1. History of the Lambda Calculus

- The lambda calculus was introduced in the 1930s by the logician Alonzo Church at Princeton University as a mathematical system for defining computable functions.
- Alan Turing (who was Church's PhD student) showed that the lambda calculus is equivalent in definitional power to Turing machines.
- The lambda calculus serves as the model of computation for functional programming languages and has applications to artificial intelligence, proof systems, and logic.
- The programming language Lisp was developed by John McCarthy at MIT in 1958 around the lambda calculus.
- Haskell, considered by many as one of the purest functional programming languages, was developed by Simon Peyton Jones, Paul Houdak, Phil Wadler and others in the late 1980s and early 90s.
- Many established languages such as C++ and Java and many more recent languages such as Python and Ruby have adopted lambda expressions as anonymous functions from the lambda calculus.
- Because of its simplicity, lambda calculus is a very useful tool for the study and analysis of programming languages.

2. CFG for the Lambda Calculus

- The central concept in lambda calculus is an expression which we can think of as a program that when evaluated using beta reductions returns a result consisting of another lambda calculus expression.
- Here is a grammar for lambda expressions:

```
expr → λ variable . expr | expr expr | variable | ( expr ) | constant
```

- A `variable` is an identifier.
- A `constant` is a built-in function such as addition or multiplication, or a constant such as an integer or boolean. However, as we shall see, all programming language constructs can be implemented as functions with only the first four productions so constants are unnecessary. However, we will often use constants for notational clarity.

3. Function Abstraction

- A function abstraction, often called a lambda abstraction, is a lambda expression that defines a function.
- A function abstraction consists of four parts: a lambda followed by a variable, a period, and then an expression as in `λx.expr`.
- In the function abstraction `λx.expr` the variable `x` is the formal parameter of the function and `expr` is the body of the function.
- For example, the function abstraction $\lambda x. + x\ 1$ defines a function of x that adds x to 1. Parentheses can be added to lambda expressions for clarity. Thus,

we could have written this function abstraction as $\lambda x.(+\ x\ 1)$ or even as $(\lambda x.\ (+\ x\ 1))$.

- In C this function definition might be written as

```
int addOne (int x)
{
  return (x + 1);
}
```

- Note that unlike C the lambda abstraction does not give a name to the function. The lambda expression itself is the function.
- We say that `λx.expr` *binds* the variable `x` in `expr` and that `expr` is the *scope* of the variable.

4. Function Application

- A function application, often called a lambda application, consists of an expression followed by an expression: `expr expr`. The first expression is a function abstraction and the second expression is the argument to which the function is applied. All functions in lambda calculus have exactly one argument. Multiple-argument functions are represented by currying, which we shall explain below.
- For example, the lambda expression $\lambda x.\ (+\ x\ 1)\ 2$ is an application of the function $\lambda x.\ (+\ x\ 1)$ to the argument 2.
- This function application $\lambda x.\ (+\ x\ 1)\ 2$ can be evaluated by substituting the argument 2 for the formal parameter x in the body $(+\ x\ 1)$. Doing this we get $(+\ 2\ 1)$. This substitution is called a *beta reduction*.
- Beta reductions are like macro substitutions in C. To do beta reductions correctly we may need to rename bound variables in lambda expressions to avoid name clashes.
- Function application associates left-to-right; thus, $f\ g\ h = (f\ g)h$.
- Function application binds more tightly than λ; thus, $\lambda x.\ f\ g\ x = (\lambda x.\ (f\ g)x)$.
- Functions in the lambda calculus are first-class citizens; that is to say, functions can be used as arguments to functions and functions can return functions as results.

5. Free and Bound Variables

- In the function definition $\lambda x.x$ the variable x in the body of the definition (the second x) is *bound* because its first occurrence in the definition is λx.
- A variable that is not bound in `expr` is said to be *free* in `expr`. In the function $(\lambda x.xy)$, the variable x in the body of the function is bound and the variable y is free.
- Every variable in a lambda expression is either bound or free. Bound and free variables have quite a different status in functions.
- In the expression $(\lambda x.x)(\lambda y.yx)$:
    - The variable x in the body of the leftmost expression is bound to the first lambda.

    - The variable *y* in the body of the second expression is bound to the second lambda.
    - The variable *x* in the body of the second expression is free.
    - Note that *x* in second expression is independent of the *x* in the first expression.
- In the expression $(\lambda x.xy)(\lambda y.y)$:
    - The variable *y* in the body of the leftmost expression is free.
    - The variable *y* in the body of the second expression is bound to the second lambda.
- Given an expression *e*, the following rules define FV(*e*), the set of free variables in *e*:
    - If *e* is a variable *x*, then FV(*e*) = {*x*}.
    - If *e* is of the form $\lambda x.y$, then FV(*e*) = FV(*y*) − {*x*}.
    - If *e* is of the form *xy*, then FV(*e*) = FV(*x*) ∪ FV(*y*).
- An expression with no free variables is said to be *closed*.

6. Beta Reductions

- A function application `λx.e f` is evaluated by substituting the argument `f` for all free occurrences of the formal parameter `x` in the body `e` of the function definition `λx.e`.
- We will use the notation `[f/x]e` to indicate that `f` is to be substituted for all free occurrences of `x` in the expression `e`.
- Examples:
    1. $(\lambda x.x)y \rightarrow [y/x]x = y$.
    2. $(\lambda x.xzx)y \rightarrow [y/x]xzx = yzy$.
    3. $(\lambda x.z)y \rightarrow [y/x]z = z$ since the formal parameter *x* does not appear in the body *z*.
- This substitution in a function application is called a *beta reduction* and we use a right arrow to indicate it.
- If expr1 → expr2, we say expr1 *reduces* to expr2 in one step.
- In general, `(λx.e)f → [f/x]e` means that applying the function `(λx.e)` to the argument expression `f` reduces to the expression `[f/x]e` where the argument expression `f` is substituted for the function's formal parameter `x` in the function body `e`.
- A lambda calculus expression (aka a "program") is "run" by computing a final result by repeatly applying beta reductions. We use →* to denote the reflexive and transitive closure of →; that is, zero or more applications of beta reductions.
- Examples:
    1. $(\lambda x.x)y \rightarrow y$ (illustrating that $\lambda x.x$ is the identity function).
    2. $(\lambda x.xx)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y.y)$; thus, we can write $(\lambda x.xx)(\lambda y.y) \rightarrow^* (\lambda y.y)$. Note that here we have applied a function to a function as an argument and the result is a function.

7. Evaluating a Lambda Expression

- A lambda calculus expression can be thought of as a program which can be executed by evaluating it. Evaluation is done by repeatedly finding a reducible expression (called a redex) and reducing it by a function evaluation until there are no more redexes.
- Example 1: The lambda expression $(\lambda x.x)y$ in its entirety is a redex that reduces to y.
- Example 2: The lambda expression (+ (* 1 2) (− 4 3)) has two redexes: (* 1 2) and (− 4 3). If we choose to reduce the first redex, we get (+ 2 (− 4 3)). We can then reduce (+ 2 (− 4 3)) to get (+ 2 1). Finally we can reduce (+ 2 1) to get 3.
- Note that if we had chosen the second redex to evaluate first, we would have ended up with the same result:

$$(+ (* 1 2) (- 4 3)) \rightarrow (+ (* 1 2) 1) \rightarrow (+ 2 1) \rightarrow 3.$$

8. Currying

- All functions in lambda calculus are prefix and take exactly one argument.
- If we want to apply a function to more than one argument, we can use a technique called *currying* that treats a function applied to more than one argument to a sequence of applications of one-argument functions. For example, to express the sum of 1 and 2 we can write (+ 1 2) as ((+ 1) 2) where the expression (+ 1) denotes the function that adds 1 to its argument. Thus ((+ 1) 2) means the function + is applied to the argument 1 and the result is a function (+ 1) that adds 1 to its argument: (+ 1 2) = ((+ 1) 2) → 3

9. Renaming Bound Variables by Alpha Reduction

- The name of a formal parameter in a function definition is arbitrary. We can use any variable to name a parameter, so that the function $\lambda x.x$ is equivalent to $\lambda y.y$ and $\lambda z.z$. This kind of renaming is called *alpha reduction*.
- Note that we cannot rename free variables in expressions.
- Also note that we cannot change the name of a bound variable in an expression to conflict with the name of a free variable in that expression.

10. Eta Conversion

- The two lambda expressions $(\lambda x.+ 1\ x)$ and (+ 1) are equivalent in the sense that these expressions behave in exactly the same way when they are applied to an argument — they add 1 to it. η-conversion is a rule that expresses this equivalence.
- In general, if x does not occur free in the function F, then $(\lambda x.F\ x)$ is η-convertible to F.

11. Substitutions

- For a beta reduction, we introduced the notation `[f/x]e` to indicate that the expression `f` is to be substituted for all free occurrences of the formal parameter `x` in the expression `e`:

```
(λx.e) f → [f/x]e
```

- To avoid name clashes in a substitution `[f/x]e`, first rename the bound variables in `e` and `f` so they become distinct. Then perform the textual substituion of `f` for `x` in `e`.
  - For example, consider the substitution `[y(λx.x)/x] λy.(λx.x)yx`.
  - After renaming all the bound variables to make them all distinct we get `[y(λu.u)/x] λv.(λw.w)vx`.
  - Then doing the substitution we get `λv.(λw.w)v(y(λu.u))`.
- The rules for substitution are as follows. We assume `x` and `y` are distinct variables, and `e`, `f` and `g` are expressions.
  - For variables

```
[e/x]x = e
[e/x]y = y
```

  - For function applications

```
[e/x](f g) = ([e/x]f) ([e/x]g)
```

  - For function abstractions

```
[e/x](λx.f) = λx.f
```

    `[e/x](λy.f) = λy.[e/x]f`, provided `y` is not free in `e` (this is called the "freshness" condition).

- Examples:
  1. `[y/y](λx.x) = λx.x`
  2. `[y/x]((λx.y) x) = ([y/x](λx.y)) ([y/x]x) = (λx.y)y`
  3. Note that the freshness condition does not allow us to make the substitution `[y/x](λy.x) = λy.([y/x]x) = λy.y` because `y` is free in the expression `y`.

12. Disambiguating Lambda Expressions

- The grammar we gave in section 4 for lambda expressions is ambiguous. A few simple rules will remove the ambiguities.
- Function application is left associative: $f\ g\ h = ((f\ g)\ h)$
- Function application binds more tightly than lambda: $\lambda x.f\ g\ x = (\lambda x.(f\ g)\ x)$
- The body in a function abstraction extends as far to the right as possible: $\lambda x.\ +\ x\ 1 = \lambda x.\ (+\ x\ 1)$.
- We can always use parentheses to remove any ambiguities.