation of any name for a variable. In other words, it is not the case that you only have one identifier available to write your program.
- When we write M(M) (the same for e+e) it is not the case that the function and the argument need to be the same. M stands for any program generated from the grammar.
- The function (also called abstraction) does not have a name, it corresponds to an *anonymous function*; this is now very popular, it is present in Java 8, Swift, Python, Ruby, C#, Delphi, C++, PHP.

Example 1. $function(x)\{return\ x;\}$ is the identity function and is equivalent to f(x)=x. It is a program generated from our grammar.

Let us now ask the question: can we simplify the grammar by eliminating some terminal symbols. We start by eliminating parenthesis, to obtain:

$$M ::= x \mid function(x)\{return\ M;\} \mid MM$$

Why is JavaScript not defined this way? Suppose we have:

$$function(x)\{return\ x;\}z\ y$$

Is this a string generated from this grammar? Yes, it is. However, there are different ways to parse this string. Consider even the simpler example

$$x\ y\ z$$

This also is a legal string. However, it can be interpreted in two ways:

$$(x\ y)\ z \quad \text{or} \quad x\ (y\ z)$$

Since there are two ways to parse the string $x\ y\ z$, it means that our simplified grammar is ambiguous. Now we understand why JavaScript has the extra syntax. The extra syntax allows for the creation of unique parse trees. Indeed, the above strings will be written in JavaScript as

$$x\ (y)\ (z) \qquad x\ (y\ (z))$$

Some of these ambiguities can be fixed also by making assumptions such as assuming left associativity. The more semantics is embedded into the compiler, the more syntax we can eliminate from the grammar.

2 Lambda Calculus

Let's go back to the beginning of the 19th century. Set theory was born, which was very promising because if offered a common foundation to all fields of mathematics. However, consider the following set definition:

$$R = \{x \mid x \notin x\}$$

We have:

$$R \in R \qquad \text{if and only if} \qquad R \notin R$$

In other words, we have a sentence which is both true and false (i.e. a paradox). This caused some panic, since it seemed that all mathematics lied on shaky grounds.... Mathematicians, like David Hilbert (1862-1943) and Gerhard Gentzen (1909-1945), set out to devise a complete and finite set of axioms from which all true statements can be derived. Unfortunately, Kurt Godel (1906-1978) in 1935 proved that the goal wasn't attainable; even all theorems of arithmetic cannot be derived from a finite set of axioms. However, it turned out that the systems that were developed (even before the computer was invented) have something to do with *computation*.

One notion which needed to be clarified was the notion of a *variable*, since that is often present in mathematical notations such as

$$\{x \mid P(x)\} \quad \forall x.P(x) \quad \exists x.P(x) \quad \bigcup_{i=1}^{n} a_i \quad \prod_{i=1}^{n} a_i \quad \sum_{i=1}^{n} a_i \quad \lim_{n\to\infty} n^2 \quad \int_0^\infty x^2\,dx$$

To capture this notion and to precisely formalise the notion of a function, Alonzo Church (1903-1995) invented the λ-calculus in 1932[1].

Lambda-calculus terms (i.e. programs) are defined as follows:

$$\begin{array}{lcll} M & ::= & x & \text{variable} \\ & | & \lambda x.M & \text{abstraction} \\ & | & M\ M & \text{application} \end{array}$$

Note how this grammar is obtained by applying a series of simplifications to the grammar of the core of JavaScript, as given below:
(1) $M ::= x \mid function(x)\{return x;\} \mid M(M)$
(2) $M ::= x \mid \lambda(x)\{M\} \mid MM$
(3) $M ::= x \mid \lambda x\{M\} \mid MM$
(4) $M ::= x \mid \lambda x.M \mid MM$

Note how the "." separates the formal parameter from M. It is remarkable that three productions define a computational model which is Turing complete, that is equivalent to Turing machines[2]. This was proved in 1937 by Turing, both λ-calculus and Turing machines define the same set of computable functions. Note that modern machines are just blown up Turing machines, and functional languages are just extensions of λ-calculus.

Notation: in a lambda-abstraction $\lambda x.M$ we call M the body of the lambda-abstraction, and refer to x as the formal parameter. In a term such as $(\lambda x.M)\ N$ we refer to N as the actual parameter.

Example 2. Consider the term $\lambda x.x$, is it a legal term? As discussed in week 1, we can give a derivation from the root symbol of the grammar:

$$M \to \lambda x.M \to \lambda x.x$$

or we can give the parse tree as given in Figure 1. There is also a more compact representation, called the *abstract syntax tree*, which better captures the structure of a sentence. In particular, the tree only contains terminal symbols, as shown in Figure 2. The parse tree is created first by the compiler to guarantee that the program follow the syntactic restrictions, the abstract syntax tree is then generated by getting rid of the non-terminal in the tree.

Looking back at the grammar of lambda-calculus:

$$M ::= x \mid \lambda x.M \mid MM$$

we can see it as defining the following trees:

[1] Church was a professor at Princeton (1929-1967) and at UCLA (1967-1990). He had a few successful graduate students, including: Stephen Kleene (regular expressions), Michael O. Rabin (non-deterministic automata), Dana Scott (formal programming languages), and Turing (Turing machines)

[2] Turing machines were invented in 1936

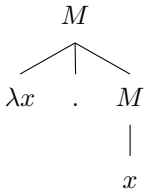
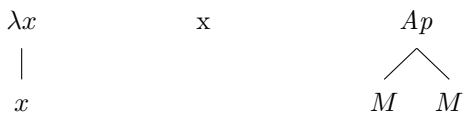


Figure 1: Parse tree for $\lambda x.x$

Figure 2: Abstract syntax tree for $\lambda x.x$

function or a variable or an application:



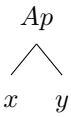
Ap just means, in the case of the application tree above, M (the first M) applied to M (the second M).

Example 3.

- Is xy a legal program [3]? We can take the steps below by starting at the root:

$$M \to MM \to x\ M \to x\ y$$

So this shows that this is legal! The abstract syntax tree of $x\ y$ is :



- Consider the term $\lambda x.x\ y$. How can it be read? There are two ways:

$$(\lambda x.(x\ y)) \qquad \text{or} \qquad (\lambda x.x)\ y$$

These two readings are depicted in the following abstract syntax trees:

[3]Note that xy can be interpreted as an identifier named xy. If we mean x applied to y then it is better to separate x from y as $x\ y$

which in Javascript would be written as

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

We need to assume a convention which guarantees a unique way to parse that term, and the convention is that the scope of a lambda extends as far as possible to the right-hand side. Thus, the convention is to read $\lambda x.x\ y$ as $\lambda x.(x\ y)$.

To summarise, in lambda calculus, there are two rules to follow when reading expressions:

(1) Application is read left to right (left associative)

(2) The scope of a λ abstraction extends as far right as possible to the right (seen in the previous example)

3 Operational Semantics of Lambda Calculus

In Week 1 we saw how to give semantics of arithmetic expressions, by giving the axiom:

$$n + m \to p \qquad \text{where } p = +(m, n)$$

plus inferences rules explaining where to apply the axiom. In our setting, we are interested in explaining how function application works. Consider $(\lambda x.x)\ z$, you would say that the result of that application is z; you obtain that result by substituting z for x in the body of the lambda-abstraction.

This is captured by the following β-axiom:

$$(\lambda x.M)\ N \to [N/x]M$$

where $[N/x]M$ is read as "substitute N for each occurrence of variable x in M. Remember that:

- N is the actual parameter
- x is the formal parameter
- So, we would do the substitution in M and then execute the result of that substitution

Example 4. • Apply the β rule to $(\lambda x.x + x)(2 + 4)$

We can start by using the rule defined above which reads as substitute $2 + 4$ for x in $x + x$:

$$(\lambda x.x + x)(2 + 4) \rightarrow [2 + 4/x]x + x = (2 + 4) + (2 + 4)$$

This is also called inlining in compilers. The compiler does the β rule often to avoid the cost of function application.

- How would we apply inlining (β rule) to:

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

$\lambda x.\lambda y.x$ is a function which when invoked returns another function . Our first approximation is:

$$(\lambda x.\lambda y.x)y \rightarrow [y/x]\lambda y.x = \lambda y.y \tag{1}$$

We will come back to the above reduction shortly, and ask if the reduction is correct.

Let us next ask a question. Consider the following functions

$$\lambda x.x \qquad \text{and } \lambda w.w$$

Are these functions the same, even though they differ in the bound variable? Indeed, they are the same as captured by the following axiom.

Renaming

We can change the names of the formal parameters, this is expressed as follows[4]:

$\lambda x.M = \lambda z.[z/x]M$ where z is a fresh variables (one that isn't used elsewhere)

Example 5. •

$$\lambda x.x = [w/x]\lambda w.w$$

•

$$\lambda x.x\ x = \lambda z.[z/x]x\ x = \lambda z.z\ z$$

•

$$\lambda x.\lambda y.x\ y = \lambda w.\lambda z.[w/x][z/y]w\ z$$

- Are the following functions equal?

$$\lambda x.\lambda y.x \stackrel{?}{=} \lambda y.\lambda y.y$$

[4]The axiom is presented either in terms of equality or reduction. In here we use equality.

No. We can rename variables but we need to pick a fresh variable. Note that y is already in the program. Instead, we could have:

$$\lambda x.\lambda y.x = \lambda x.\lambda z.x$$

Example 6. Are these the same?

$$(\lambda x.\lambda y.x)y \stackrel{?}{=} (\lambda x.\lambda z.x)y$$

Yes they are the same by applying the α axiom and renaming the bound variable y to z. Now let us applying the β rule to both programs:

$$(\lambda x.\lambda y.x)y \to [y/x]\lambda y.x = \lambda y.y$$

$$(\lambda x.\lambda z.x)y \to [y/x]\lambda z.x = \lambda z.y$$

Notice that $\lambda y.y$ and $\lambda z.y$ are different. We started with something that we claimed was equal but ended up with something different. We have a problem, we must have done something wrong. Where is the error? Either we claimed that the original expressions were equal when they weren't, or they are equal and we did something wrong in performing the function application. It will turn out that the β reduction was done incorrectly, so we reduced the expression incorrectly and the expressions were equal at the beginning. So the reduction given in 1 is wrong! This is corrected and explained more in the next lecture!