# CMPSC 461: Programming Language Concepts

Lecture Note 1: The Lambda Calculus

Prepared by Professor Danfeng Zhang

## 1 Introduction

Lambda calculus is a notation for describing mathematical functions and programs. It is a mathematical system for studying the interaction of functional abstraction and functional application. It captures some of the essential, common features of a wide variety of programming languages. Because it directly supports abstraction, it is a much more natural model of universal computation than an imperative language.

## 2 Lambda calculus

### 2.1 Syntax

Let *Var* be a countable set of variables. A $\lambda$-calculus term $t$ is:

- a variable $x \in$ *Var*,
- a function $t_0$ applied to an argument $t_1$, written $t_0\ t_1$, or
- a lambda term: an expression $\lambda x.\ t$ representing a function with input parameter $x$ and body $t$. Where a mathematician might write $x \to x^2$, in the $\lambda$-calculus we would write $\lambda x.\ x^2$.

More formally, we can define the lambda calculus in context-free grammar (CFG[1]) as follows:

$$\textit{Term} \qquad t ::= \ x \mid t_0\ t_1 \mid \lambda x.\ t$$

Parentheses in lambda terms are used just for grouping; they have no meaning on their own. Like other familiar binding constructs from mathematics (e.g., sums, integrals), lambda terms are greedy, extending as far to the right as they can. Therefore, the term $\lambda x.\ x\ \lambda y.\ y$ is the same as $\lambda x.\ (x\ (\lambda y.\ y))$, not $(\lambda x.\ x)\ (\lambda y.\ y)$. In lambda calculus, function application is regarded as left-associative. In other words, $t_1\ t_2\ t_3$ is the same as $(t_1\ t_2)\ t_3$. Moreover, we use a space to separate two terms: $t_0\ t_1$. Notice that $t_0\ t_1$ means applying function $t_0$ to $t_1$, but not $t_0$ *multiplies* $t_1$.

For simplicity, multiple variables may be placed after the lambda, and this is considered shorthand for having a lambda in front of each variable. For example, we write $(\lambda x\ y.\ t)$ as shorthand for $(\lambda x.\ \lambda y.\ t)$. This shorthand is an example of syntactic sugar. The process of removing it in this instance is called currying (named after Haskell Curry, a professor in the Mathematics department at Penn State from 1929 to 1966).

We can apply a curried function like $\lambda x.\ \lambda y.\ x$ one argument at a time. Applying it to one argument results in a function that takes in a value for $y$ and returns the first parameter $x$. A function can take another function as a parameter in $\lambda$-calculus. For example, $(\lambda f\ x.\ f\ x)$ takes a function $f$ and a parameter, and then applies $f$ to the parameter. So $(\lambda f\ x.\ f\ x)\ (\lambda y.\ y)$ should evaluate to $\lambda x.\ x$. As this suggests, functions are just ordinary values, and can be the results of functions or passed as arguments to functions. Thus, in the lambda calculus, functions are first-class values. Lambda terms serve both as functions and data. A function that can accept other functions is called a *higher-order function*. Higher-order functions are a powerful means of abstraction and one of the best tools to master in functional programming languages.

[1] Covered later in this course.

### 2.2 Variable Binding

In order to define the meaning of a $\lambda$-term, we first need to distinguish bound variables and free variables. Occurrences of variables in a $\lambda$-term can be *bound* or *free*. In the $\lambda$-term $\lambda x.\ t$, the lambda abstraction operator $\lambda x$ binds all the free occurrences of $x$ in $t$. More precisely, let $\texttt{FV}$ be a function that takes an arbitrary $\lambda$-term and returns all free variables in that term. $\texttt{FV}$ is defined inductively as follows:

$$\texttt{FV}(x) = \{x\} \qquad \texttt{FV}(t_0\ t_1) = \texttt{FV}(t_0) \cup \texttt{FV}(t_1) \qquad \texttt{FV}(\lambda x.\ t) = \texttt{FV}(t) - \{x\}$$

Let $\mathit{Var}(t)$ be all variables used in term $t$, then the bound variables of $t$, written $\texttt{BD}(t)$, are simply $\mathit{Var}(t) - \texttt{FV}(t)$.

This definition is recursive, but notice that in each of the three cases, the right-hand side defines the value of $\texttt{FV}(t)$ in terms of proper subterms of $t$. Hence, computing free variables as above always terminates.

Each variable in a $\lambda$-term is either bound or free. A bound variable $x$ is defined by the closest $\lambda$ operator $\lambda x$. For example, $\lambda x.\ \lambda x.\ x$ is a function that takes two parameters and returns the second, since the innermost $x$ is bound to the second lambda.

### 2.3 Closed terms and Stoy diagrams

The *scope* (visible code fragment) of $\lambda x$ in $\lambda x.\ t$ is $t$. This is called lexical (static) scoping; the variable's scope is defined by the text of the program[2]. It is "lexical" because it is possible to determine its scope before the program runs by inspecting the program text. A term is *closed* if all variables are bound. A term is *open* if it is not closed.

In lambda calculus, the name of a bound variable is irrelevant. That is, $\lambda x.\ x$ and $\lambda y.\ y$ are identical terms. This is more obvious in a Stoy diagram. We can create a Stoy diagram (after Joseph Stoy) for a closed term in the following manner. Instead of writing a term with variable names, we write dots to represent the variables and connect variables with the same binding with edges. For example, the term $\lambda x.\ (\lambda y.\ (\lambda x.\ x\ y)\ x)\ x)$ has the following Stoy diagram:


## 3 Capture-Avoiding Substitution and Reductions

Now we see how to evaluate a lambda term. Before that, we will introduce a building block for reductions, called capture-avoiding substitution.

### 3.1 Capture-avoiding substitution

Replacing a variable with a $\lambda$-term is a basic operation in $\lambda$-calculus. We use notation $t\{t'/x\}$ to represent substituting all *free* occurrences of $x$ in $t$ with $t'$. For instance, by definition, we have $x\{y/x\} = y$ and $(x\ x)\{y/x\} = (y\ y)$.

However, one subtlety in $\lambda$-calculus is that the substitution must avoid corner cases where free variables in $t'$ is already *captured* in $t$ (i.e., there is some $\lambda y$ in $t$ and $y \in \texttt{FV}(t')$). For instance, $(\lambda y.\ x)\{y/x\} \neq \lambda y.\ y$, since $y$ is free, but it is captured in $\lambda y.\ x$. Intuitively, the two lambda terms before and after substitution are not the same since before the substitution, the function body $x$ is not bound, but after the substitution, it is bound to $\lambda y$.

[2] We will discuss scopes in more details later.

More formally, the notation $t\{t'/x\}$ is called *capture-avoiding substitution*. In particular, it means:

Replace all free occurrences of $x$ in $t$ with $t'$,

if no free variable in $t'$ is captured in $t$ (i.e., there is no $\lambda y$ in $t$ for any free variable $y$ in $t'$)

### 3.2 $\alpha$-reduction

The names of the bound variables in a $\lambda$-term do not really matter. Renaming bound variables is known as an $\alpha$-reduction. In an $\alpha$-reduction, the new bound variable must be chosen so as to avoid capture (i.e., to avoid introducing a variable that is free). If a term $\alpha$-reduces to another term, then the two terms are said to be $\alpha$-equivalent. This defines an equivalence relation on the set of terms, denoted $t_1 =_\alpha t_2$. An $\alpha$-reduction doesn't really make computational progress, so it is often referred to as $\alpha$-renaming.

Recall the definition of free variables $\texttt{FV}(t)$ of a term $t$. To define $\alpha$-reduction more formally, we have

$$\lambda x.\ t =_\alpha \lambda y.\ t\{y/x\} \text{ when } y \notin \texttt{FV}(t)$$

Note that we used the capture-avoiding substitution in the definition, to make sure that $y$ is not *captured* in $t$ (i.e., there is no $\lambda y$ in $t$). The condition $y \notin \texttt{FV}(t)$ is to avoid the capture of a free occurrences of $y$ in $t$ as a result of the renaming. For instance, $\lambda x.\ y$ is not $\alpha$-equivalent to $\lambda y.\ y$, since $y \in \texttt{FV}(y)$.

When a capture-avoiding substitution involves an captured variable, we will need to remove the lambda that captures the new variable via $\alpha$ reduction. For example, $y$ is captured by $\lambda y$ in $(\lambda y.\ x)\{y/x\}$. To proceed, we can rename $y$ to $z$ via $\alpha$-reduction: $(\lambda y.\ x) =_\alpha \lambda z.\ x$. Hence, $(\lambda y.\ x)\{y/x\} = (\lambda z.\ x)\{y/x\} = \lambda z.\ y$. Although new variables are rarely captured in the substitution, we need to use the capture-avoiding substitution to derive the correct terms after reduction.

### 3.3 $\beta$-reduction

The main computational rule in $\lambda$-calculus is called $\beta$-reduction. This rule applies whenever there is a subterm of the form $(\lambda x.\ t)\ t'$ representing the application of a function $\lambda x.\ t$ to an argument $t'$. This is done by capture-avoiding substitution. More specifically, we have

$$(\lambda x.\ t)\ t' = t\{t'/x\}$$

**Example** Consider the term $(\lambda x.\ x\ x)\ y$. It is easy to check that $y$ is not captured in $(x\ x)$. Hence, we have $(\lambda x.\ x\ x)\ y =_\beta (x\ x)\{y/x\} = y\ y$.

Consider another term $(\lambda x.\ (\lambda y.\ x\ y))\ y$. Here, free variable $y$ is captured in the first term since $y$ is captured in $\lambda y.\ x\ y$ due to the $\lambda y$. Hence, $\alpha$-reduction is needed to remove the captured variable $y$. One way to do that is:

$$\begin{aligned}(\lambda x.\ (\lambda y.\ x\ y))\ y &=_\alpha (\lambda x.\ (\lambda z.\ x\ z))\ y \\ &=_\beta (\lambda z.\ x\ z)\{y/x\} \\ &= \lambda z.\ y\ z\end{aligned}$$

### 3.4 $\eta$-reduction

There is another notion of equality. Compare the terms $t$ and $\lambda x.\ t\ x$. If these two terms are both applied to an argument $t'$, they will both reduce to $t\ t'$, provided $x$ has no free occurrence in $t$. Formally, $(\lambda x.\ t_1\ x)\ t_2 =_\eta$

$(t_1\ t_2)$ if $x \notin \texttt{FV}(t_1)$. Therefore, $t$ and $\lambda x.\ t\ x$ behave the same way when treated as functions and should be considered equal. This gives rise to a reduction rule called $\eta$-reduction: $\lambda x.\ t\ x$ reduces to $t$ if $x \notin \texttt{FV}(t)$. The reverse operation, called $\eta$-expansion, has practical uses as well. In practice, $\eta$-expansion is used to delay divergence by trapping expressions inside $\lambda$-terms.

# 4 Reduction order

In the classical $\lambda$-calculus, no reduction strategy is specified, and no restrictions are placed on the order of reductions. Any redex (reduction expression) may be chosen to be reduced next. For example, $(\lambda x.\ (\lambda y.\ y)\ x)\ z$ can be reduced to either $(\lambda x.\ x)\ z$ (redex is $(\lambda y.\ y)\ x$), or $(\lambda y.\ y)\ z$ (redex is the entire term). A $\lambda$-term in general may have many redexes, so the process is nondeterministic. We can think of a reduction strategy as a mechanism for resolving the nondeterminism, but in the classical $\lambda$-calculus, no such strategy is specified.