Chapter 9

The Lambda Calculus: a minimal ML?

Note-taker of 31.10.05, 2.11.05: Sam Tucker

In this chapter, we explore the most minimal version of an ML-like language that still qualifies as a functional programming language. We’ll call it MiniMiniML, for now. Its syntax consists of essentially only two constructs: a term for describing a function’s application rule when applied to its formal parameter, and a term for application of a function term to a parameter term.

Here is the concrete syntax for MiniMiniML:

\[
\begin{align*}
E & \rightarrow \ (E) \\
E & \rightarrow \ x \\
E & \rightarrow \ E\ E \\
E & \rightarrow \ \text{fn } x \rightarrow E \\
x & \rightarrow \ \text{any valid variable identifier .}
\end{align*}
\]

Here is an example MiniMiniML expression:

\[
\text{fn } x \rightarrow \ \text{fn } y \rightarrow y\ x
\]

I describes a curried function that takes its two arguments and applies the second (as a function) to the first (as its argument).
We say that an expression of the form \( \text{fn} x \Rightarrow E \) binds occurrences of \( x \) in the subexpression \( E \). The occurrences of \( x \) and \( y \) in the subexpression \( y \ x \) are bound in the full expression \( \text{fn} x \Rightarrow \text{fn} y \Rightarrow y \ x \). Note that the grammar also allows variables to occur freely in its expressions. For example, the following is a valid expression:

\[
z \ \text{fn} x \Rightarrow \text{fn} y \Rightarrow y \ x \ z
\]
despite the fact that \( z \) is not bound, but free. We will allow free variables in our semantics, as well.

Our language has this corresponding abstract syntax terms of the forms \( \text{VAR}(x) \), \( \text{APPLY}(e_1, e_2) \), and \( \text{FN}(x, e) \) so that the expression

\[
\text{fn} x \Rightarrow \text{fn} y \Rightarrow y \ x
\]
has the AST term

\[
\text{FN}(x, \text{FN}(y, \text{APPLY}(\text{VAR}(x), \text{VAR}(y))))
\]
and the expression

\[
z \ \text{fn} x \Rightarrow \text{fn} y \Rightarrow y \ x \ z
\]
has the AST term

\[
\text{APPLY}(\text{VAR}(z), \text{FN}(x, \text{FN}(y, \text{APPLY}(\text{APPLY}(\text{VAR}(y), \text{VAR}(x)), \text{VAR}(z)))))
\]

Note that we assume that the parsing conventions of Standard ML apply to MiniML, namely, that application has higher precedence than function expression and is left-associative. The above term is then assumed to be parsed the same by our conventions as the fully parenthesized term

\[
z \ (\text{fn} x \Rightarrow (\text{fn} y \Rightarrow ((y \ x) \ z)))
\]
Generally, we parse a subexpression that starts with an \( \text{fn} \) token until we hit the first unmatched left parenthesis or the end of the full expression.

Using this minimal language we embark on an investigation of pure functional languages that focuses entirely on their functional aspects. Surprisingly, despite the absence of primitive operations and literals, we can encode in these functional terms the constructs of MiniML including conditional expression, variable binding and scope, integers and integer operations, and
pairs and lists. Thus, our investigation allows us to devise a language using very few constructs without essentially reducing the range of computations we can express.

In doing so, we will be inheriting a legacy from two figures that greatly influenced the practical construction of functional programming systems. The first, the logician Alonzo Church, pursued a similar investigation in his Lambda Calculus. He devised similar notation but for different purposes, namely to precisely define what exactly a function is, in the mathematical sense.

Church was a seminal figure in mathematical logic, and his investigation of his $\lambda$-calculus in the 1930s paralleled the development of recursive function theory and the conception of the Turing machine around the same time. His thinking occurred before general purpose computers had been built or even designed, and hence before programming languages were conceived.

Nonetheless, some of the ideas Church presented in this work were read by our second figure, John McCarthy, who incorporated them in his design of LISP. Among other notions, McCarthy adopted Church’s notation for LISP’s anonymous functions. The LISP S-expression

$$(\text{lambda (n) (+ n 1))}$$

is a way of describing the integer successor function, like the ML expression.

$$\text{fn n => n + 1}$$

To quote McCarthy(1979) from his candid “History of LISP” \footnote{1}{from http://www-formal.stanford.edu/jmc/history/lisp/lisp.html}

As a programming language, LISP is characterized by the following ideas: computing with symbolic expressions rather than numbers, representation of symbolic expressions and other information by list structure in the memory of a computer, representation of information in external media mostly by multi-level lists and sometimes by S-expressions, a small set of selector and constructor operations expressed as functions, composition of functions as a tool for forming more complex functions, the use of conditional expressions for getting branching into function definitions, the recursive use of conditional expressions as a sufficient tool for building computable functions, the use of
S-expressions for naming functions, the representation of LISP programs as LISP data, the conditional expression interpretation of Boolean connectives, the LISP function eval that serves both as a formal definition of the language and as an interpreter, and garbage collection as a means of handling the erasure problem...

...To use functions as arguments, one needs a notation for functions, and it seemed natural to use the λ-notation of Church(1941).² I didn’t understand the rest of his book, so I wasn’t tempted to try to implement his more general mechanism for defining functions. Church used higher order functionals instead of using conditional expressions. Conditional expressions are much more readily implemented on computers.

We will shortly also directly adopt Church’s original notation. We will also start to view the terms that we express more syntactically (perhaps unlike Church and his contemporaries). We will view his system as a rewrite system, as he fashioned it, and imbue our MiniML language with a rewrite-, rather than evaluation-, style operational semantics. In this style, a computation proceeds by a series of steps that each simplify the terms of a program’s expression. After such reduction steps are applied, the resulting unsimplifiable term is the program’s resulting value.

Having a rewrite semantics will make it easier for us to consider more carefully some alternative evaluation strategies for functional programs—say, to define and compare forms of eager and lazy evaluation. It will also hint at what is necessary for a system to perform lazy evaluation, and hint at some of the code optimizations available to the compilers and runtime systems of purely functional programming languages. In Church’s λ-calculus, the rewrite system is also particularly clean. It turns out that every term whose reduction terminates does so with a unique simplified term.

²Church, Alonzo. Calculi of Lambda Conversion, Princeton University Press, Princeton, New Jersey, 1941.
9.1  History

Let’s review the history of Church’s lambda calculus briefly. Its ideas first came from logicians who wanted to study functions more carefully. For example, Frege (1893) noticed that it suffices to consider functions of single arguments. Take for example the addition function that takes A and B and returns A + B. Frege saw that you can model this binary operation by two function applications: we devise a function ⊕ which, when applied to any A we get back a function that, when applied to B, outputs A + B. More succinctly, (⊕(A))(B) = A + B. This is of course the notion of curried functions, attributed instead to Schönfinkel and Haskell Curry (who not only lent his name to this notion, but to the functional language Haskell).

Schönfinkel (1924) and Curry (1928, 1930) independently initiated exploration of what became combinatory logic, developed more extensively by Curry and his colleagues (see Curry et al. (1958, 1972)). Roughly, the goal of combinator logic was to develop rules of logic that were variable-free. Expression of functions in combinatory logic used an alphabet of a fixed set of combinators, essentially functional terms that operated on other terms.

For example, in his work, Schönfinkel discovered the completeness of two functions, the combinators K and S. K is a function that whose behavior is given by (K(A))(B) = A. In other words, K applied to A yields a constant function that ignores its argument and returns the term A. The combinator S seems even more obtuse in its action—((S(A))(B))(C) = (A(C))(B(C)). In other words, S applied to A yields a function that when applied to B yields a function that when applied to C yields a result equivalent to the function A applied to C applied to the function B applied to C (more or less). We’ll return to our own investigation of these special combinators shortly.

Note that the common mathematical notation for writing function application can become quite cumbersome when working with curried function expressions and higher-order functions, in general. Rather than writing \( f(x) \) for “the function \( f \) applied to the argument \( x \)” we instead write \( f \ x \). This notation was adopted early by mathematical logicians. You will also often see \( P \ x \) stand for the assertion that “\( x \) has property \( P \)” in their writings.

---

\(^3\text{This history is culled from Rosser, J., “Highlights of the history of the lambda-calculus,” in Proceedings of the ACM symposium on LISP and Functional Programming, Pittsburgh, Pennsylvania, pps, 216 - 225, 1982.}\)
This is allied with our new functional notation. A predicate can be viewed as a function that yields a truth value. The generalization of this notation to higher-order functions can be adopted easily since, again, we take the view that all functions take only one argument. With this adoption, one writes \(((+_A)(+_B))\) instead as \(+_A+B\), using left-associativity as our friendly shepherd to the promised land of fewer parentheses.

One desire at the time of these developments was to have an equational theory for functions. That is, build a system that enables us to answer questions like “When is a functional term \(F\) equivalent to a functional term \(G\)?” A reasonable notion is that two terms \(F\) and \(G\) are equivalent when we can show that \(F\ x = G\ x\) for all terms \(x\).\(^4\) Church’s lambda calculus arose from the following question: given any term \(M\), say, composed of the combinators \(S\) and \(K\) and the variable \(x\), what is the function \(L\) whose application to \(x\) yields \(M\)? That is, for what \(L\) is \(L\ x = M\), regardless of the term substituted for \(x\)?

Church’s answer was that \(L = \lambda x. M\). In other words, the term \(\lambda x. M\) describes a function that, when applied to any \(x\), yields a result equivalent to that expressed by the term \(M\). And thus lambda was born. This notion of lambda abstraction is a powerful notion, but hopefully at this point it appears very tame. The legacy of Church is that in Standard ML we write \(\text{fn } x \Rightarrow x + 1\) to stand for the lambda abstraction \(\lambda x. +_1\), ultimately describing the integer successor function.

There is certainly more to this history, and more to its technical legacy. We refer to Barendregt(1984) for a more complete technical survey, and continue here with highlights that most directly illuminate our understanding of functional programming.

### 9.2 Lambda terms

Let’s throw out our MiniMiniML syntax and replace it with Church’s, keeping our parsing conventions (his also)

\[
\begin{align*}
E & \rightarrow (E) \\
E & \rightarrow x \\
E & \rightarrow E\ E
\end{align*}
\]

\(^{4}\)”We can laugh at them 75 years later. I mean, Schönfinkel, come on.” —Jim
9.2. LAMBDA TERMS

\[ E \rightarrow \lambda x. E \]
\[ x \rightarrow \text{any valid variable identifier} . \]

Thus, we write \( z \text{ fn } x \Rightarrow \text{ fn } y \Rightarrow y \times z \) instead as \( z \lambda x.\lambda y.yx z \). At times in our presentation of the lambda calculus, we’ll also mix in terms not strictly derivable from this grammar, for example we might write \( \oplus \) as \( \lambda x.\lambda y.x + y \).

We call the expressions derivable from the above grammar lambda terms, or simply \( \lambda \)-terms. We say that a lambda term of the form \( \lambda x. E \) is a lambda abstraction of \( E \) with respect to \( x \). \( E \) is the body of its lambda abstraction, more specifically, its \( x \)-body. A lambda term of the form \( \lambda x_1.\lambda x_2.\ldots\lambda x_k. E \) is a lambda abstraction with respect to \( x_1, x_2, \ldots, x_k \). \( E \) is its \( x_1, x_2, \ldots, x_k \)-body.

The abstract syntax abstract syntax of lambda terms is given by

\[
\begin{align*}
\lambda &: \text{ lambda-term} \rightarrow \text{ lambda-term} \\
\oplus &: \text{ lterm} \times \text{ lterm} \rightarrow \text{ lambda-term} \\
x &: ( ) \rightarrow \text{ lambda-term} \quad \text{for any } x \in \text{Id}.
\end{align*}
\]

Thus any term \( M \in \text{TERM}_{\text{lambda-term}} \) is a lambda term. We write instead that \( M \in \text{TERM}_\lambda \). The above example expression corresponds to the AST term \( \oplus(z, \lambda(x, \lambda(y, \oplus(y, x)))) \), or more visually,

\[
\begin{array}{c}
\oplus \\
z \\
x \\
y \\
y \\
x
\end{array}
\]

**Example 8.** According to our conventions, what are the full parenthesizations of the following lambda terms? Alternatively, what are their abstract syntax trees and/or terms?

1. \( z \lambda x.\lambda y.xy z \)
2. \( x(yx)z \)
(3) \( \lambda x. (\lambda y. y z) \lambda x. x \)
(4) \( \lambda x. \lambda y. \lambda z. y (x z) \)
(5) \( \lambda x. y (\lambda x. x) x \)

They are as follows:

(1’) \( z (\lambda x. (\lambda y. ((x y) z))) \)
(2’) \( ((x (y x)) z) \)
(3’) \( \lambda x. ((\lambda y. (y z)) (\lambda x. x)) \)
(4’) \( \lambda x. (\lambda y. (\lambda z. ((x y)(x z)))) \)
(5’) \( \lambda x. ((y(\lambda x. x) x). \)

The tree for (5) and (5’) is

\[
\lambda
\]

\[
\neg \neg \neg \neg \neg
\]

At this point, it is probably useful to have a brief discussion about what we plan to do with these \( \lambda \)-terms. Consider the following term.

\( (\lambda x. (\lambda y. x + y)) 3 5 \)

Clearly this is meant to communicate application of \( \oplus \), our curried add function, to the successive arguments 3 and 5. It seems natural to write that

\( (\lambda x. (\lambda y. x + y)) 3 5 \rightarrow (\lambda y. 3 + y) 5 \rightarrow 3 + 5. \)

The expression manipulations that allow us to rewrite the leftmost term above to the rightmost term, in order to deduce a resulting value of 8, are called reduction steps. In particular, Church specified a particular rule, called
\section{Bound and Free Variables}

Just as the variable $x$ is bound, and $a$ is free, in the expression

$$\int \sin(ax) \, dx$$

so too are the variables $x$ and $a$, respectively, in the $\lambda$-term

$$\lambda x.xa$$

Continuing this further, in the $\lambda$-terms

1. $z \lambda x.\lambda y.x y z$
2. $x \,(y \,x) \, z$
3. $\lambda x.(\lambda y,y \,z) \, \lambda x.x$
4. $\lambda x.\lambda y.\lambda z.x y \,(x \,z)$
5. $\lambda x.y(\lambda x.x) \, x$

we see that in (1) $z$ is free, in (2) all three variables are free, in (3) $z$ is free, in (4) no variable is free, and in (5) $y$ is free. The remaining variables within each expression are bound. Notice a limitation in our terminology: in the term

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

written more carefully as

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

the variable $x$ is both free and bound. There are two free occurrences of $x$ here, the first and the second, whereas the third within the term $(\lambda x.x)$ is bound.
Below, we make these notions more precise, but our formulation will still have these warts. There are formulations of the lambda-calculus that enable us to differentiate particular occurrences of the same variable $x$, to have a better naming scheme for variable terms, and to say easily which occurrences are bound and which are free. These do not entirely suit are purposes, here, so we refer the reader to Barendregt(1984). Instead, we proceed more recklessly with these more rudimentary definitions:

**Definition 14** (subterm). For any $M, N \in \text{Term}_\lambda$, $N$ is a *subterm* of $M$ whenever

- $M = N$,
- $M = \lambda x. M'$ and $N$ is a subterm of $M'$,
- $M = M_1 M_2$ and $N$ is a subterm of $M_1$, or
- $M = M_1 M_2$ and $N$ is a subterm of $M_2$.

**Definition 15** (free variables, combinator). The *free variables* of $M \in \text{Term}_\lambda$ are given by

$$
\text{fv}(M) := \begin{cases} 
\{x\} & \text{if } M = x \\
\text{fv}(M') \cup \{x\} & \text{if } M = \lambda x. M' \\
\text{fv}(M_1) \cup \text{fv}(M_2) & \text{if } M = M_1 M_2 \\
\emptyset & \text{if } M = \lambda y. M' \\
\text{fv}(M') \cap \{x\} \cup \text{bv}(M') & \text{if } M = \lambda x. M' \\
\emptyset & \text{if } M = x
\end{cases}
$$

If $\text{fv}(M) = \emptyset$, then $M$ is called a *combinator*.

**Definition 16** (bound variables). The *bound variables* of $M \in \text{Term}_\lambda$ are given by

$$
\text{bv}(M) := \begin{cases} 
\text{bv}(M_1) \cup \text{bv}(M_2) & \text{if } M = M_1 M_2 \\
\text{fv}(M') \cap \{x\} \cup \text{bv}(M') & \text{if } M = \lambda x. M' \\
\emptyset & \text{if } M = \lambda y. M' \\
\emptyset & \text{if } M = x
\end{cases}
$$

**Definition 17** (lambda term substitution). A substitution of $N$ for all free occurrences of $x$ in $M$, written $[N/x]M$, is given by

$$
[N/x]M := \begin{cases} 
([N/x]M_1) ([N/x]M_2) & \text{if } M = M_1 M_2 \\
\lambda x. M' & \text{if } M = \lambda x. M' \\
\lambda y. ([N/x]M') & \text{if } M = \lambda y. M' \text{ and } y \neq x \\
N & \text{if } M = y \\
y & \text{if } M = y \text{ and } y \neq x.
\end{cases}
$$
9.4 Rules for rewriting λ-terms

With these definitions in hand, we are now more prepared to define the rewrite rules, that is, the calculus, of lambda terms. First, we define Church’s β-reduction, a syntactic specification of function application.

**Definition 18** (beta reduction, redex). A term \( M \in \text{TERM}_\lambda \) is β-reducible, or a β-redex, when it is of the form \( M = (\lambda x . M') N \).

In most instances we say that β-redex \( (\lambda x . M') N \) reduces to the term \([N/x] M'\) and we’ll write that

\[
(\lambda x . M') N \rightarrow_\beta [N/x] M'
\]

In other words, a term with the tree

\[
\begin{array}{c}
\lambda \\
\downarrow
\end{array}
\begin{array}{c}
x \\
\downarrow
\end{array}
\begin{array}{c}
M' \\
\end{array}
\begin{array}{c}
x \\
\end{array}
\begin{array}{c}
x \\
\end{array}

\]

results in the tree

\[
\begin{array}{c}
M' \\
\end{array}
\begin{array}{c}
\downarrow
\end{array}
\begin{array}{c}
N \\
\end{array}
\begin{array}{c}
N \\
\end{array}
\begin{array}{c}
N \\
\end{array}

\]

There is a problem with our definition of β-reduction at this point. Consider the λ expression

\[
\lambda y . (\lambda x . (\lambda y . x) y).
\]

The subterm starting with \( \lambda x \) is a β-redex. When we substitute \( y \) for \( x \) according to our rule above we obtain \( \lambda y . \lambda y . y \), which is not at all an equivalent term. We end up binding the variable \( y \) inappropriately by the substitution. If instead we simply rename the variables in the first term, we obtain

\[
\lambda y . (\lambda x . (\lambda z . x) y) \rightarrow_\beta \lambda y . (\lambda z . y)
\]
more in line with what we wanted. Our problem, in general, is that we do not want to apply a substitution of \( N \) for \( x \) when it results in binding the variables that are free in \( N \). To fix this, we define a more careful substitution rule.

**Definition 19** (proper beta reduction). The term \( M = (\lambda x. M') N \) reduces to the term \([N/x]_\alpha M'\) in one step, written

\[
(\lambda x. M') N \to_\beta [N/x]_\alpha M',
\]

where \([N/x]_\alpha M'\) is given by

\[
[N/x]_\alpha = \begin{cases} 
  x & = N \\
  y & = y \\
  (M_1 M_2) & = ([N/x]_\alpha M_1) ([N/x]_\alpha M_2) \\
  \lambda x. N' & = \lambda x. N' \\
  \lambda y. N' & = \lambda y.([N/x]_\alpha N') \quad \text{if } y \notin \text{fv}(N) \\
  \lambda y. N' & = \lambda z.([N/x]_\alpha ([z/y]_\alpha N')) \quad \text{if } y \in \text{fv}(N) \text{ where } z \notin (\text{fv}(N') \cup \text{fv}(N))
\end{cases}
\]

In addition, we say that

\[
M_1 M_2 \to_\beta M'_1 M'_2 \quad \text{whenever } M_1 \to_\beta M'_1,
\]

\[
M_1 M_2 \to_\beta M'_1 M'_2 \quad \text{whenever } M_2 \to_\beta M'_2, \text{ and}
\]

\[
\lambda x. M \to_\beta \lambda x. M' \quad \text{whenever } M \to_\beta M'.
\]

Finally, we say that \( M' \) \( \beta \)-expands to \( M \) whenever \( M \to_\beta M' \), and that each term \( \beta \)-converts to the other.

**Definition 20** (alpha conversion). The term \( \lambda x. M' \) \( \alpha \)-converts to the term \( \lambda y.([y/x]_\alpha M') \) whenever \( y \notin \text{fv}(M') \), written

\[
\lambda x. M' \to_\alpha \lambda y.([y/x]_\alpha M') \quad \text{for } y \notin \text{fv}(M').
\]

In addition, we say that

\[
M_1 M_2 \to_\alpha M'_1 M'_2 \quad \text{whenever } M_1 \to_\alpha M'_1,
\]

\[
M_1 M_2 \to_\alpha M'_1 M'_2 \quad \text{whenever } M_2 \to_\alpha M'_2, \text{ and}
\]

\[
\lambda x. M \to_\alpha \lambda x. M' \quad \text{whenever } M \to_\alpha M'.
\]

Finally, we say that two lambda terms \( M, M' \) are \( \alpha \)-equivalent, and write \( M \equiv_\alpha M' \) whenever \( M = M_0 \to_\alpha M_1 \to_\alpha \cdots \to_k M = M' \).
Finally, note that if $x \notin \text{fv}(M)$ then the following term

$$\lambda x. M x$$

is a bit “baroque” in a sense. Certainly, for any term $a$, since there are no free occurrences of $x$ in $M$, the $\beta$-reduction rule gives us

$$(\lambda x. M x) a \rightarrow_{\beta} Ma.$$ 

And so, depending on our purposes below, we sometimes view the terms $\lambda x. M x$ and $M$ as equivalent.

**Definition 21** (eta-conversion). For any $x \notin \text{fv}(M)$ the terms $\lambda x. M x$ and $M$ $\eta$-convert to each other in one step. We write

$$\lambda x. M x \rightarrow_{\eta} M.$$ 

The above rewrite rules, it turns out, can be used to build a consistent equational theory for lambda terms. Namely In addition, we can show that $\beta$-reduction has some nice properties that enable a well-defined notion of computation over lambda terms. We reduce terms until no more reductions can be applied, in a manner that’s similar to evaluation of a functional language’s expressions. The lambda terms that have a reduced form — this is analogous to them having a halting evaluation — have a *unique* reduced form.

Before we establish these ideas more formally, we first motivate the lambda calculus’ use in modeling functional languages. For now we informally say that two lambda terms $M$ and $M'$ are equivalent, and write

$$M \equiv M'$$

if they are syntactically identical, or convertible to each other by a sequence of $\alpha$-conversion, $\beta$-expansion and $\eta$-reduction, and $\eta$-conversion steps.

### 9.5 MiniML primitives ⇒ $\lambda$-terms

It turns out that the lambda calculus is actually useful, in a sense. All of the constructs of MiniML can, in fact, be represented as combinators. There are no datatypes, no primitives other than function abstraction, and yet we can express any computation that we could express in MiniML.
Below, we show how to replace each of the primitive MiniML constructs with corresponding lambda terms. Here is a summary of the terms we define:

\[
\begin{align*}
I & \equiv \lambda x.x \\
K & \equiv \lambda x.\lambda y.x \\
S & \equiv \lambda x.\lambda y.\lambda z.xz(yz) \\
true & \equiv \lambda t.\lambda e.t \\
false & \equiv \lambda t.\lambda e.e \\
if & \equiv \lambda c.\lambda t.\lambda e.cte \\
succ & \equiv \lambda n.\lambda f.\lambda x.f(nfx) \\
plus & \equiv \lambda n.\lambda m.(n\ succ)m \\
eq 0 & \equiv \lambda n.(\lambda b.false)true \\
0 & \equiv \lambda f.\lambda x.x \\
1 & \equiv \lambda f.\lambda x.fx \\
2 & \equiv \lambda f.\lambda x.f(fx) \\
Y & \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))
\end{align*}
\]

**fn**
First we tackle the obvious replacement. In MiniML any expression of the form

\[
\text{fn } x \Rightarrow E
\]

can be written instead, in the lambda calculus, as a term

\[
\lambda x.E.
\]

Functional abstraction is just lambda abstraction, by definition.

**let val**
In a pure functional language, a let expression of the form

\[
\text{let val } x = E \text{ in } B \text{ end}
\]
allows you to express a computed value as a term $B$, essentially as a function of the value of the term $E$. The identifier $x$ serves as a placeholder for the computed value of $E$, and we express $B$ using $x$. The let val expression is syntactic sugar, however. A functional language does not need let val, since it provides functional abstraction.

A let expression like the above can be converted to the term

$$(\lambda x. B) E$$

since, what we want, is any occurrence of $x$ in the term $E$ to be the term $B$, exactly what we get from the $\beta$-reduction rule.

if, true, false

Let’s now convert an expression of the form

$$\text{if } C \text{ then } T \text{ else } E$$

to a lambda term. We define a curried functional version of if that takes three arguments, and will be used in this way:

$$\text{if } C T E$$

If you think about it, the $\lambda$ term if essentially gives meaning to the boolean values true and false, so let’s collectively define them as follows:

$$\text{if} := \lambda c. \lambda t. \lambda e. c t e$$

true := $\lambda t. \lambda e. t$
false := $\lambda t. \lambda e. e$

The idea behind the design of the above is quite simple. We define true and false to be curried functions of two arguments, corresponding to the term of the “then” and the term of the “else”, respectively. true behaves by ignoring the else term and yielding the then term. false behaves by ignoring the then term and yielding the else term. With these in place, the lambda term for if simply applies its condition term to the then and the else term, knowing that the condition term does the right thing.

I find this definition quite elegant. In a functional language whose only construct is functional abstraction, booleans are given meaning by their use in implementing the if expression.
natural numbers, their operations, and their comparison
We give a definition of the Church numerals, lambda terms that represent
the natural numbers. Again, we devise a functional representation that has
an elegance similar to our definition of boolean terms, above. Here are the
terms for the first four Church numerals:

\[
\begin{align*}
0 &:= \lambda f. \lambda x. x \\
1 &:= \lambda f. \lambda x. fx \\
2 &:= \lambda f. \lambda x. f(fx) \\
3 &:= \lambda f. \lambda x. f(f(fx))
\end{align*}
\]

I hope that you see the pattern— the Church numeral corresponding to a
non-negative integer \( n \) is a term of the form

\[
\begin{equation*}
\lambda f. \lambda x. f(f \ldots f(x) \ldots)
\end{equation*}
\]

or, in pseudo-lambda term syntax

\[
\lambda f. \lambda x. f^{(n)} x
\]

In general, the Church numeral for \( n \) is a curried function of two arguments
that applies its first argument to its second \( n \) times. This makes sense, I
think, and gives meaning to the \( n \)-th natural number in the lambda calculus.

In places where we use these terms, we write integers in sans serif. Also,
let \( \text{cn}(n) \) to be the Church numeral for \( n \).

It is possible to devise the primitive operations on the natural numbers
as lambda terms. For example, the successor function \( \text{succ} \) takes as its
argument a Church numeral term \( n \), and builds a church numeral term that
applies \( f \) one more time to its argument \( x \) than \( n \) does, that is,

\[
\text{succ} := \lambda n. \lambda f. \lambda x. n(fx)
\]

or, alternatively,

\[
\text{succ} := \lambda n. \lambda f. \lambda x. nf(fx)
\]

Let’s see this in action:

\[
\text{succ } 2 \equiv (\lambda n. \lambda f. \lambda x. n(fx)) 2
\]
\[
\begin{align*}
&\rightarrow_\beta \lambda f.\lambda x. (2 f (f x)) \\
&\equiv_\alpha \lambda f.\lambda x. (\lambda g.\lambda y. g (g y)) f (f x) \\
&\rightarrow_\beta \lambda f.\lambda x. (\lambda y. f (f y)) (f x) \\
&\rightarrow_\beta \lambda f.\lambda x. (\lambda y. f (f (f x))) \\
&\equiv 3
\end{align*}
\]

Have a successor function makes addition easy to express. It’s a curried function of two Church numeral arguments that applies \textit{succ} enough times (say, the first argument number of times) to the second argument:

\[
\text{plus} := \lambda x.\lambda y. x \textit{succ} y
\]

Using the \(\eta\)-conversion rule, we can write this instead as

\[
\text{plus} := \lambda n. n \text{succ}
\]

Multiplication and exponentiation can be devised, building up from these.

It is useful to relate the natural numbers to booleans by making it possible to compare Church numerals. For example, we can test whether a Church numeral corresponds to 0 by applying the following term:

\[
\text{eq0} := \lambda n. n (\lambda b. \text{false}) \text{true}
\]

If \(n\) corresponds to a positive numeral, then the constant function that yields \text{false} gets applied at least one time to \text{true}, yielding \text{false}. If 0 is instead supplied for \(n\), then that function never gets applied, yielding \text{true}. This is exactly what we wanted.

We leave defining the predecessor term, \text{pred}, as an exercise (i.e. \text{pred} should be the appropriate conversion of the MiniML expression \texttt{fn n => if n=0 then 0 else n-1}). Alternatively, the set of integers, non-negative and negative, can also be devised. This is most easily done by combining pairs, discussed below, with the Church numerals.

pairs, lists

Though our MiniML language does not include them, it is not hard to construct terms that behave like primitive data structures in the lambda calculus. For example, a small modification of the argument order of the \texttt{if} statement yields a pairing operation:

\[
\text{pair} := \lambda p_1. \lambda p_2. \lambda s. s p_1 p_2
\]
The above term takes two terms and constructs a lambda term of one argument $s$ that applies $s$ to its two arguments. Roughly speaking, pair constructs a closure containing $p_1$ and $p_2$. To extract the appropriate term from a pair term $\pi$, either its first or its second component, we simply apply the appropriate boolean term:

$$\text{first} := \lambda \pi. \pi \text{ true}$$
$$\text{second} := \lambda \pi. \pi \text{ false}$$

Let’s consider reductions steps applied to the term first (pair 1 2) below:

$$\begin{align*}
\text{first (pair 1 2)} & \equiv \text{first } ((\lambda p_1. \lambda p_2. \lambda s. s p_1 p_2) \ 1 \ 2) \\
& \rightarrow_{\beta} \text{first } ((\lambda p_2. \lambda s. s p_1 p_2) \ 2) \\
& \rightarrow_{\beta} \text{first } (\lambda s. s \ 1 \ 2) \\
& \equiv (\lambda \pi. \pi \text{ true}) (\lambda s. s \ 1 \ 2) \\
& \rightarrow_{\beta} (\lambda s. s \ 1 \ 2) \text{ true} \\
& \rightarrow_{\beta} \text{ true } 1 \ 2 \\
& \equiv (\lambda t. \lambda e. t) \ 1 \ 2 \\
& \rightarrow_{\beta} (\lambda e. \ 1) \ 2 \\
& \rightarrow_{\beta} \ 1
\end{align*}$$

the result that we expect.

Construction of lists is similar. Since we don’t have types, we can build a list with head term $H$ and tail term $T$ as the pair term pair $H \ T$. The pair extraction terms can then be used to extract the head and the tail components of such a list. Finally, we need to define a useful representation for empty list term nil. We leave this as an exercise. The representation should be one that enables an eq-nil term, a term that determines whether its list argument is empty or not.

**let fun**

Finally, we deal with MiniML expressions of the form

$$\text{let fun } f \ x = E \ \text{ in } B \ \text{ end}$$

In the case that the function $f$ being defined is not recursive, the task is easy. For example, an expression like

$$\text{let fun } s \ x = x \ + \ 1 \ \text{ in } s \ 2 \ \text{ end}$$
defines the successor function as \( s \) and then apply it. The body of \( s \) does not invoke \( s \) itself, so this whole expression can be converted to a lambda term as if it were the expression

\[
\text{let val } s = \text{fn } x \Rightarrow x + 1 \text{ in } s \ 2 \ \text{end}
\]

That conversion yields the term

\[
(\lambda s.s2)(\lambda x.\text{plus } x \ 1)
\]

Now consider instead the expression

\[
\text{let fun } f \ n = \text{if } n=0 \ \text{then } 1 \ \text{else } n \ast (f \ (n-1)) \ \text{in } f \ 4 \ \text{end}
\]

that defines, and then applies, the factorial function. This is clearly not so easy to convert because of the recursive invocation of \( f \) within its body. Performing the previous conversion method we’d use the expression

\[
\text{let val } f = \text{fn } n \Rightarrow \text{if } n=0 \ \text{then } 1 \ \text{else } n \ast (f \ (n-1)) \ \text{in } f \ 4 \ \text{end}
\]

and obtain the converted term

\[
(\lambda s.s2)(\lambda n.\text{if } \text{eq0} n \ 1 \ \text{times } n \ (f \ (\text{pred } n)))
\]

The above term is not a closed term however. The variable \( f \) is unbound within the term.

So, how does one express the invocation of a recursive call to a function within its body when its term has no bound name? In essence you cannot. When I say this, I do not mean that we cannot express recursive functions using lambda terms. Instead, to do so, the trick will be to have a name for the term that we are defining accessible within its body. We leave this for a full section, following below, because there is an elegant general concept that will be useful to introduce, namely, the notion of a fixed point of a lambda term.

Let’s first summarize our conversion strategy, including the handling of \texttt{let fun} terms, by defining the function \texttt{MLto\lambda : Term}_{\text{exp}} \rightarrow \text{Term}_{\lambda} that converts MiniML abstract syntax terms to lambda abstract syntax terms.
We write $ML(t)$ more succinctly as $[t]_\lambda$.

\[
\begin{align*}
\text{val} & := \lambda  \text{fn}Mx,eN, \\
\text{let} & := \lambda  \text{val}Mx,eN, \\
\text{fun} & := \lambda  \text{val}Mf,x,eN, \\
\text{if} & := \lambda  \text{const}M\langle c, t, e \rangle, \\
\text{const} & := \lambda  \text{val}M\langle c, e \rangle, \\
\text{plus} & := \lambda  \text{val}M\langle e_1, e_2 \rangle, \\
\text{times} & := \lambda  \text{val}M\langle e_1, e_2 \rangle, \\
\text{eq0} & := \lambda  \text{val}M\langle e_0 \rangle, \\
\text{times} & := \lambda  \text{val}M\langle e_1, e_2 \rangle, \\
\text{pred} & := \lambda  \text{val}M\langle e_0 \rangle, \\
\text{apply} & := \lambda  \text{val}M\langle f, e \rangle, \\
\text{Y} & := \lambda  \text{val}M\langle f, e \rangle.
\end{align*}
\]

In the above, the term $\text{Y}_{f,e}$ is based on the $\text{Y}$ combinator that we describe in the next section and is given by

\[
\text{Y}_{f,e} := \lambda  \text{val}M\langle g, \text{apply}(\text{val}(\lambda  \text{val}M\langle f, \text{val}(\text{val}(\langle x, e \rangle_\lambda), \langle e \rangle_\lambda)\rangle))\rangle.
\]

### 9.6 Fixed points

Let us return to our investigation of a lambda term for the factorial function. Let’s close the lambda term that was our first attempt in a way that binds the variable $f$

\[
\text{fact-maker} := \lambda  f. \lambda n. \text{if} (\text{eq0} n) 1 \text{times} n (f (\text{pred} n))
\]

Recall that the variable $f$ in the body of this lambda abstraction is a placeholder for the term that we are trying to define, namely the term for the factorial function.

Again, the real need is to know the term that $f$ stands-for while evaluating its own term’s body. With this in mind, an alternative formulation is a function that keeps passing itself to itself when a recursive application is made. In other words, it takes itself as an argument, and, when it applies itself within its body, it passes itself as its first argument, and the parameter as the second argument. Here is what I am trying to describe:

\[
\text{fact-applier} := \lambda  f. \lambda n. \text{if} (\text{eq0} n) 1 \text{times} n (f (f (\text{pred} n))
\]

Note the difference: $f$ appears twice in succession within the body.
9.6. FIXED POINTS

How do we use fact-applier? If we want to compute 4! we simply apply it to itself and 4 like so

\[
\text{fact-applier fact-applier 4}
\]

Just to make this clear, let’s look at how this term can be rewritten using the rules we’ve laid out:

\[
= (\lambda f.\lambda n.\text{if} \ (eq0 \ n) \ 1 \ (\text{times} \ n \ (f \ f \ (\text{pred} \ n)))) \ \text{fact-applier} \ 4 \\
\rightarrow_\beta (\lambda n.\text{if} \ (eq0 \ n) \ 1 \ (\text{times} \ n \ (\text{fact-applier} \ \text{fact-applier} \ (\text{pred} \ n)))) \ 4 \\
\rightarrow_\beta \text{if} \ (eq0 \ 4) \ 1 \ (\text{times} \ 4 \ (\text{fact-applier} \ \text{fact-applier} \ (\text{pred} \ 4))) \\
\rightarrow_\beta \text{times} \ 4 \ (\text{fact-applier} \ \text{fact-applier} \ 3)
\]

It should not be hard to see an inductive proof here that

\[
\text{fact-applier} \ \text{fact-applier} \ cn(n) \equiv cn(n!)
\]

for any integer \( n \geq 0 \).

Let’s step back a little and see what we’ve done here. We were looking for a lambda term, say fact, with the property that

\[
\text{fact} \ cn(n) \equiv cn(n!)
\]

Our intuition tells us that any term with this property should satisfy the term equivalence

\[
\text{fact} \equiv \lambda n.\text{if} \ (eq0 \ n) \ 1 \ (\text{times} \ n \ (\text{fact} \ (\text{pred} \ n)))
\]

or, alternatively, the term equivalence

\[
\text{fact} \equiv \text{fact-maker} \ \text{fact}
\]

such satisfying terms have significance.

**Definition 22** (fixed point). Let \( F \) and \( x \) be lambda terms with the property that

\[
x \equiv F \ x.
\]

Then the term \( x \) is called a *fixed point* of the term \( F \).
If we think of $F$ as being a function, then the application of $F$ to $x$ “returns” $x$, and thus $F$’s argument $x$ remains fixed by its application.

Suppose again more generally that we have a MiniML expression of the form

$$\text{let } \text{fun } f \ x = E \ \text{in } B \ \text{end}$$

Following what we’ve done with let we convert it to a term

$$(\lambda f. \llbracket B \rrbracket \lambda) \ f\text{-fp}$$

where $f\text{-fp}$ is a term that is a solution to the equivalence

$$f\text{-fp} \equiv f\text{-maker} \ f\text{-fp}$$

for

$$f\text{-maker} := \lambda f. \lambda x. \llbracket E \rrbracket \lambda .$$

As we did with factorial, we can solve this equivalence by defining

$$f\text{-applier} := \lambda g. (\lambda f. \lambda x. \llbracket E \rrbracket \lambda) (g g) .$$

and simply applying the term to itself. Thus, setting

$$f\text{-fp} := f\text{-applier} f\text{-applier}$$

gives us the full conversion.

Let’s make this conversion a little more formal. Note that the following holds

$$= (\lambda g. (\lambda f. \lambda x. \llbracket E \rrbracket \lambda) (g g)) (\lambda g. (\lambda f. \lambda x. \llbracket E \rrbracket \lambda) (g g))$$

$$= (\lambda g. \ f\text{-maker} (g g)) (\lambda g. \ f\text{-maker} (g g))$$

$$\rightarrow_{\beta} (\lambda h. (\lambda g. h (g g)) (\lambda g. h (g g))) \ f\text{-maker}$$

**Definition 23** (Y-combinator). The $Y$ combinator is the lambda term

$$Y := \lambda h. (\lambda g. h (g g)) (\lambda g. h (g g)) .$$

We have the following claim

**Claim 1.** For any lambda term $F$, $Y F$ is a fixed point of $F$, that is

$$Y F \equiv F (Y F)$$

holds for any $F \in \text{Term}_\lambda$. 

9.6. FIXED POINTS

Proof.

\[ YF = (\lambda h. (\lambda g. h (g g)) (\lambda g. h (g g))) F \]
\[ \rightarrow_{\beta} F (\lambda g. F (g g)) (\lambda g. F (g g)) \]
\[ \rightarrow_{\beta} F (F ((\lambda g. F (g g)) (\lambda g. F (g g)))) \]

Now, the bottom-most term on the right-hand side contains a subterm that is identical to the term on the right-hand side immediately above it, so we have that

\[ YF \equiv F (F ((\lambda g. F (g g)) (\lambda g. F (g g)))) \rightleftharpoons F (YF) \]

and hence the claim follows.

Before we end the section, let’s play with this strange combinator for a little bit. Note that by the above proof, we have that

\[ YF \equiv F (YF) \equiv F (F (YF)) \equiv F (F (F (YF))) \equiv \cdots \]

Finally, let’s play with our original factorial example:

\[ Y\text{fact-maker} \]
\[ = \text{fact-maker} (Y\text{fact-maker}) 4 \]
\[ \equiv (\lambda f. \lambda n. \text{if eq0 n} 1 \left( \times n \left( f \left( \text{pred} n \right) \right) \right)) \left( Y\text{fact-maker} \right) 4 \]
\[ \rightarrow_{\beta} (\lambda n. \text{if eq0 n} 1 \left( \times n \left( \left( Y\text{fact-maker} \right) \left( \text{pred} n \right) \right) \right)) 4 \]
\[ \rightarrow_{\beta} \text{if eq0 4} 1 \left( \times 4 \left( \left( Y\text{fact-maker} \right) \left( \text{pred} 4 \right) \right) \right) \]
\[ \equiv \times 4 \left( \left( Y\text{fact-maker} \right) 3 \right) . \]

So we’ve managed to express the factorial in, at first glance, a most unnatural fashion, but also somewhat succinctly! Well, at least there’s none of that nonsense! We end with a shout out to the Y combinator. For an appropriate definition of the “integer whose Church numeral has the lambda term of” function, and for any \( n \geq 0 \) we exclaim

\[ cn^{-1}(Y\text{fact-maker} cn(n)) = n! \]
9.7 Complete combinators

It is reasonable to be suspicious of the game that we were playing in Sections ?? and ??, above. One would probably never write programs that look like the combinators we strung together to mimic ML primitives.

It is certainly the case that the study of the lambda calculus has had some practical use. Namely, when building tools to analyze programs, it can be extremely useful to boil a program into a representation that’s difficult for a human to read, but easy for a programming tool to analyze and to execute efficiently. In addition, if we’d like to prove correctness or assert mathematical guarantees about a programming tool’s program analyses, transformations, and execution strategies, it’s useful to have a minimal representation to prove theorems.

Finally, there is something to “combinator programming” that requires a rethinking of algorithm expression that I feel is ultimately useful. A number of programming languages, namely Haskell, encourage a combinator style of programming. The value in writing programs this way, and the value of such languages, is that they are making an impact on current research in program analysis tools. I expect this research to influence mainstream languages and their compilers shortly.

In this chapter we continue a similar program. We give examples of combinators “alphabets” that are complete in that they enable us to express any lambda term entirely as parenthesized sequences of the combinators.

Recall the definition of the combinators $S$, $K$, and $I$

$S := \lambda x.\lambda y.\lambda z.x (y z)$

$K := \lambda x.\lambda y.x$

$I := \lambda x.x$

It turns out that any lambda term can be rewritten to an equivalent term that only uses the combinators $S, K,$ and $I,$ particularly no lambda abstractions.

Let’s define these notions a little more formally:

**Definition 24.** (combinator alphabet) A combinator alphabet is a pair $(\Sigma, c)$ where $c : \Sigma \to \text{TERM}_\lambda$, and $\text{FV}(c(s)) = \emptyset$ for any $s \in \Sigma$.

**Definition 25.** (combinator terms) For combinator alphabet $\Sigma$, we let Let $\text{TERM}_\Sigma$ be the terms constructed from the abstract syntax corresponding to
the grammar

\[
\begin{align*}
E & \rightarrow EE \\
E & \rightarrow (E) \\
E & \rightarrow x \\
E & \rightarrow \text{any symbol of } \Sigma \\
x & \rightarrow \text{any valid variable identifier}
\end{align*}
\]

**Definition 26.** (complete combinator alphabet) A combinator alphabet \((\Sigma, c)\) is complete if there exists a function \(\text{conv} : \text{TERM}_\lambda \rightarrow \text{TERM}_\chi\) where for any \(t \in \text{TERM}_\lambda\)

\[
t \equiv [c(s_1)/s_1][c(s_2)/s_2] \cdots [c(s_k)/s_k]\text{conv}(t)
\]

for \(\Sigma = \{s_1, s_2, \ldots, s_k\}\).

Let \(\chi = \{S, K, I\}\) be our set of combinator symbols with the obvious correspondence with \(\lambda\)-terms.

We let \(\chi = \{S, K, I\}\) be our alphabet of combinators. Consider the following grammar for combinator terms that use only application, free variables, and these combinators in their expression:

\[
\begin{align*}
E & \rightarrow EE \\
E & \rightarrow (E) \\
E & \rightarrow x \\
E & \rightarrow \chi \\
x & \rightarrow \text{any valid variable identifier} \\
\chi & \rightarrow S \mid K \mid I
\end{align*}
\]

It turns out that any It has the obvious abstract syntax that enables us to construct what I will call \(\text{TERM}_\chi\), the set of all \(\chi\)-terms.