The Lambda Calculus

Bharat Jayaraman
Department of Computer Science and Engineering
University at Buffalo (SUNY)
January 2010

The *lambda-calculus* is of fundamental importance in the study of programming languages. It was founded by Alonzo Church in the 1930s well before programming languages or even electronic computers were invented. Church's thesis is that the computable functions are precisely those that are definable in the lambda-calculus. This is quite amazing, since the entire syntax of the lambda-calculus can be defined in just one line:

$$T \ ::= \ V \ \mid \ \lambda V.T \ \mid \ (T \ \ T)$$

The syntactic category T defines the set of well-formed expressions of the lambda-calculus, also referred to as *lambda-terms*. The syntactic category V stands for a *variable*; $\lambda V.T$ is called an *abstraction*, and $(T \ \ T)$ is called an *application*. We will examine these terms in more detail a little later, but, at the outset, it is remarkable that the lambda-calculus is capable of defining recursive functions even though there is no provision for writing recursive definitions. Indeed, there is no provision for writing *named* procedures or functions as we know them in conventional languages. Also absent are other familiar programming constructs such as control and data structures. We will therefore examine how the lambda-calculus can encode such constructs, and thereby gain some insight into the power of this little language.

In the field of programming languages, John McCarthy was probably the first to recognize the importance of the lambda-calculus, which inspired him in the 1950s to develop the Lisp programming language. Lisp may be regarded as the synthesis of two key ideas: the use of lambda-terms for representing functions, and the use of a built-in list abstract datatype for constructing structured data objects. In the 1960s, Peter Landin showed how the statically scoped procedural language Algol 60 could be translated into the lambda-calculus, hence showing that the semantics of imperative languages could be reduced to the semantics of the lambda-calculus. In the 1970s, the lambda-calculus received increased attention, soon after Dana Scott showed how to give mathematical semantics (or "models") for the lambda-calculus. This result in fact provided a mathematical basis for all programming languages—earning Scott the prestigious ACM Turing Award in 1976—but it especially motivated interest in *functional programming languages* with *higher-order* functions and *lazy evaluation*. These topics continue to be of great interest today.

The language of lambda-terms referred to above is also called the *untyped lambda-calculus*, meaning that no type information is associated with variables. There are two interesting variants of the untyped lambda-calculus: the *simply typed* lambda-calculus, and the *second-order typed* lambda-calculus. (One encounters the latter in the study of ML, a higher-order functional programming language.) It suffices for now to note that the difference between them is that second-order types can contain *type variables* but simple types cannot. Second-order types are also referred to as "polymorphic" types or "generic" types. Both these calculi require each variable to have a type, and application terms must respect this type information. As an interesting consequence, both calculi are strictly less powerful than the untyped lambda-calculus, and they also have different expressive powers. The type requirements effectively prevent a simply typed or second-order typed lambda-term from representing a nonterminating program.

Typed lambda-calculi are currently a very active topic of research. There are close connections between the structure of proofs in certain systems of logic and that of typed lambda-terms, and this idea has prompted research in extracting (typed functional) programs from proofs (of certain logical statements). Recently, an interesting language called λProlog—essentially Prolog with typed lambda-terms replacing first-order terms—was shown to have potential use in such diverse areas as proof systems, natural language analysis, and program transformation and synthesis. Thus typed lambda-calculi appear to be a unifying force between theories of functions and logics.

We provide an introduction to the untyped and typed lambda-calculi. Section 1 covers syntactic issues, including *bound* and *free* occurrences of variables, as well as *substitutions.* Section 2 presents the *reduction rules* of the lambda-calculus, and shows how computation is modeled in terms of reduction. This section also states the well-known *Confluence* and *Church-Rosser* properties of the lambda-calculus. Section 3 shows how to encode various programming language constructs in the lambda-calculus, especially data types, control structures, and recursive definitions. Section 4 introduces the typed lambda-calculi, and briefly covers simple and second-order types.

A good introductory treatment of the lambda-calculus can be found in the book "Programming Language Theory and its Implementation," by M.J.C. Gordon, published by Prentice-Hall. A more advanced treatment, covering the theory of fixed-points and models of the lambda-calculus, is in the book "Denotational Semantics," by J. Stoy, published by MIT Press. "Lambda Calculus - its Syntax and Semantics," by H.P. Barendregt, published by North-Holland, is an advanced, encyclopaedic treatment of the subject. The simple and second-order typed lambda-calculi are discussed in the paper by S. Fortune, D. Leivant, and M.J. O'Donnell, "The Expressiveness of Simple and Second-Order Type Structures," *Journal of the ACM*, vol. 30, no. 1, pp. 151-185, 1983. A more introductory treatment of this topic is in the ACM Computing Surveys article, "On Understanding Types, Data Abstraction, and Polymorphism," by L. Cardelli and P. Wegner, vol. 17, no. 4, 1985.

1. Syntactic Issues

As noted earlier, the set of well-formed lambda-terms is given by the grammar:

$$T \ ::= \ V \ \mid \ \lambda V.T \ \mid \ (T \ \ T)$$

where V stands for a *variable* term, $\lambda V.T$ stands for an *abstraction* term, and $(T\ \ T)$ stands for an *application* term. The variable V in $\lambda V.T$ will be referred to as a binder variable. Our convention in these notes will be to write *variables* using lowercase letters, possibly subscripted, e.g. $x,\ y,\ z,\ x_1,\ x_2,\ y_1,\ldots$. We will use $T,\ T_1,\ T_2,\ldots$ to stand for an arbitrary lambda-term, and $V,\ V_1,\ V_2,\ldots$ to stand for an arbitrary variable.

Abstraction terms (or *abstractions*) are of the form $\lambda V.T$ where T is any term. Examples are:

$\lambda x.y$

$\lambda x.x$

$\lambda x.(x\ x)$

$\lambda x.(x\ y)$

$\lambda x.\lambda y.x$

$\lambda x.\lambda y.y$

$\lambda f.\lambda x.(f\ x)$

$\lambda f.\lambda x.(f\ (f\ x))$

$\ldots$

Application terms (or *applications*) are of the form $(T_1\ T_2)$ where T_1 and T_2 are any terms. Examples are:

$(x\ y)$

$((x\ y)\ z)$

$(x\ (y\ z))$

$(\lambda x.x\ y)$

$((\lambda f.\lambda y.(f\ (f\ y))\ \ \lambda z.z)\ \ w)$

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

$\ldots$

Thus, abstractions and applications can be arbitrarily nested inside one another.

Some intuition underlying this unusual syntax is probably necessary at this point: Since the lambda-calculus was designed to be a theory of functions, lambda-terms must somehow encode the essential features of functions. Towards this end, *abstractions* serve as way of encoding (or writing) function definitions†, and *applications* serve to encode the application

† The difference between a definition of a function and the function *per se* should be noted. For example, there are many different ways—indeed, infinitely many ways—of defining, say, the factorial function, but the factorial function itself is specified by the set of ordered pairs $\{0 \mapsto 1,\ 1 \mapsto 1,\ 2 \mapsto 2,\ 3 \mapsto 6, \ldots\}$.

of a function to its arguments. More precisely, an abstraction $\lambda V.T$ defines a function of one argument V and whose result is defined by T. In programming language terminology, $\lambda V.T$ can be thought of as a (nameless) function with one formal parameter V and whose result expression is given by T; and $(F\ T)$ can be thought of as a call to function F with actual parameter T. Thus, for example,

$$\lambda x.x$$

is an encoding of the identity function. The abstraction

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

encodes a function that takes two arguments, and returns the first. The abstraction

$$\lambda f.\lambda x.(f\ x)$$

encodes a function of two arguments, f and x, whose result is that of applying f to x. These examples illustrate two points:

1. A multi-argument function can be encoded in the lambda-calculus by "cascading" a series of single-argument functions. For example, $\lambda x.\lambda y.T$ actually encodes a function that takes one argument x and *returns a function* that takes the second argument y and has body T. It follows that the application of a multi-argument function to actual parameters must be accomplished in the lambda-calculus through a series of single-argument applications.

2. The lambda-calculus is a *higher-order* language. For example, $\lambda f.\lambda x.(f\ x)$ represents a function that takes another function f as argument and returns the function $\lambda x.(f\ x)$ as its result.

1.1 Bound and Free Occurrences of Variables

An occurrence of a variable V in a lambda-term is said to be *bound* if it lies within the lexical scope of a surrounding abstraction of the form $\lambda V.T$; otherwise the variable occurrence is *free*. The notion of a *free* occurrence of a variable is similar to that of a *non-local* occurrence of a variable in a lexically-scoped programming language. For example, in the following term,

$$\lambda x.(\underline{x}\ \underline{y})$$

in which variable occurrences are underlined, the occurrence of y is free, but the occurrence of x is bound by the surrounding abstraction. In the term

$$(\lambda x.\underline{y}\ \lambda y.\underline{y})$$

the first occurrence of y is free, but the second is bound. In the term

$$\lambda x.(\lambda y.\underline{x}\ \ \lambda x.(\underline{y}\ \underline{x}))$$

the two occurrences of x are bound by two different abstractions, but the occurrence of y is free. Note that the occurrence of y above is not within the scope of the term $\lambda y.\underline{x}$.

A more precise specification of a free occurrence of a variable is provided by the following infix binary relation *occurs_free_in*, which is a case analysis of the three syntactic categories of lambda-terms (note that V and V_1 range over variables in the lambda-calculus, whereas T, T_1 and T_2 range over arbitrary lambda-terms):

$$V \ \textit{occurs\_free\_in}\ V_1 \quad \leftrightarrow \quad V = V_1$$

$$V \ \textit{occurs\_free\_in}\ \lambda V_1\ .\ T \quad \leftrightarrow \quad V \neq V_1 \ \wedge\ (V \ \textit{occurs\_free\_in}\ T)$$

$$V \ \textit{occurs\_free\_in}\ (T_1\ T_2) \quad \leftrightarrow \quad (V \ \textit{occurs\_free\_in}\ T_1)\ \vee\ (V \ \textit{occurs\_free\_in}\ T_2)$$

Exercise 1: Write a function, free(T), that returns the set of all free occurrences of variables in a lambda-term T.

1.2 Substitution

In preparation for the next section, we explain how to substitute some term T_1 for all *free* occurrences of a variable V in a term T_2. We write this substitution operation as follows†:

$$T_2[V \leftarrow T_1].$$

For example, the result of substituting $\lambda w.w$ for all free occurrences of f in $\lambda x.(f\ (f\ \lambda f.(f\ x)))$ is $\lambda x.(\lambda w.w\ (\lambda w.w\ \lambda f.(f\ x)))$, i.e.,

$$\lambda x.(f\ (f\ \lambda f.(f\ x)))\ [f \leftarrow \lambda w.w] \ = \ \lambda x.(\lambda w.w\ (\lambda w.w\ \lambda f.(f\ x))).$$

Note that the occurrence of f inside $\lambda f.(f\ x)$ is bound, and hence it is not replaced. From the above example, it appears that the substitution could be achieved by a straightforward textual replacement. Therefore we might try to define the substitution operation recursively as follows (assuming, as before, that V and V_1 range over variables in the lambda-calculus, and T, T_1 and T_2 stand for arbitrary lambda-terms):

$$V\ \ [V \leftarrow T] \ = \ T$$

$$V_1\ \ [V \leftarrow T] \ = \ V_1, \ \ \textbf{if}\ \ V \neq V_1$$

$$(T_1\ \ T_2)\ \ [V \leftarrow T] \ = \ (T_1[V \leftarrow T]\ \ \ T_2[V \leftarrow T])$$

$$\lambda V.T_1\ \ [V \leftarrow T] \ = \ \lambda V.T_1$$

† Another way of writing the substitution operation is $[T_1/V]T_2$.

$$\lambda V_1.T_1 \;\; [V \leftarrow T] \; = \; \lambda V_1.T_1[V \leftarrow T], \;\; \textbf{if} \;\; V \neq V_1$$

The above definition of substitution is, however, deficient in one crucial respect. To see the problem, let us reconsider the earlier example, but now suppose we wish to substitute the term $\lambda w.(w\ x)$ for all free occurrences of f in $\lambda x.(f\ (f\ \lambda f.(f\ x)))$. According to the above recursive specification of substitution, our result will be:

$$\lambda x.(\lambda w.(w\ x)\ \ (\lambda w.(w\ x)\ \ \lambda f.(f\ x))).$$

Note that, in this result, the *free* occurrence x in $\lambda w.(w\ x)$ has become *bound* to the surrounding abstraction $\lambda x. \ldots$ in the resulting term. This problem is referred to as the *variable capture* problem. Precisely why this outcome is unsatisfactory will become clearer when we examine the reduction rules in the next section. For now, we simply prevent such free variables from becoming bound during substitution by replacing the last case of the recursive specification of substitution by the following two cases:

$$\begin{aligned}
\lambda V_1.T_1 \;\; [V \leftarrow T] &= \lambda V_1.T_1[V \leftarrow T], && \textbf{if} \;\; V \neq V_1 \;\wedge\; \neg(V_1 \; \mathit{occurs\_free\_in} \; T) \\
\lambda V_1.T_1 \;\; [V \leftarrow T] &= \lambda V_2.T_1[V_1 \leftarrow V_2][V \leftarrow T], && \textbf{if} \;\; V \neq V_1 \;\wedge\; (V_1 \; \mathit{occurs\_free\_in} \; T) \\
&&& \wedge\; \neg(V_2 \; \mathit{occurs\_free\_in} \; T) \\
&&& \wedge\; \neg(V_2 \; \mathit{occurs\_free\_in} \; T_1)
\end{aligned}$$

The first case above simply says that we may substitute, as before, T for V in $\lambda V_1.T_1$ as along as the V_1 does not occur free in T. If V_1 does occur free in T, the second case specifies that we first construct the term $\lambda V_2.T_1[V_1 \leftarrow V_2]$ by renaming the binder variable V_1 by a variable V_2—which does not occur free in T or T_1—and replacing all free occurrences of V_1 in T_1 by V_2, and we then replace T for V in the constructed term. For example, the result of substituting the term $\lambda w.(w\ x)$ for all free occurrences of f in $\lambda x.(f\ (f\ \lambda f.(f\ x)))$ is obtained by first renaming the binder variable x in the outer abstraction by a new variable. This new variable should not be free in $\lambda w.(w\ x)$ or $(f\ (f\ \lambda f.(f\ x)))$. That is, it can be any variable other than f or x. Suppose we choose y. We first construct the term $\lambda y.(f\ (f\ \lambda f.(f\ y)))$, and substitute $\lambda w.(w\ x)$ for all free occurrences of f in this term. Our result therefore is:

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

If, on the other hand, we wished to substitute $\lambda w.(w\ z)$ for all free occurrences of f in $\lambda x.(f\ (f\ \lambda f.(f\ x)))$, no renaming is necessary to obtain the correct answer:

$$\lambda x.(\lambda w.(w\ z)\ \ (\lambda w.(w\ z)\ \ \lambda f.(f\ x))).$$

Finally, consider the following substitution operation:

$$\lambda y.(f\ (y\ \lambda y.(y\ f)))[f \leftarrow \lambda w.(w\ y)]$$

Following the last case of the definition of the substitution operation, we must first rename the outer binder variable y to (say) y_1. Hence the above substitution operation is equivalent to:

$$\lambda y_1.(f\ \ (y_1\ \lambda y.(y\ f)))[f \leftarrow \lambda w.(w\ y)].$$

In applying the substitution recursively to the subterms, the binder variable y gets renamed to y_2, and hence the result is:

$$\lambda y_1.(\lambda w.(w\ y)\ \ (y_1\ \lambda y_2.(y_2\ \lambda w.(w\ y)))).$$

Exercise 2: Following the definition of the substitution operation, determine the result of the following expression:

(i) $\lambda x.x\ [x \leftarrow a]$.

(ii) $\lambda y.(x\ \ \lambda z.(z\ \ x))[x \leftarrow z]$.

(iii) $\lambda y.(x\ \ \lambda z.(z\ \ (x\ \ y)))\ [x \leftarrow \lambda z.(z\ \ (x\ \ y))]$.

Exercise 3: Prove that $T[x \leftarrow T_1][y \leftarrow T_2] = T[y \leftarrow T_2][x \leftarrow T_1[y \leftarrow T_2]]$, assuming that $x \neq y$ and $\neg(x\ \mathit{occurs\_free\_in}\ T_2)$.

2. Reduction and Equality of Lambda Terms

2.1 Reduction

The following are the three famous *reduction rules* of the lambda-calculus:

1. α-reduction: $\lambda x.T \longrightarrow \lambda y.T[x \leftarrow y]$, **if** $\neg(y\ \mathit{occurs\_free\_in}\ T)$.
2. β-reduction: $(\lambda x.T_1\ \ T_2) \longrightarrow T_1[x \leftarrow T_2]$.
3. η-reduction: $\lambda x.(T\ x) \longrightarrow T$, **if** $\neg(x\ \mathit{occurs\_free\_in}\ T)$.

Computation is modeled in the lambda-calculus by the notion of a *reduction sequence*. In preparation for the definition of a reduction sequence, we first define a *reduction step* as:

$$T_1 \Longrightarrow T_2$$

if T_1 and T_2 are obtained as a direct instance of one of the three reduction rules or T_2 is the result of replacing some subterm of T_1 according to one of the reduction rules—the latter case is referred to as *subterm-replacement*.

Examples of α-reduction steps:

$$\lambda x.x \implies \lambda y.y$$

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

$$\lambda f.\lambda x.(f\ (f\ x)) \implies \lambda f.\lambda y.(f\ (f\ y)) \qquad \text{(subterm-replacement)}$$

The α-reduction rule basically states that systematic renaming of a binder variable in an abstraction produces an equivalent lambda-term. Note that $\lambda x.((x\ (f\ y))\ x) \not\Longrightarrow \lambda y.((y\ (f\ y))\ y))$, because y occurs free in the term on the left. That is, renaming is permissible only if this does not cause a free variable to be bound (or "captured") by the new binder variable.

Examples of β-reduction steps:

$$(\lambda x.x\ a) \implies a$$

$$(\lambda f.\lambda x.(f\ (f\ x))\ \ \lambda x.x) \implies \lambda x.(\lambda x.x\ (\lambda x.x\ x))$$

$$(\lambda f.\lambda x.(f\ (f\ x))\ \ \lambda y.(x\ y)) \implies \lambda x_2.(\lambda y.(x\ y)\ (\lambda y.(x\ y)\ x_2))$$

$$\lambda y.(\lambda x.(f\ (f\ x))\ \ a) \implies \lambda y.(f\ (f\ a)) \qquad \text{(subterm-replacement)}$$

The β-reduction rule models function application. It basically states that function application involves the replacement of all occurrences of the formal parameter in the body of the function definition by the actual parameter. This replacement is given by the substitution operation defined in the previous subsection. The parameter transmission mode can be seen to be similar to call-by-name.

Examples of η-reduction steps:

$$\lambda x.(f\ x) \implies f$$

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

$$(\lambda y.\lambda x.(f\ x)\ z) \implies (\lambda y.f\ z) \qquad \text{(subterm-replacement)}$$

The η-reduction rule is related to the rule of *extensionality*, which states that two functions are equivalent if they return the same results for all arguments (see exercise 8). For example, suppose that the lambda-terms $\lambda x.(f\ x)$ and f are applied to some arbitrary term T. These applications are represented by $(\lambda x.(f\ x)\ \ T)$ and $(f\ \ T)$. Since the latter is the result of β-reducing the former, the terms $\lambda x.(f\ x)$ and f are taken to be equivalent. In other words, f is a more compact (or reduced) form of $\lambda x.(f\ x)$. Note that $\lambda x.((f\ x)\ x) \not\Longrightarrow (f\ x)$ by η-reduction since the subterm $(f\ x)$ of $\lambda x.((f\ x)\ x)$ contains an occurrence of x free in it.

In order to define a reduction sequence, we define

$T_1 \Longrightarrow^* T_2$

as the reflexive, transitive closure of $\Longrightarrow$, and take a *reduction sequence* to be all terms that appear in a derivation of $\Longrightarrow^*$. Examples:

$x \Longrightarrow^* x$	(reflexive rule)
$\lambda x.(f\ x) \Longrightarrow^* \lambda x.(f\ x)$	(reflexive rule)
$\lambda x.\lambda y.(x\ y) \Longrightarrow^* \lambda p.\lambda q.(p\ q)$	(transitive rule, two α-reduction steps)
$((\lambda x.\lambda y.(x\ y)\ a)\ b) \Longrightarrow^* (a\ b)$	(transitive rule, two β-reduction steps)
$\lambda x.\lambda y.((f\ x)\ y) \Longrightarrow^* f$	(transitive rule, two η-reduction steps)

Exercise 4: Show the sequence of reduction steps to prove the following:

$$(\lambda t.(\lambda w.(w\ t)\ \lambda p.p)\ p) \Longrightarrow^* p$$

Exercise 5: In view of the reduction rules, explain why the substitution operation was defined so as to avoid the "variable capture" problem.

2.2 The Confluence Property

We will say that a lambda-term T is *reducible* if it is (or contains a subterm) of the form $(\lambda x.T_1\ \ T_2)$ or of the form $\lambda x.(T_3\ \ x)$, with x not occurring free in T_3; otherwise we will say that T is *irreducible.* A subterm of the form $(\lambda x.T_1\ \ T_2)$ or $\lambda x.(T_3\ \ x)$, with x not occurring free in T_3, is called a *redex.* In general, a term may contain more than one redex, and this is the reason that we might have different reduction sequences starting from some lambda-term. An irreducible term is said to be in *normal form*, or *$\beta\eta$-normal form.* If $T_1 \Longrightarrow^* T_2$ and T_2 is in normal form, we will say that T_2 *is a normal form of* T_1. If a reduction sequence is thought of as representing a computation, then the computation "terminates" when a normal-form term appears at the end of the sequence.

Two questions arise in connection with normal forms:

Does every lambda-term have a normal form?

Is the normal form for a term unique (assuming it exists)?

The first question is equivalent to asking: Does the computation arising from every lambda-term terminate? In view of our opening remark that the lambda-calculus can represent any computable function, one should expect that the answer to this question is 'no'. As a concrete example, below is a nonterminating reduction sequence:

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

$\Longrightarrow (\lambda x.(x \;\; x) \;\; \lambda x.(x \;\; x))$

$\Longrightarrow (\lambda x.(x \;\; x) \;\; \lambda x.(x \;\; x))$

$\Longrightarrow \ldots$

(An application such as $(x \; x)$ is called *self application*. It is because of such terms that nontermination is possible in the lambda-calculus. In typed lambda-calculi, such terms are prohibited because it is not possible to assign a type to x—the type of a function is necessarily different from the types of its arguments.)

To get a feel for the question on the uniqueness of normal forms, let us examine the following reduction sequence.

Reduction Sequence 1:

$(\lambda x.(\lambda f.(f \;\; x) \;\; \lambda z.z) \;\; \lambda y.(f \;\; y))$

$\Longrightarrow (\lambda f_2.(f_2 \;\; \lambda y.(f \;\; y)) \;\; \lambda z.z)$ (β-reduction)

$\Longrightarrow (\lambda z.z \;\; \lambda y.(f \;\; y))$ (β-reduction)

$\Longrightarrow \lambda y.(f \;\; y)$ (β-reduction)

$\Longrightarrow f$ (η-reduction)

Firstly, it is clear that the η-reduction which was performed in the last step could have been performed at any earlier step without altering the final outcome. Assuming that we leave the η-reduction to the last step, there still are two other possible reduction sequences, as shown below. At each step, the underlined subterm is the redex that is reduced:

Reduction Sequence 2:

$(\lambda x.\underline{(\lambda f.(f \;\; x) \;\; \lambda z.z)} \;\; \lambda y.(f \;\; y))$

$\Longrightarrow (\lambda x.\underline{(\lambda z.z \;\; x)} \;\; \lambda y.(f \;\; y))$ (β-reduction)

$\Longrightarrow \underline{(\lambda x.x \;\; \lambda y.(f \;\; y))}$ (β-reduction)

$\Longrightarrow \underline{\lambda y.(f \;\; y)}$ (β-reduction)

$\Longrightarrow f$ (η-reduction)

Reduction Sequence 3:

$(\lambda x.\underline{(\lambda f.(f \;\; x) \;\; \lambda z.z)} \;\; \lambda y.(f \;\; y))$

$\Longrightarrow \underline{(\lambda x.(\lambda z.z \;\; x) \;\; \lambda y.(f \;\; y))}$ (β-reduction)

$\Longrightarrow \underline{(\lambda z.z \;\; \lambda y.(f \;\; y))}$ (β-reduction)

$\Longrightarrow \underline{\lambda y.(f \;\; y)}$ (β-reduction)

$\Longrightarrow f$ (η-reduction)

Although the three reduction sequences from the term $(\lambda x.(\lambda f.(f \;\; x) \;\; \lambda z.z) \;\; \lambda y.(f \;\; y))$ do not make the same choice for the redex to be reduced at each step, the normal-form of this term is the same. But consider the reduction sequence from $(\lambda x.\lambda f.(f \;\; x) \;\; (f \;\; y))$. There are infinitely many normal forms for this term:

$\lambda f_1.(f_1 \;\; (f \;\; y))$,

$\lambda f_2.(f_2 \;\; (f \;\; y))$,

$\lambda f_3.(f_3 \;\; (f \;\; y))$,

...

However, these terms are all α-reducible to one another, and hence they may be regarded as essentially identical. The lambda-calculus has the property that the normal form for every term is unique up to α-reduction, if the normal form exists. We are now ready to state the first major theorem of the lambda-calculus:

Theorem 1 (*Confluence Property*): If $T \Longrightarrow^* T_1$ and $T \Longrightarrow^* T_2$, then there is a term U such that $T_1 \Longrightarrow^* U$ and $T_2 \Longrightarrow^* U$.

The *Confluence Property* states that whenever a term can be reduced to two different terms, these two terms can be reduced to a common term. Using this property, it is easy prove that the normal form of a term is unique (modulo α-reduction) if it exists: Suppose that a term T has two normal forms T_1 and T_2, i.e., $T \Longrightarrow^* T_1$ and $T \Longrightarrow^* T_2$. By the Confluence Property, there is a term U such that $T_1 \Longrightarrow^* U$ and $T_2 \Longrightarrow^* U$. Since T_1 and T_2 are normal forms, it must be the case that, if T_1 and T_2 are not identical, they could be related to U only by α-reduction. Hence T_1 and T_2 are identical modulo α-reduction.

Given that nonterminating reduction sequences are possible, how should we carry out reductions so that a normal-form term is produced whenever one exists? To appreciate this question, consider the term:

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

Since the subterm $(\lambda x.(x \;\; x) \;\; \lambda x.(x \;\; x))$ can be reduced *ad infinitum*, there is a nonterminating reduction sequence from the given term. However, there is also a reduction sequence leading to the normal form, as follows:

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

$$\Longrightarrow (\lambda y.a \;\; (\lambda x.(x \;\; x) \;\; \lambda x.(x \;\; x))$$

$$\Longrightarrow a$$

In general, a normal form can be produced if the *leftmost redex* is chosen at each reduction step. The leftmost redex of a lambda-term, assuming it has a redex, can be defined recursively. The main cases are as follows (the full definition is left as an exercise):

i. if the term is a variable V, it has no redex;

ii. if the term is of the form $\lambda V.(T \; V)$ and V does not occur free in T, then it is $\lambda V.(T \; V)$; for all other abstractions of the form $\lambda V.T$, it is the leftmost redex of T.

iii. if the term is of the form $(\lambda V.T \; U)$, it is $(\lambda V.T \; U)$; otherwise, assuming the term is of the form $(T_1 \; T_2)$, it is the leftmost redex of T_1 if there is a redex in T_1, otherwise it is the leftmost redex of T_2.

Exercise 6: Define a function lm(T) that returns the leftmost redex of a term T if one exists; otherwise returns a special symbol, say **none**. Write your definition similar to that of substitution operation.

While the leftmost reduction strategy avoids nontermination, it does not guarantee that the reduction sequence will be the shortest possible one. For example, consider the following two reduction sequences:

Leftmost strategy:

$$(\lambda x.(x \; (x \; a)) \;\; \lambda x.(f \; x))$$

$$\Longrightarrow (\lambda x.(f \; x) \;\; (\lambda x.(f \; x) \;\; a))$$

$$\Longrightarrow (f \; (\lambda x.(f \; x) \;\; a))$$

$$\Longrightarrow (f \; (f \; a))$$

Another strategy:

$$(\lambda x.(x \; (x \; a)) \;\; \lambda x.(f \; x))$$

$$\Longrightarrow (\lambda x.(x \; (x \; a)) \;\; f)$$

$$\Longrightarrow (f \; (f \; a))$$

The leftmost strategy ended up taking more steps. The reason is that β-reduction of a term such as $(\lambda x.T_1 \;\; T_2)$ involves substituting T_2 for all occurrences of x in T_1. Thus, after substitution, the term T_2 may be reduced multiple times within the body of T_1. This situation is similar to name parameters *vs* value parameters in conventional languages†. While the use of a name parameter avoids simplification of the corresponding actual parameter expression if it is not needed in a particular call, its use might result in the argument expression being simplified more than once. For lambda-terms, although there are techniques, such as *graph reduction*, which avoid re-simplifying the argument term, the problem of devising an *optimal reduction strategy*, *i.e.*, one that finds the shortest reduction sequence for an arbitrary lambda-term, still remains an open problem in the field.

2.3 The Church-Rosser Property

The reduction rules implicitly define certain lambda-terms to be equal to one another. While the α- and η-reduction rules serve as a basis for defining which function definitions are equal (because these rules are defined for *abstractions*), the β-reduction rule (which is defined for *applications*) serves to define which function applications are equal to other terms. We make this connection more explicit as follows. We first define

$$T_1 \Longleftrightarrow T_2$$

if $T_1 \Longrightarrow T_2$ or $T_2 \Longrightarrow T_1$. If $T_1 \Longleftrightarrow T_2$, we say that they are related to one another by a *conversion* step. We then take the desired equality between lambda-terms to be

$$T_1 \Longleftrightarrow^* T_2$$

which is the reflexive, symmetric, and transitive closure of $\Longleftrightarrow$. Note that the symmetry rule is used in the definition of $\Longleftrightarrow^*$, and this is the main difference between equality and reduction. Below are examples of terms that related by zero or more conversion steps, i.e., zero or more applications of the $\Longleftrightarrow$ relation:

$x \Longleftrightarrow^* x$	(reflexive rule)
$\lambda x.(f\ x) \Longleftrightarrow^* \lambda x.(f\ x)$	(reflexive rule)
$(f\ \lambda x.x) \Longleftrightarrow (f\ \lambda y.y)$	(α-conversion step)
$(f\ a) \Longleftrightarrow (f\ (\lambda x.x\ a))$	(β-conversion step)
$f \Longleftrightarrow \lambda x.(f\ x)$	(η-conversion step)
$(a\ b) \Longleftrightarrow^* ((\lambda x.\lambda y.(x\ y)\ a)\ b)$	(transitive rule, two β-conversion steps)
$f \Longleftrightarrow^* \lambda x.\lambda y.((f\ x)\ y)$	(transitive rule, two η-conversion steps)

Having introduced the notion of equality between lambda-terms, we can state the second major theorem of the lambda-calculus:

† We can now see that reduction order and parameter transmision in the lambda-calculus differs from that in Lisp. Instead of using leftmost(-outermost) reduction, Lisp adopts innermost reduction as the default parameter transmission mode. And, instead of using call-by-name, Lisp uses call-by-value

Theorem 2 (*Church-Rosser Property*): If $T_1 \Longleftrightarrow^* T_2$ then there is a term U such that $T_1 \Longrightarrow^* U$ and $T_2 \Longrightarrow^* U$.

The *Church-Rosser property* relates the notion of equality of lambda-terms ($\Longleftrightarrow^*$) with the notion of reduction ($\Longrightarrow^*$). It states that any two terms that are equal can be reduced to a common term. Note $T_1 \Longleftrightarrow^* T_2$ does *not* imply $T_1 \Longrightarrow^* T_2$ or $T_2 \Longrightarrow^* T_1$: For example, we might have $T_1 \Longrightarrow U$ and $T_2 \Longrightarrow U$, from which it follows that $T_1 \Longleftrightarrow^* T_2$, but it does not follow that $T_1 \Longrightarrow^* T_2$ or $T_2 \Longrightarrow^* T_1$. Although it is relatively easy to show that the Confluence and Church-Rosser properties are equivalent to one another, proving either of them independently of the other is quite hard†.

Exercise 7a: Prove that the *Church-Rosser property* implies the *Confluence property*.

Exercise 7b: Prove that the *Confluence property* implies the *Church-Rosser property*. *Hint: Use induction on the length of a $\Longleftrightarrow^*$ derivation.*

Exercise 8: The *extensionality* rule can be formally stated as follows: $(T_1 \quad x) = (T_2 \quad x) \rightarrow T_1 = T_2$, for any x. Prove that this rule equivalent to the rule of η equality, i.e., $\lambda x.(T \quad x) = T$ if x does not occur free in T. Note: Treat $=$ as $\Longleftrightarrow^*$.

3. Representation of Programming Constructs

3.1 Data Types

We begin by noting that a type is characterized by its operations. That is, it does not matter how we represent data values as long as the semantics of the operations of the type are preserved. In representing various data types using lambda-terms, we will justify the chosen representations by showing that the semantics of the operations of these types are preserved. Still, we need a way to represent data values, and the first idea is that a finite set of n values can be represented by n abstraction terms, each with n binders such that the i^{th} value is represented by returning the i^{th} binder. For example, a set of three values may be represented as follows:

$\lambda v_1.\lambda v_2.\lambda v_3.v_1$

$\lambda v_1.\lambda v_2.\lambda v_3.v_2$

$\lambda v_1.\lambda v_2.\lambda v_3.v_3$

The *boolean* type may thus be represented as:

† Because of this equivalence, some authors refer to the Confluence property as the Church-Rosser property. But it is important to note that they are two distinct properties.

let $\textbf{true} = \lambda x.\lambda y.x$

let $\textbf{false} = \lambda x.\lambda y.y$

The identifiers **true** and **false** are mere abbreviations for the respective lambda-terms to which they are equated. These abbreviations are written in boldface so as to distinguish them from lambda-terms. Henceforth we will use these abbreviations with the understanding that the reader substitutes the terms to which they are equated. The simplest operation on a boolean is negation, which may be encoded as:

let $\textbf{not} = \lambda b.((b\ \textbf{false})\ \textbf{true})$

The correctness of this definition may be verified by proving the following equalities which express the properties of the negation operation:

$\textbf{(not true)} \Longrightarrow^* \textbf{false}$

$\textbf{(not false)} \Longrightarrow^* \textbf{true}$

We illustrate the proof of the first equality:

$$\begin{aligned}
& \textbf{(not true)} \\
= \ & (\lambda b.((b\ \textbf{false})\ \textbf{true})\ \ \textbf{true}) \\
= \ & (\lambda b.((b\ \textbf{false})\ \textbf{true})\ \ \lambda x.\lambda y.x) \\
\Longrightarrow \ & ((\lambda x.\lambda y.x\ \textbf{false})\ \textbf{true}) \\
\Longrightarrow \ & (\lambda y.\textbf{false}\ \textbf{true}) \\
\Longrightarrow \ & \textbf{false}
\end{aligned}$$

Another basic operation on booleans is the conditional **if**, defined as:

$\textbf{if(true}, T_1, T_2\textbf{)} = T_1$

$\textbf{if(false}, T_1, T_2\textbf{)} = T_2$

The representation of **if** in the lambda-calculus is as follows:

let $\textbf{if} = \lambda b.\lambda x.\lambda y.((b\ \ x)\ \ y)$

The correctness of this definition can be easily verified since $(((\textbf{if true})\ T_1)\ T_2) \Longrightarrow^* T_1$ and $(((\textbf{if false})\ T_1)\ T_2) \Longrightarrow^* T_2$. For the sake of readability, henceforth we will write lambda-terms so that the application of a function abbreviation such as **if** to its arguments is written as $(\textbf{if}\ B\ T_1\ T_2)$ rather than as $(((\textbf{if}\ B)\ T_1)\ T_2)$.

Exercise 9: Devise lambda-term representations for the boolean operations **and** and **or**.

Now let us turn our attention to representing a type with infinitely many values—the *natural number* (or non-negative number). From the study of data types, we know that these values can be represented in terms of two constructors, *zero* and *succ* (for successor), as follows:

$$zero, \quad succ(zero), \quad succ(succ(zero)), \quad succ(succ(succ(zero))), ...$$

Using lambda-calculus notation, these terms can be re-stated as follows:

$$zero, \quad (succ\ zero), \quad (succ\ (succ\ zero)), \quad (succ\ (succ\ (succ\ zero))), ...$$

To arrive at the desired representation for natural numbers, we note that the names *zero* and *succ* are not significant. We therefore abstract these names using lambda-abstraction, as follows:

$$\lambda s.\lambda z.z, \quad \lambda s.\lambda z.(s\ z), \quad \lambda s.\lambda z.(s\ (s\ z)), \quad \lambda s.\lambda z.(s\ (s\ (s\ z)))), \quad \ldots$$

The precise motivation for abstracting these names will be clear when operations on numbers are defined. This idea of representing the natural numbers is due to Alonzo Church, and these representations are hence referred to as *Church numerals*. It is easy to see that this idea generalizes to any data type, e.g. lists, trees, etc., which are represented by a finite set of constructors.

To understand how simple arithmetic operations are defined, it is worth examining closely the definition of the successor function on Church numerals:

let $\mathbf{succ} = \lambda n.\lambda f.\lambda x.((n\ \ f)\ \ (f\ \ x))$

This function takes as input a Church numeral n, which would be of the form $\lambda s.\lambda z.\ldots$, and returns a new Church numeral of the form $\lambda f.\lambda x.\ldots$ (using two constructors f and x). The body of this new numeral is obtained by replacing s by f and z by $(f\ \ x)$ in the body of the input numeral. This replacement is done by applying n to the arguments f and $(f\ \ x)$ as shown in the definition of **succ**. This example should make it clear why abstractions were introduced in the representation of natural numbers. To see how the above definition works, let us reduce the term $(\mathbf{succ}\ \ \mathbf{0})$, where $\mathbf{0}$ abbreviates $\lambda s.\lambda z.z$:

$$\begin{aligned}
&(\mathbf{succ}\ \ \mathbf{0})\\
=\ &(\lambda n.\lambda f.\lambda x.((n\ \ f)\ \ (f\ \ x))\ \ \lambda s.\lambda z.z)\\
\Longrightarrow\ &\lambda f.\lambda x.((\lambda s.\lambda z.z\ \ f)\ \ (f\ \ x))\\
\Longrightarrow\ &\lambda f.\lambda x.(\lambda z.z\ \ (f\ \ x))\\
\Longrightarrow\ &\lambda f.\lambda x.(f\ \ x)
\end{aligned}$$

The idea used in the definition of the **succ** definition can be readily generalized to define the addition and multiplication operations on two Church numerals, as follows:

let **add** $= \lambda n_1.\lambda n_2.\lambda f.\lambda x.((n_1 \;\; f) \;\; ((n_2 \;\; f) \;\; x))$

let **mult** $= \lambda n_1.\lambda n_2.\lambda f.\lambda x.((n_1 \;\; (n_2 \;\; f)) \;\; x)$

Exercise 10: Which arithmetic operation on Church numerals does the following lambda-term represent?

$\lambda n_1.\lambda n_2.(n_2 \;\; n_1)$

The following is a definition for the operation **iszero** which returns a boolean indicating whether the input Church numeral represents the number zero or not.

let **iszero** $= \lambda n.((n \;\; \lambda x.\textbf{false}) \;\; \textbf{true})$

The correctness of this definition can be easily verified by showing that $(\textbf{iszero} \;\; \lambda s.\lambda z.z) \Longrightarrow^* \textbf{true}$, and $(\textbf{iszero} \;\; N) \Longrightarrow^* \textbf{false}$ for any Church numeral $N \neq \lambda s.\lambda z.z$.

Before proceeding to representation of recursive definitions, we show how to represent the familiar Lisp primitives for building and accessing components of binary trees:

let **cons** $= \lambda l.\lambda r.\lambda c.((c \;\; l) \;\; r)$

let **car** $= \lambda c.(c \;\; \lambda l.\lambda r.l)$

let **cdr** $= \lambda c.(c \;\; \lambda l.\lambda r.r)$

It can easily be verified that **(car (cons l r))** $\Longrightarrow^*$ **l** and **(cdr (cons l r))** $\Longrightarrow^*$ **r**.

Exercise 11: Devise a representation for the predecessor function, **pred**, on Church numerals. *Note: This representation is not so obvious. Hint: Suppose that we want to find the predecessor of Church numeral n. First define a function that maps* $\textbf{cons}(i, \; j)$ *to* $\textbf{cons}(i + 1, \; i)$, *then compose this function n times on the input* $\textbf{cons}(0, \; 0)$, *and finally select out the desired answer.*

3.2 Recursive Definitions

The representation of recursive function definitions in the lambda-calculus is one of the most intriguing ideas in programming languages. Consider the following recursive definition of the multiplication operation (ignore that we already have a nonrecursive definition for this operation):

letrec **times** $= \lambda n_1.\lambda n_2.(\textbf{if} \; (\textbf{iszero} \; n_1) \; \textbf{0} \; (\textbf{add} \; n_2 \; (\textbf{times} \; (\textbf{pred} \; n_1) \; n_2)))$

We use "letrec" instead of "let" to emphasize that the definition is recursive. If it were not for the use of **times** on the right-hand side, the right-hand side term can be translated into a lambda-term in a straightforward way (assuming, of course, that a definition of **pred** is available to us). In order to obtain a definition in the lambda-calculus, we first abstract the name **times** on the right-hand side as follows:

let $\mathbf{t} = \lambda f.\lambda n_1.\lambda n_2.(\mathbf{if}\ (\mathbf{iszero}\ n_1)\ \mathbf{0}\ (\mathbf{add}\ n_2\ ((f\ (\mathbf{pred}\ n_1))\ n_2)))$

Note that the definition of **t** is not equivalent to the definition of **times**. In order to obtain the desired non-recursive definition of **times**, we need to find the *fixed point* of the transformation defined by **t**. The fixed point is a function g such that $(\mathbf{t}\ g) = g$. It should be clear that this fixed point would also satisfy the original recursive definition. In the general case, a transformation could have many fixed points, and here we would be interested in the *least fixed point*. Multiple fixed points could arise when the function being defined is nonterminating on some inputs. The least fixed point simply avoids making arbitrary choices on such inputs, whereas other fixed points could. To keep these notes brief, we do not delve into the theory of fixed-points; the interested reader would find a readable account in Stoy's book on Denotational Semantics.

The following is a definition of a general fixed-point operator:

let $\mathbf{Y} = \lambda f.(\lambda x.(f\ \ (x\ \ x))\ \ \lambda x.(f\ \ (x\ \ x)))$

For any lambda-term T, the term $(\mathbf{Y}\ \ T)$ gives a fixed point of T, i.e.,

$(\mathbf{Y}\ \ T) \Longleftrightarrow^* (T\ \ (\mathbf{Y}\ \ T))$.

When a leftmost reduction strategy is used, $(\mathbf{Y}\ \ T)$ would in general reduce to the least fixed-point of T. For our transformation **t** given earlier, the desired fixed point is given by $(\mathbf{Y}\ \ \mathbf{t})$. Since **Y** and **t** are non-recursive, $(\mathbf{Y}\ \ \mathbf{t})$ is a non-recursive equivalent of the **times** definition! Before we test this claim, first let us prove that **Y** is indeed a fixed point operator:

$$\begin{aligned}
&(\mathbf{Y}\ \ T)\\
&= (\lambda f.(\lambda x.(f\ \ (x\ \ x))\ \ \lambda x.(f\ \ (x\ \ x)))\ \ T)\\
&\Longleftrightarrow (\lambda x.(T\ \ (x\ \ x))\ \ \lambda x.(T\ \ (x\ \ x)))\\
&\Longleftrightarrow (T\ \ (\lambda x.(T\ \ (x\ \ x))\ \ \lambda x.(T\ \ (x\ \ x))))\\
&\Longleftrightarrow (T\ \ (\mathbf{Y}\ \ T))
\end{aligned}$$

Returning to the non-recursive equivalent of **times**, let us reduce the following expression to normal form, to see how the **Y** operator works:

$(((\mathbf{Y}\ \ \mathbf{t})\ \ \mathbf{1})\ \ \mathbf{k})$

The above expression represents the multiplication of **1** and **k**, where **1** stands for $\lambda s.\lambda z.(s\ \ z)$, and **k** is the Church numeral representation of some number k. We expect **k** to be result.

The reduction sequence (choosing leftmost redexes) that represents this computation is as follows:

$$
\begin{aligned}
&(((\mathbf{Y}\ \ \mathbf{t})\ \ \mathbf{1})\ \ \mathbf{k})\\
&\Longrightarrow (((\mathbf{t}\ \ (\mathbf{Y}\ \ \mathbf{t}))\ \ \mathbf{1})\ \ \mathbf{k})\\
&= (((\lambda f.\lambda n_1.\lambda n_2.(\mathbf{if}\ \ (\mathbf{iszero}\ n_1)\ \ \mathbf{0}\ \ (\mathbf{add}\ n_2\ ((f\ (\mathbf{pred}\ n_1))\ n_2)))\ \ (\mathbf{Y}\ \ \mathbf{t}))\ \ \mathbf{1})\ \ \mathbf{k})\\
&\Longrightarrow ((\lambda n_1.\lambda n_2.(\mathbf{if}\ \ (\mathbf{iszero}\ n_1)\ \ \mathbf{0}\ \ (\mathbf{add}\ n_2\ (((\mathbf{Y}\ \ \mathbf{t})\ (\mathbf{pred}\ n_1))\ n_2)))\ \ \mathbf{1})\ \ \mathbf{k})\\
&\Longrightarrow (\lambda n_2.(\mathbf{if}\ \ (\mathbf{iszero}\ \mathbf{1})\ \ \mathbf{0}\ \ (\mathbf{add}\ n_2\ (((\mathbf{Y}\ \ \mathbf{t})\ (\mathbf{pred}\ \mathbf{1}))\ n_2)))\ \ \mathbf{k})\\
&\Longrightarrow (\mathbf{if}\ \ (\mathbf{iszero}\ \mathbf{1})\ \ \mathbf{0}\ \ (\mathbf{add}\ \mathbf{k}\ (((\mathbf{Y}\ \ \mathbf{t})\ (\mathbf{pred}\ \mathbf{1}))\ \mathbf{k})))\\
&\Longrightarrow (((\mathbf{iszero}\ \mathbf{1})\ \ \mathbf{0})\ \ (\mathbf{add}\ \mathbf{k}\ (((\mathbf{Y}\ \ \mathbf{t})\ (\mathbf{pred}\ \mathbf{1}))\ \mathbf{k})))\\
&\Longrightarrow^* ((\mathbf{false}\ \ \mathbf{0})\ \ (\mathbf{add}\ \mathbf{k}\ (((\mathbf{Y}\ \ \mathbf{t})\ (\mathbf{pred}\ \mathbf{1}))\ \mathbf{k})))\\
&= ((\lambda x.\lambda y.y\ \ \mathbf{0})\ \ (\mathbf{add}\ \mathbf{k}\ (((\mathbf{Y}\ \ \mathbf{t})\ (\mathbf{pred}\ \mathbf{1}))\ \mathbf{k})))\\
&\Longrightarrow^* (\mathbf{add}\ \mathbf{k}\ (((\mathbf{Y}\ \ \mathbf{t})\ (\mathbf{pred}\ \mathbf{1}))\ \mathbf{k}))
\end{aligned}
$$

The derivation is continued in a similar manner: since $(\mathbf{pred}\ \mathbf{1}) \Longrightarrow^* \mathbf{0}$ and $(((\mathbf{Y}\ \ \mathbf{t})\ \mathbf{0})\ \mathbf{k}) \Longrightarrow^* \mathbf{0}$, it is easy to see that $(\mathbf{add}\ \mathbf{k}\ \mathbf{0}) \Longrightarrow^* \mathbf{k}$. Note the importance of choosing the leftmost redex at each step. It is possible to choose the inner redex $(\mathbf{Y}\ \ \mathbf{t})$ at every step, but doing so would lead to a nonterminating reduction. By uniformly choosing the leftmost redex at each step, we indeed obtain the least fixed-point.

Exercise 12: Devise a representation for a $list$ datatype over the constructors $\mathbf{nil}$ and $\mathbf{cons}(elem, list)$ with operations $\mathbf{first}$, $\mathbf{rest}$, $\mathbf{null}$. Given this representation, translate the following definition into a non-recursive form and test it to append two lists:

$$\text{letrec } \mathbf{append} = \lambda l_1.\lambda l_2.\mathbf{if}(\mathbf{null}(l_1),\ \ l_2,\ \ \mathbf{cons}(\mathbf{first}(l_1),\ \mathbf{append}(\mathbf{rest}(l_1),\ l_2))).$$

We digress briefly to make some remarks on the *combinator calculus*: A lambda-term that has no free variables is called a *closed term*, and is also referred to as a *combinator*. There are three famous combinators, called $\mathbf{S}$, $\mathbf{K}$, and $\mathbf{I}$:

$$\mathbf{S} = \lambda f.\lambda g.\lambda x.((f\ \ x)\ \ (g\ \ x))$$

$$\mathbf{K} = \lambda x.\lambda y.x$$

$$\mathbf{I} = \lambda x.x$$

The combinator calculus (also called *combinatory logic*) is interesting because terms built up of $\mathbf{S}$, $\mathbf{K}$, variables and application can represent any computable function. (The $\mathbf{I}$ combinator is not needed because $\mathbf{I} = (\mathbf{S\ K\ K})$.) The use of combinators also avoids the variable-capture problem of lambda-abstractions. As a historical note, combinators were first introduced by Moses Schönfinkel in the 1920s and later developed by Haskell B. Curry. Curry also influenced the development of the lambda-calculus, by suggesting the use of Greek letters for the names of conversion rules—Church's original name for α-conversion was *Rule of procedure I*, and for β-conversion was *Rule of procedure II.* In the late 1970s, David Turner showed how to compile functional programming languages using a combinator-based instruction set. This idea was developed further in the 1980s, and it underlies the implementation techniques for modern functional languages.

Expressions built up from **S**, **K** and **I** tend to be rather huge. For example, the **Y** operator introduced earlier, also called the *fixed-point combinator*, can be defined in terms of **S**, **K** and **I** as follows:

$$\begin{aligned}\mathbf{Y} = (&\mathbf{S\ (S\ (S\ (K\ S)\ (S\ (K\ K)\ I))\ (K\ (S\ I\ I)))}\\ &\quad\mathbf{(S\ (S\ (K\ S)\ (S\ (K\ K)\ I))\ (K\ (S\ I\ I))))}\end{aligned}$$

There is a systematic way to translate a lambda-term into a combinator expression; the reader may consult one of the references cited earlier for this translation. The following combinators are useful in obtaining more compact representations of functions, but they can also be defined in terms of **S** and **K**:

$$\mathbf{B} = \lambda x.\lambda y.\lambda z.(x\ \ (y\ \ z))$$

$$\mathbf{C} = \lambda x.\lambda y.\lambda z.((x\ \ z)\ \ y)$$

Exercise 13: Prove the following equalities:

$$\mathbf{I} = (\mathbf{S\ K\ K})$$

$$\mathbf{B} = (\mathbf{S\ (K\ S)\ K})$$

$$\mathbf{C} = (\mathbf{S\ ((B\ B)\ S)\ (K\ K)})$$

3.3 Imperative Constructs

We now examine how to encode imperative programming constructs in the lambda-calculus. Since we have already seen how to represent data types and recursive definitions, we focus now on programs made up of assignment statements, sequencing, and other simple control structures. The basic idea is to represent each statement s of an imperative program by a function ϕ_s and to represent the entire program by composing the constituent functions together in some way. This raises two issues:

What is the input and output of ϕ_s?

How do we compose the constituent ϕ_{s_i} functions together?

It should be clear that the input to ϕ_s should at the very least consist of the values of the variables occurring in the statement s. To have a uniform scheme, we make ϕ_s a function of all the variables in the program. We will refer to these variables collectively as the *environment*. To see what should be the output of ϕ_s, note that the effect of statement s would be to produce new bindings for some of the variables in the environment. Hence the output of ϕ_s can also be thought of as an environment. In order to compose the constituent functions for individual statements together, we use the concept of the *continuation* of a program. For any statement s in a program, the continuation of s is a function Φ that represents the remainder of the program, i.e., the continuation carries out the remaining computation after statement s finishes, assuming that s provides it with suitable inputs. To obtain a uniform scheme, we make ϕ_s also take in the continuation as an additional argument, and this continuation is invoked with the environment at the end of s.

We make the above ideas more concrete by discussing lambda-term representations for the assignment statement and control structures.

3.3.1 Assignments and Sequencing

Assuming that the environment consists of n variables v_1, …, v_n, an assignment statement of the form

$$v_i := expr$$

is represented by the lambda-term

$$\lambda\Phi.\lambda v_1.\ldots.\lambda v_n.(\ldots((((\ldots(\Phi \;\; v_1) \;\; \ldots) \;\; v_{i-1}) \;\; \mathbf{expr}) \;\; v_{i+1})\ldots v_n)$$

This term expresses the idea that the continuation Φ of the assignment statement is invoked with a new environment in which all variables remain unchanged except that v_i has the value **expr**, which is the lambda-term representation of $expr$.

Assuming that statements s_1, s_2, …, s_n are represented respectively by $\mathbf{s}_1$, $\mathbf{s}_2$, …, $\mathbf{s}_n$, a sequence of n statements

begin s_1; s_2; … s_n end

can be represented by the lambda-term

$$\lambda\Phi.(\mathbf{s}_1 \;\; (\mathbf{s}_2\ldots(\mathbf{s}_n \;\; \Phi)\ldots)).$$

This composition expresses our intention of passing the continuation Φ to the function $\mathbf{s}_n$ corresponding to the last statement.

We illustrate the above ideas by showing the representation for the following sequence of three assignment statements (refer to the previous subsection for the definitions of **0**, **1**, and **add**):

Program Fragment	**Statement Representations**
begin	
$a := 0;$	let $\mathbf{s}_1 = \lambda\Phi.\lambda a.\lambda b.((\Phi\ \mathbf{0})\ b)$
$b := a + 1;$	let $\mathbf{s}_2 = \lambda\Phi.\lambda a.\lambda b.((\Phi\ a)\ (\mathbf{add}\ a\ 1))$
$a := a + b$	let $\mathbf{s}_3 = \lambda\Phi.\lambda a.\lambda b.((\Phi\ (\mathbf{add}\ a\ b))\ b)$
end	

The representation of the statement $a := 0$ basically takes as input a continuation and invokes this continuation with a replaced by $\mathbf{0}$ and b replaced by b, i.e., b is left unaltered. The representations of statements $a := a + 1$ and $a := a + b$ are similarly constructed. The composition of the constituent functions $\mathbf{s}_1$, $\mathbf{s}_2$ and $\mathbf{s}_3$ gives the representation for the entire sequence:

let $\mathbf{s} = \lambda\Phi.(\mathbf{s}_1\ \ (\mathbf{s}_2\ \ (\mathbf{s}_3\ \ \Phi)))$.

Let us reduce the above term to normal form (choosing leftmost redexes) to see how it works:

$$
\begin{aligned}
&\lambda\Phi.(\mathbf{s}_1\ \ (\mathbf{s}_2\ \ (\mathbf{s}_3\ \ \Phi)))\\
&= \lambda\Phi.(\lambda\Phi.\lambda a.\lambda b.((\Phi\ \mathbf{0})\ b)\ \ (\mathbf{s}_2\ \ (\mathbf{s}_3\ \ \Phi)))\\
&\Longrightarrow \lambda\Phi.\lambda a.\lambda b.(((\mathbf{s}_2\ \ (\mathbf{s}_3\ \ \Phi))\ \ \mathbf{0})\ \ b)\\
&= \lambda\Phi.\lambda a.\lambda b.(((\lambda\Phi.\lambda a.\lambda b.((\Phi\ a)\ (\mathbf{add}\ a\ \mathbf{1}))\ \ (\mathbf{s}_3\ \ \Phi))\ \ \mathbf{0})\ \ b)\\
&\Longrightarrow^* \lambda\Phi.\lambda a.\lambda b.(((\mathbf{s}_3\ \ \Phi)\ \ \mathbf{0})\ (\mathbf{add}\ \mathbf{0}\ \mathbf{1}))\\
&= \lambda\Phi.\lambda a.\lambda b.(((\lambda\Phi.\lambda a.\lambda b.((\Phi\ (\mathbf{add}\ a\ b))\ b)\ \ \Phi)\ \ \mathbf{0})\ (\mathbf{add}\ \mathbf{0}\ \mathbf{1}))\\
&\Longrightarrow^* \lambda\Phi.\lambda a.\lambda b.((\Phi\ (\mathbf{add}\ \mathbf{0}\ (\mathbf{add}\ \mathbf{0}\ \mathbf{1})))\ \ (\mathbf{add}\ \mathbf{0}\ \mathbf{1}))\\
&\Longrightarrow^* \lambda\Phi.\lambda a.\lambda b.((\Phi\ \mathbf{1})\ \mathbf{1})
\end{aligned}
$$

The above normal form represents a function that takes a continuation Φ in which variables a and b will both be substituted by the value 1. It is clear that this term correctly models the program fragment shown, since variables a and b will both hold the value 1 at the end of that fragment.

3.3.2 Conditionals and Iteration

The representation of conditional statements is based upon the representation of boolean operations and the **if** abbreviation introduced in section 3.1. Thus, if the representation of the condition b is $\mathbf{b}$ and that of statement s_1 is $\mathbf{s}_1$ and that of statement s_2 is $\mathbf{s}_2$, then we have the following representations for the two types of conditional statements (note that $\mathbf{I} = \lambda\Phi.\Phi$ and $\bar{v}$ stands for the list of variables $v_1, \ldots, v_n$):

if b then s_1 else s_2	$\lambda\Phi.\lambda\bar{v}.(((\mathbf{if}\ \ \mathbf{b}\ \ \mathbf{s}_1\ \ \mathbf{s}_2)\ \ \Phi)\ \ \bar{v})$

if b then s_1 $\lambda\Phi.\lambda\bar{v}.(((\mathbf{if}\ \ \mathbf{b}\ \ \mathbf{s}_1\ \ \mathbf{I})\ \ \Phi)\ \ \bar{v})$

The above definitions basically use the **if** function to select out the representation for the **then** or **else** part. The continuation Φ is then passed on to the selected representation. Note that the representation **b** of the predicate could contain free occurrences of the variables in $\bar{v}$. To illustrate the above scheme, we develop the representation for the following program:

```
begin
    y := 0;
    if y = 0 then x := 1 else x := 0
end
```

Let s_1 be $y := 0$, s_2 be if $y = 0$ then $x := 1$ else $x := 0$, s_3 be $x := 1$, and s_4 be $x := 0$. We then have the following representations for these statements:

$$\mathbf{s}_1 = \lambda\Phi.\lambda x.\lambda y.((\Phi\ \ x)\ \ \mathbf{0})$$

$$\mathbf{s}_2 = \lambda\Phi.\lambda x.\lambda y.((((\mathbf{if}\ (\mathbf{iszero}\ y)\ \mathbf{s}_3\ \mathbf{s}_4)\ \ \Phi)\ x)\ y)$$

$$\mathbf{s}_3 = \lambda\Phi.\lambda x.\lambda y.((\Phi\ \ \mathbf{1})\ \ y)$$

$$\mathbf{s}_4 = \lambda\Phi.\lambda x.\lambda y.((\Phi\ \ \mathbf{0})\ \ y)$$

Let us reduce the term $\lambda\Phi.(\mathbf{s}_1\ (\mathbf{s}_2\ \Phi))$ to understand the above representations better:

$$\lambda\Phi.(\mathbf{s}_1\ (\mathbf{s}_2\ \Phi))$$

$$= \lambda\Phi.(\lambda\Phi.\lambda x.\lambda y.((\Phi\ \ x)\ \ \mathbf{0})\ \ (\mathbf{s}_2\ \Phi))$$

$$\Longrightarrow \lambda\Phi.\lambda x.\lambda y.(((\mathbf{s}_2\ \Phi)\ \ x)\ \ \mathbf{0})$$

$$=\lambda\Phi.\lambda x.\lambda y.(((\lambda\Phi.\lambda x.\lambda y.((((\mathbf{if}\ (\mathbf{iszero}\ y)\ \mathbf{s}_3\ \mathbf{s}_4)\ \ \Phi)\ x)\ y)\ \ \Phi)\ \ x)\ \ \mathbf{0})$$

$$\Longrightarrow^* \lambda\Phi.\lambda x.\lambda y.((((\mathbf{if}\ (\mathbf{iszero}\ \mathbf{0})\ \mathbf{s}_3\ \mathbf{s}_4)\ \ \Phi)\ \ x)\ \ \mathbf{0})$$

$$\Longrightarrow^* \lambda\Phi.\lambda x.\lambda y.(((\mathbf{s}_3\ \Phi)\ \ x)\ \ \mathbf{0})$$

$$= \lambda\Phi.\lambda x.\lambda y.(((\lambda\Phi.\lambda x.\lambda y.((\Phi\ \ \mathbf{1})\ \ y)\ \Phi)\ \ x)\ \ \mathbf{0})$$

$$\Longrightarrow^* \lambda\Phi.\lambda x.\lambda y.((\Phi\ \ \mathbf{1})\ \ \mathbf{0})$$

The resulting term represents the correct final state in which variable x has value **1** and y has value **0**.

To represent an iterative statement of the form while b do s

we first give a recursive definition assuming, as before, that b is represented by $\mathbf{b}$ and s is represented by $\mathbf{s}$:

letrec $\mathbf{whiledo} = \lambda\Phi.\lambda\bar{v}.((\mathbf{if}\ \ \mathbf{b}\ \ (\mathbf{s}\ \ (\mathbf{whiledo}\ \ \Phi))\ \ \Phi)\ \ \bar{v})$

This recursive definition states that, if the condition **b** reduces to **true**, the statement **s** is performed, followed by a re-execution of **whiledo** with the same continuation Φ; if the condition **b** reduces to **false**, the continuation Φ is invoked. A non-recursive definition can now be obtained by abstracting the **while** on the right-hand side and taking the fixed point of the resulting term with the help of the **Y** combinator:

let $\mathbf{while} = \lambda w.\lambda\Phi.\lambda\bar{v}.((\mathbf{if}\ \ \mathbf{b}\ \ (\mathbf{s}\ \ (w\ \ \Phi))\ \ \Phi)\ \ \bar{v})$

Then $(\mathbf{Y}\ \ \mathbf{while})$ represents the desired representation of the while-do statement.

Exercise 14: Show the representation for the following program, and reduce it to normal form:

```
begin
    $(f, n) := (1, 3)$;
    while $not(n = 0)$ do
        $f := f * n$;
        $n := n - 1$
    end
end
```

Exercise 15: Devise a representation for the repeat-until statement.

4. Typed Lambda Calculi

Church introduced typed lambda calculi in the 1940s in recognition of the semantic difficulties with the untyped calculus. To appreciate these difficulties, consider, for example, the untyped lambda-term

$$\lambda x.(\underline{x}\ \ (\underline{x}\ \ \underline{x}))$$

What function does it stand for? Consider the third underlined occurrence of x and suppose the type of x were taken to T. Considering the second underlined occurrence of x, its type must be of the form $T \to U$ for some type U. And considering the first underlined occurrence of x, its type must be of the form $U \to V$ for some type V. Since a variable can have only one type, all these three types must be made compatible, and thus the type T must be made compatible with the type $T \to T$. If we regard types as sets and functions as sets of ordered pairs—the prevalent view in the 1940s—we need to construct a set T that is isomorphic to the set of functions $T \to T$. If the size of T is n, the size of $T \to T$, *i.e.*, the number of functions in this set, is n^n. Since $n \neq n^n$ (for all $n > 1$), we cannot assign a reasonable set-theoretic meaning to the above lambda-term.

Thus, while the lambda-calculus had a clear *operational semantics* (in terms of the reduction rules), in the 1940s no one could give a *denotational semantics* (in terms for mathematical functions). In fact, the lambda-calculus was believed not to have a denotational semantics, *i.e.*, the lambda-calculus was believed to be inconsistent. This impasse was broken by Dana Scott in the early 1970s who showed that if a type T is not thought of as an *arbitrary* set, but instead as a *structured* set, also called a *domain*, and if $T \to T$ is not thought of as the set of *arbitrary* functions but instead as the set of *continuous functions* from domain T to itself, then we can make T isomorphic to $T \to T$. This result was significant because the semantics of nearly all procedural and functional languages can be reduced to that of the lambda-calculus. A very readable account of these details is in Stoy's book on Denotational Semantics. This result earned Scott the ACM Turing Award in 1976.

4.1 Simply Typed Lambda Calculus

The simply-typed lambda-calculus was proposed by Church in the early 1940s as a way to circumvent the semantic problems with the untyped lambda-calculus. The key idea was to introduce *types* with all variables. Given a set B of *basic types*, the set of *simple types* is defined as

$$\tau \quad ::= \quad B \quad | \quad \tau \to \tau$$

Suppose i and o are basic types—Church referred to them as 'iota' and 'omicron', for the types of *individuals* and *propositions* respectively. The following are examples of simple types, where $\to$ associates to the right:

$i \to o$
$i \to i \to o$
$(i \to o) \to o$

The simple types basically define a hierarchy of functional types starting from some basic types B. The basic types are any pre-existing sets. Simple types are formulated with the idea that types are sets and functions are mappings between sets. (This view, of course, does not work for the untyped lambda-calculus.) A simple type is also called a *monotype*, *i.e.*, no polymorphism is permitted. In the simply-typed lambda-calculus, an application $(f \;\; e)$ is said to be well-typed if $f : \tau_1 \to \tau_2$ and $e : \tau_1$ for some simple types τ_1 and τ_2. In this case $(f \;\; e) : \tau_2$.

Note that self-application terms such as $(x \;\; x)$ cannot be given a simple type because, given the set-theoretic basis for simple types, a simple type $\alpha \to \beta$ is necessarily different from the simple types α and β. As a consequence of banishing terms such as $(x \;\; x)$, one can show that all reductions from every simply-typed lambda-term terminate. This is called the *strong normalization* property. If every program in a programming language terminates, the language does not have the full computational power of Turing machines or the (untyped) lambda-calculus. Thus, we may ask: how expressive is the simply-typed lambda-calculus? The paper by Fortune, Leivant, and O'Donnell (*JACM*, 1983) discusses this issue in some detail. We give one of their results, without going into the details:

> **Theorem**: *Consider a basic type i and Church numerals of type $n = (i \to i) \to (i \to i)$. The functions representable by simply-typed terms of type*
>
> $(n \to (n \to \ldots (n \to n) \ldots))$
>
> *are exactly the functions generated by constants* **0** *and* **1** *using the operations addition, multiplication, and conditional. These are the multivariate polynomials extended with the conditional function.*

It is interesting to note that exponentiation and the predecessor function can be represented only if different Church numerals may have different types (over B). And subtraction cannot be represented at all!

Exercise 16: To get a feel for the restriction imposed by types, enumerate all possible *closed* terms of types: $i \to i$, $i \to (i \to i)$, $(i \to i) \to i$, etc.

4.2 Second-Order Typed Lambda-Calculus

The second-order typed-lambda-calculus was invented in order to expand the expressiveness of the simply-typed calculus. (A standard reference is the book Type Theory with Type Variables, by P.B. Andrews, North-Holland publishers, 1965.) The term *second-order* applies to the type system. We may call the type system of the simply-typed lambda-calculus *first-order* because it has no type variables: a simple type is either a type constant (*i.e.*, a member of the basic type) or a function type that is built up from type constants. Given a set B of *basic types* and a set V of *type variables*, the set of *second-order types* is defined as

$$\tau \quad ::= \quad B \quad | \quad V \quad | \quad \tau \to \tau \quad | \quad \forall t.\tau$$

where t in the above definition is a type variable. In addition, we require every variable occurring in a type expression to lie in the lexical scope of a surrounding universal quantifier. Examples of second-order types are given below (assuming o is a basic type).

$\forall t.t \to t$
$\forall t.t \to t \to o$
$\forall t.(t \to t) \to (t \to t)$
$\forall t1.\forall t2.\forall t3.(t1 \to t2) \to (t2 \to t3) \to (t1 \to t3)$

Let us see why second-order types are needed. Consider the untyped λ-term

$\lambda f.\lambda x.(f \quad (f \quad x))$.

In order for this term to be meaningful, the variable f cannot stand for arbitrary functions, but must stand for functions that have identical input and output types. To express this constraint on the type of f, we write a second-order typed term as follows:

let **twice** $= \Lambda t.\ \ \lambda f : t \to t.\ \ \lambda x : t.\ \ (f \quad (f \quad x))$

Here, Λ stands for *type abstraction*. It introduces a *type variable* t which is used subsequently giving the types for f and x. Just as we have abstraction and application at the *term-level*, we also have abstraction and application at the *type-level*. The notation for *type application* is $[t_1 \quad t_2]$. The function **twice** has a second-order type

$\forall t.\ (t \to t) \to (t \to t)$.

This type is also referred to as a *universal type* due to the use of the universal quantifier.

Despite the increased expressiveness of the second-order-typed lambda-calculus, all reductions from every second-order typed lambda-term terminate. We provide below a couple of examples to illustrate reduction of second-order typed lambda-terms. In the first example, we see one use of type application, [**twice** *int*]. In the second example, we see that the type abstraction Λt in the definition of **twice** is applied in two different ways: once with a type constant, [**twice** *int*], and a second time with a functional type, [**twice** $int \to int$].

Example 1 (let **succ** : $int \to int$ and **0** : *int*)

$(([\mathbf{twice}\ \ int] \quad \mathbf{succ}) \quad \mathbf{0})$

$\Rightarrow ((\lambda f : int \to int.\ \ \lambda x : int.\ \ (f \quad (f \quad x)) \quad \mathbf{succ}) \quad \mathbf{0})$

$\Rightarrow (\lambda x : int.\ \ (\mathbf{succ} \quad (\mathbf{succ} \quad x)) \quad \mathbf{0})$

$\Rightarrow (\mathbf{succ} \quad (\mathbf{succ} \quad \mathbf{0}))$

$\Rightarrow^* \mathbf{2}$

Example 2

$((([\mathbf{twice}\ \ int \to int] \quad [\mathbf{twice}\ \ int]) \quad \mathbf{succ}) \quad \mathbf{0})$

$\Rightarrow (((\ \lambda f : (int \to int) \to (int \to int).$

$\qquad \lambda x : int \to int.\ \ (f \quad (f \quad x)) \quad [\mathbf{twice}\ \ int]) \quad \mathbf{succ}) \quad \mathbf{0})$

$\Rightarrow^* (([\mathbf{twice}\ \ int] \quad ([\mathbf{twice}\ \ int] \quad \mathbf{succ})) \quad \mathbf{0})$

$\Rightarrow^* (\lambda x : int.\ \ (([\mathbf{twice}\ \ int] \quad \mathbf{succ}) \quad (([\mathbf{twice}\ \ int] \quad \mathbf{succ}) \quad x)) \quad \mathbf{0})$

$\Rightarrow$ ((**[twice** int] **succ**) ((**[twice** int] **succ**) **0**))

$\Rightarrow^*$ ((**[twice** int] **succ**) **2**)

$\Rightarrow^*$ **4**

Using Church numerals of the form

$\Lambda t.\ \ \lambda f : t \to t.\ \ \lambda x : t.\ \ (f \ldots (f\ \ x) \ldots)$

exponentiation and subtraction can be represented as uniformly as addition, multiplication, *etc*. The expressiveness of the second-order typed lambda-calculus is given by the following theorem due to Fortune, Leivant, and O'Donnell.

> **Theorem**: *If $f : N \to N$ is representable in the second-order-typed λ-calculus, then so is any function $g : N \to N$ computable by a Turing machine in time bounded by f.*

We can use the second-order types to express the types of a polymorphically typed programming language. A type definition such as

$list(\alpha)$ `::=` `nil` | `cons`(α, $list(\alpha)$)

induces second-order types for the constructors `nil` and `cons`:

`nil`: $\forall t.\ list(t)$
`cons`: $\forall t.\ t \times list(t) \to list(t)$

Type applications are specified by type definitions such as

`type` $intlist = list(int)$

which induces the following *simple* types:

`nil`: $list(int)$
`cons`: $int \times list(int) \to list(int)$