1 The Lambda Calculus

1.1 Recap—λ-terms

Lambda calculus is a notation for describing mathematical functions and programs. A λ-term is:

1. a variable $x \in \mathbf{Var}$, where **Var** is a countably infinite set of variables;
2. a function e_0 applied to an argument e_1, usually written $e_0\ e_1$ or $e_0(e_1)$; or
3. an expression $\lambda x.\, e$ denoting a function with input parameter x and body e.

In BNF notation,

$$e \ ::= \ x \ \mid \ \lambda x.\, e \ \mid \ e_0\, e_1$$

Parentheses are used just for grouping; they have no meaning on their own. Lambdas are greedy, extending as far to the right as they can. 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 $\lambda xy.\, e$ as shorthand for $\lambda x.\, \lambda y.\, e$. This shorthand is an example of *syntactic sugar*. The process of removing it in this instance is called *currying*.

Where a mathematician might write $x \mapsto x^2$, in the λ-calculus we would write $\lambda x.x^2$. This suggests that functions are just ordinary values, and can be passed as arguments to functions (even to themselves!).

The λ-calculus is a mathematical system for studying the interaction of *functional abstraction* and *functional application*. In the *pure* λ-calculus, λ-terms serve as both functions and data.

1.2 Recap—BNF Notation

The grammar

$$e \ ::= \ x \ \mid \ \lambda x.\, e \ \mid \ e_0\ e_1$$

describing the syntax of the pure λ-calculus, the e is not a variable in the language, but a *metavariable* representing a syntactic class (in this case λ-terms) in the language. It is not a variable at the level of the programming language. We use subscripts to differentiate syntactic metavariables of the same syntactic class. For example, e_0, e_1 and e all represent λ-terms.

1.3 Recap—Variable Binding

Occurrences of variables in a λ-term can be *bound* or *free*. In the λ-term $\lambda x.\, e$, the lambda abstraction operator λx binds all the free occurrences of x in e. The *scope* of λx in $\lambda x.\, e$ is e. This is called *lexical scoping*; the variable's scope is defined by the text of the program. 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.

1.4 Digression—Terms and Types

There are different kinds of expressions in a typical programming language: *terms* and *types*. We have not talked about types yet, but we will soon. A term represents a value that exists only at run time; a type is a compile-time expression used by the compiler to rule out ill-formed programs. For now there are no types.

2 Substitution and β-reduction

Now we get to the question: How do we run a λ-calculus program? The main computational rule is called *β-reduction*. This rule applies whenever there is a subterm of the form $(\lambda x.\, e_1)\ e_2$ representing the application of a function $\lambda x.\, e_1$ to an argument e_2.

Intuitively, to perform the β-reduction, we substitute the argument e_2 for all free occurrences of the formal parameter x in the body e_1, then evaluate the resulting expression (which may involve further such steps).

We have to be a little careful though. We cannot just substitute e_2 blindly for x in e_1, because bad things could happen which could alter the semantics of expressions in undesirable ways. For example, if e_2 contained a free occurrence of a variable y, and there were a free occurrence of x in the scope of a λy in e_1, then the free occurrence of y in e_2 would be "captured" by that λy and would end up bound to it after the substitution. This would not be good. However, we can avoid the problem by renaming the bound variable y.

2.1 Safe Substitution

We wish to define a notion of *safe substitution* in which undesired capture of variables is avoided by judicious renaming of bound variables. We write $e_1\{e_2/x\}$ to denote the result of substituting e_2 for all free occurrences of x in e_1 according to the following rules:

$$
\begin{array}{rcll}
x\{e/x\} & = & e & \\
y\{e/x\} & = & y & \text{where } y \neq x \\
(e_1\, e_2)\{e/x\} & = & e_1\{e/x\} \cdot e_2\{e/x\} & \\
(\lambda x.\, e_0)\{e_1/x\} & = & \lambda x.\, e_0 & \\
(\lambda y.\, e_0)\{e_1/x\} & = & \lambda y.\, e_0\{e_1/x\} & \text{where } y \neq x \text{ and } y \notin FV(e_1) \\
(\lambda y.\, e_0)\{e_1/x\} & = & \lambda z.\, e_0\{z/y\}\{e_1/x\} & \text{where } y \neq x,\ z \neq x,\ z \notin FV(e_0), \text{ and } z \notin FV(e_1).
\end{array}
$$

(There are many notations for substitution. Pierce writes $[x \mapsto e_2]e_1$. Because we will be using similar notation for something else, we will use the notation $e_1\{e_2/x\}$.)

Note that the rules are applied inductively. That is, the result of a substitution in a compound term is defined in terms of substitutions on its subterms. The very last of the six rules applies when $y \in FV(e_1)$. In this case we can rename the bound variable y to z to avoid capture of the free occurence of y. One might well ask: But what if y occurs free in the scope of a λz in e_0? Wouldn't the z then be captured? The answer is that it will be taken care of in the same way, but inductively on a smaller term.

Rewriting $(\lambda x.\, e_1)\ e_2$ to $e_1\{e_2/x\}$ is the basic computational step of the λ-calculus and is called *β-reduction*. In the pure λ-calculus, we can start with a λ-term and perform β-reductions on subterms in any order.

3 Call-by-Name and Call-by-Value

Now we have another issue. In general there may be many possible β-reductions that can be performed on a given λ-term. How do we choose which beta reductions to perform next? Does it matter?

A specification of which β-reduction to perform next is called a *reduction strategy*. Let us define a *value* to be a closed λ-term to which no β-reductions are possible, given our chosen reduction strategy. For example, $\lambda x.x$ would always be value, whereas $(\lambda x.x)1$ would most likely not be.

Most real programming languages based on the λ-calculus use a reduction strategy known as *call By value* (CBV). In other words, they may only call functions on values. Thus $(\lambda x.e_1)e_2$ only reduces if e_2 is a value. Here is an example of a CBV evaluation sequence, assuming 3, 4, and s (the successor function) are appropriately defined.

$$((\lambda x.\lambda y.y\ x)\ 3)\ s \ \longrightarrow\ (\lambda y.y\ 3)\ s \ \longrightarrow\ s\ 3 \ \longrightarrow\ 4.$$

Another strategy is *call by name* (CBN). We defer evaluation of arguments until as late as possible, applying reductions from left to right within the expression.

4 Structured Operational Semantics (SOS)

Let's formalize CBV with a few inference rules.

$$\frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}} \quad [\beta\text{-reduction}]$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}$$

$$\frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}$$

This is a simple operational semantics for a programming language based on the lambda calculus. An operational semantics is a language semantics that describes how to run the program. This can be done through informal human-language text, as in the Java Language Specification, or through more formal rules. Rules of this form are known as a Structural Operational Semantics (SOS). They define evaluation as the result of applying the rules to transform the expression. The rules are typically defined in term of the structure of the expression being evaluated.

This kind of operational semantics is known as a *small-step* semantics because it only describes one step at a time. An alternative is a *big-step* (or *large-step*) semantics that describes the entire evaluation of the program to a final value.

We will see other kinds of semantics later in the course, such as *axiomatic semantics*, which describes the behavior of a program in terms of the observable properties of the input and output states, and *denotational semantics*, which translates a program into an underlying mathmatical representation.

Expressed as SOS, CBN has slightly simpler rules:

$$\frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1\{e_2/x\}} \quad [\beta\text{-reduction}]$$

$$\frac{e_0 \longrightarrow e_0'}{e_0\ e_1 \longrightarrow e_0'\ e_1}$$

We don't need the rule for evaluating the right-hand side of an application because β-reductions are done immediately once the left-hand side is a value.

4.1 Ω

Let us define an expression we will call Ω:

$$\Omega \quad = \quad (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

What happens when we try to evaluate it?

$$\Omega \quad = \quad (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \quad \longrightarrow \quad (x\ x)\{(\lambda x.\, x\ x)/x\} \quad = \quad \Omega$$

We have just coded an infinite loop!

Now what happens if we try using Ω as a parameter?

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

Using the CBV evaluation strategy, we must first reduce Ω. This puts the evaluator into an infinite loop. On the other hand, CBN reduces the term above to $\lambda y.\, y$. CBN has an important property: CBN will not loop infinitely unless every other semantics would also loop infinitely, yet it agrees with CBV whenever CBV terminates successfully.