Chapter 3

Lambda calculus as a programming language

One of the central questions studied by logicians in the 1930s was the *Entscheidungsproblem* or 'decision problem'. This asked whether there was some systematic, mechanical procedure for deciding validity in first order logic. If it turned out that there was, it would have fundamental philosophical, and perhaps practical, significance. It would mean that a huge variety of complicated mathematical problems could in principle be solved purely by following some single fixed method — we would now say *algorithm* — no additional creativity would be required.

As it stands the question is hard to answer, because we need to define in mathematical terms what it means for there to be a 'systematic, mechanical' procedure for something. Perhaps Turing (1936) gave the best analysis, by arguing that the mechanical procedures are those that could in principle be performed by a reasonably intelligent clerk with no knowledge of the underlying subject matter. He abstracted from the behaviour of such a clerk and arrived at the famous notion of a 'Turing machine'. Despite the fact that it arose by considering a *human* 'computer', and was purely a mathematical abstraction, we now see a Turing machine as a very simple computer. Despite being simple, a Turing machine is capable of performing any calculation that can be done by a physical computer.[1] A model of computation of equal power to a Turing machine is said to be *Turing complete*.

At about the same time, there were several other proposed definitions of 'mechanical procedure'. Most of these turned out to be equivalent in power to Turing machines. In particular, although it originally arose as a foundation for mathematics, lambda calculus can also be seen as a programming language, to be executed by performing β-conversions. In fact, Church proposed before Turing that the set of operations that can be expressed in lambda calculus formalizes the

[1] Actually more, since it neglects the necessary finite limit to the computer's store. Arguably any existing computer is actually a finite state machine, but the assumption of unbounded memory seems a more fruitful abstraction.

intuitive idea of a mechanical procedure, a postulate known as *Church's thesis*. Church (1936) showed that if this was accepted, the *Entscheidungsproblem* is unsolvable. Turing later showed that the functions definable in lambda calculus are precisely those computable by a Turing machine, which lent more credibility to Church's claims.

To a modern programmer, Turing machine programs are just about recognizable as a very primitive kind of machine code. Indeed, it seems that Turing machines, and in particular Turing's construction of a 'universal machine',[2] were a key influence in the development of modern stored program computers, though the extent and nature of this influence is still a matter for controversy (Robinson 1994). It is remarkable that several of the other proposals for a definition of 'mechanical procedure', often made before the advent of electronic computers, correspond closely to real programming methods. For example, Markov algorithms, the standard approach to computability in the (former) Soviet Union, can be seen as underlying the programming language SNOBOL. In what follows, we are concerned with the influence of lambda calculus on functional programming languages.

LISP, the second oldest high level language (FORTRAN being the oldest) took a few ideas from lambda calculus, in particular the notation '(LAMBDA $\cdots$)' for anonymous functions. However it was not really based on lambda calculus at all. In fact the early versions, and some current dialects, use a system of *dynamic binding* that does not correspond to lambda calculus. (We will discuss this later.) Moreover there was no real support for higher order functions, while there was considerable use of imperative features. Nevertheless LISP deserves to be considered as the first functional programming language, and pioneered many of the associated techniques such as automatic storage allocation and garbage collection.

The influence of lambda calculus really took off with the work of Landin and Strachey in the 60s. In particular, Landin showed how many features of current (imperative) languages could usefully be analyzed in terms of lambda calculus, e.g. the notion of variable scoping in Algol 60. Landin (1966) went on to propose the use of lambda calculus as a core for programming languages, and invented a functional language ISWIM ('If you See What I Mean'). This was widely influential and spawned many real languages.

ML started life as the metalanguage (hence the name ML) of a theorem proving system called Edinburgh LCF (Gordon, Milner, and Wadsworth 1979). That is, it was intended as a language in which to write algorithms for proving theorems in a formal deductive calculus. It was strongly influenced by ISWIM but added an innovative polymorphic type system including abstract types, as well as a system of *exceptions* to deal with error conditions. These facilities were dictated

[2] A single Turing machine program capable of simulating any other — we would now regard it as an *interpreter*. You will learn about this in the course on Computation Theory.

by the application at hand, and this policy resulted in a coherent and focused design. This narrowness of focus is typical of successful languages (C is another good example) and contrasts sharply with the failure of committee-designed languages like Algol 60 which have served more as a source of important ideas than as popular practical tools. We will have more to say about ML later. Let us now see how *pure lambda calculus* can be used as a programming language.

3.1 Representing data in lambda calculus

Programs need data to work on, so we start by fixing lambda expressions to encode data. Furthermore, we define a few basic operations on this data. In many cases, we show how a string of human-readable syntax s can be translated directly into a lambda expression s'. This procedure is known as 'syntactic sugaring' — it makes the bitter pill of pure lambda notation easier to digest. We write such definitions as:

$$s \triangleq s'$$

This notation means '$s = s'$ by definition'; one also commonly sees this written as '$s =_{def} s'$'. If preferred, we could always regard these as defining a single constant denoting that operation, which is then applied to arguments in the usual lambda calculus style, the surface notation being irrelevant. For example we can imagine 'if E then E_1 else E_2' as 'COND E E_1 E_2' for some constant COND. In such cases, any variables on the left of the definition need to be abstracted over, e.g. instead of

$$\text{fst}\ p \triangleq p\ \text{true}$$

(see below) we could write

$$\text{fst} \triangleq \lambda p.\, p\ \text{true}$$

3.1.1 Truth values and the conditional

We can use any unequal lambda expressions to stand for 'true' and 'false', but the following work particularly smoothly:

$$\begin{aligned} \text{true} &\triangleq \lambda x\ y.\, x \\ \text{false} &\triangleq \lambda x\ y.\, y \end{aligned}$$

Given those definitions, we can easily define a conditional expression just like C's '`?:`' construct. Note that this is a conditional *expression* not a conditional *command* (this notion makes no sense in our context) and therefore the 'else' arm is compulsory:

$$\text{if } E \text{ then } E_1 \text{ else } E_2 \triangleq E\ E_1\ E_2$$

indeed we have:

$$\begin{aligned}\text{if true then } E_1 \text{ else } E_2 &= \text{true } E_1\ E_2\\ &= (\lambda x\ y.\ x)\ E_1\ E_2\\ &= E_1\end{aligned}$$

and

$$\begin{aligned}\text{if false then } E_1 \text{ else } E_2 &= \text{false } E_1\ E_2\\ &= (\lambda x\ y.\ y)\ E_1\ E_2\\ &= E_2\end{aligned}$$

Once we have the conditional, it is easy to define all the usual logical operations:

$$\begin{aligned}\text{not } p &\triangleq \text{if } p \text{ then false else true}\\ p \text{ and } q &\triangleq \text{if } p \text{ then } q \text{ else false}\\ p \text{ or } q &\triangleq \text{if } p \text{ then true else } q\end{aligned}$$

3.1.2 Pairs and tuples

We can represent ordered pairs as follows:

$$(E_1, E_2) \triangleq \lambda f.\ f\ E_1\ E_2$$

The parentheses are not obligatory, but we often use them for the sake of familiarity or to enforce associations. In fact we simply regard the comma as an infix operator like $+$. Given the above definition, the corresponding destructors for pairs can be defined as:

$$\begin{aligned}\text{fst } p &\triangleq p \text{ true}\\ \text{snd } p &\triangleq p \text{ false}\end{aligned}$$

It is easy to see that these work as required:

$$\begin{aligned}
\text{fst } (p,q) &= (p,q) \text{ true} \\
&= (\lambda f.\ f\ p\ q) \text{ true} \\
&= \text{true } p\ q \\
&= (\lambda x\ y.\ x)\ p\ q \\
&= p
\end{aligned}$$

and

$$\begin{aligned}
\text{snd } (p,q) &= (p,q) \text{ false} \\
&= (\lambda f.\ f\ p\ q) \text{ false} \\
&= \text{false } p\ q \\
&= (\lambda x\ y.\ y)\ p\ q \\
&= q
\end{aligned}$$

We can build up triples, quadruples, quintuples, indeed arbitrary n-tuples, by iterating the pairing construct:

$$(E_1, E_2, \ldots, E_n) = (E_1, (E_2, \ldots E_n))$$

We need simply say that the infix comma operator is right-associative, and can understand this without any other conventions. For example:

$$\begin{aligned}
(p,q,r,s) &= (p,(q,(r,s))) \\
&= \lambda f.\ f\ p\ (q,(r,s)) \\
&= \lambda f.\ f\ p\ (\lambda f.\ f\ q\ (r,s)) \\
&= \lambda f.\ f\ p\ (\lambda f.\ f\ q\ (\lambda f.\ f\ r\ s)) \\
&= \lambda f.\ f\ p\ (\lambda g.\ g\ q\ (\lambda h.\ h\ r\ s))
\end{aligned}$$

where in the last line we have performed an alpha-conversion for the sake of clarity. Although tuples are built up in a flat manner, it is easy to create arbitrary finitely-branching tree structures by using tupling repeatedly. Finally, if one prefers conventional functions over Cartesian products to our 'curried' functions, one can convert between the two using the following:

$$\begin{aligned}
\text{CURRY } f &\triangleq \lambda x\ y.\ f(x,y) \\
\text{UNCURRY } g &\triangleq \lambda p.\ g\ (\text{fst } p)\ (\text{snd } p)
\end{aligned}$$

These special operations for pairs can easily be generalized to arbitrary n-tuples. For example, we can define a selector function to get the i^{th} component of a flat tuple p. We will write this operation as $(p)_i$ and define $(p)_1 = \text{fst}\ p$ and the others by $(p)_i = \text{fst}\ (\text{snd}^{i-1}\ p)$. Likewise we can generalize CURRY and UNCURRY:

$$\begin{aligned} \text{CURRY}_n\ f &\triangleq \lambda x_1\ \cdots\ x_n.\, f(x_1, \ldots, x_n) \\ \text{UNCURRY}_n\ g &\triangleq \lambda p.\, g\ (p)_1\ \cdots\ (p)_n \end{aligned}$$

We can now write $\lambda(x_1, \ldots, x_n).\, t$ as an abbreviation for:

$$\text{UNCURRY}_n\ (\lambda x_1\ \cdots\ x_n.\, t)$$

giving a natural notation for functions over Cartesian products.

3.1.3 The natural numbers

We will represent each natural number n as follows:[3]

$$n \triangleq \lambda f\ x.\, f^n\ x$$

that is, $0 = \lambda f\ x.\, x$, $1 = \lambda f\ x.\, f\ x$, $2 = \lambda f\ x.\, f\ (f\ x)$ etc. These representations are known as *Church numerals*, though the basic idea was used earlier by Wittgenstein (1922).[4] They are not a very efficient representation, amounting essentially to counting in unary, $1, 11, 111, 1111, 11111, 111111, \ldots$. There are, from an efficiency point of view, much better ways of representing numbers, e.g. tuples of trues and falses as binary expansions. But at the moment we are only interested in computability 'in principle' and Church numerals have various nice formal properties. For example, it is easy to give lambda expressions for the common arithmetic operators such as the successor operator which adds one to its argument:

$$\text{SUC} \triangleq \lambda n\ f\ x.\, n\ f\ (f\ x)$$

Indeed

$$\begin{aligned} \text{SUC}\ n &= (\lambda n\ f\ x.\, n\ f\ (f\ x))(\lambda f\ x.\, f^n\ x) \\ &= \lambda f\ x.\, (\lambda f\ x.\, f^n\ x) f\ (f\ x) \\ &= \lambda f\ x.\, (\lambda x.\, f^n\ x)(f\ x) \end{aligned}$$

[3] The n in $f^n\ x$ is just a meta-notation, and does not mean that there is anything circular about our definition.

[4] '6.021 A number is the exponent of an operation'.

$$
\begin{aligned}
&= \lambda f\ x.\ f^n\ (f\ x) \\
&= \lambda f\ x.\ f^{n+1}\ x \\
&= n+1
\end{aligned}
$$

Similarly it is easy to test a numeral to see if it is zero:

$$\text{ISZERO}\ n \triangleq n\ (\lambda x.\ \text{false})\ \text{true}$$

since:

$$\text{ISZERO}\ 0 = (\lambda f\ x.\ x)(\lambda x.\ \text{false})\ \text{true} = \text{true}$$

and

$$
\begin{aligned}
\text{ISZERO}\ (n+1) &= (\lambda f\ x.\ f^{n+1} x)(\lambda x.\ \text{false})\text{true} \\
&= (\lambda x.\ \text{false})^{n+1}\ \text{true} \\
&= (\lambda x.\ \text{false})((\lambda x.\ \text{false})^n\ \text{true}) \\
&= \text{false}
\end{aligned}
$$

The following perform addition and multiplication on Church numerals:

$$
\begin{aligned}
m+n &\triangleq \lambda f\ x.\ m\ f\ (n\ f\ x) \\
m*n &\triangleq \lambda f\ x.\ m\ (n\ f)\ x
\end{aligned}
$$

Indeed:

$$
\begin{aligned}
m+n &= \lambda f\ x.\ m\ f\ (n\ f\ x) \\
&= \lambda f\ x.\ (\lambda f\ x.\ f^m\ x)\ f\ (n\ f\ x) \\
&= \lambda f\ x.\ (\lambda x.\ f^m\ x)\ (n\ f\ x) \\
&= \lambda f\ x.\ f^m\ (n\ f\ x) \\
&= \lambda f\ x.\ f^m\ ((\lambda f\ x.\ f^n\ x)\ f\ x) \\
&= \lambda f\ x.\ f^m\ ((\lambda x.\ f^n\ x)\ x) \\
&= \lambda f\ x.\ f^m\ (f^n\ x) \\
&= \lambda f\ x.\ f^{m+n} x
\end{aligned}
$$

and:

$$
\begin{aligned}
m*n &= \lambda f\ x.\ m\ (n\ f)\ x \\
&= \lambda f\ x.\ (\lambda f\ x.\ f^m\ x)\ (n\ f)\ x
\end{aligned}
$$

$$
\begin{aligned}
&= \lambda f\ x.\ (\lambda x.\ (n\ f)^m\ x)\ x \\
&= \lambda f\ x.\ (n\ f)^m\ x \\
&= \lambda f\ x.\ ((\lambda f\ x.\ f^n\ x)\ f)^m\ x \\
&= \lambda f\ x.\ ((\lambda x.\ f^n\ x)^m\ x \\
&= \lambda f\ x.\ (f^n)^m\ x \\
&= \lambda f\ x.\ f^{mn} x
\end{aligned}
$$

Although those operations on natural numbers were fairly easy, a 'predecessor' function is much harder. What is required is a a lambda expression PRE so that $PRE\ 0 = 0$ and $PRE\ (n+1) = n$. Finding such an expression was the first piece of mathematical research by the logician Kleene (1935). His trick was, given $\lambda f\ x.\, f^n\ x$, to 'throw away' one of the applications of f. The first step is to define a function 'PREFN' that operates on pairs such that:

$$\text{PREFN}\ f\ (\text{true}, x) = (\text{false}, x)$$

and

$$\text{PREFN}\ f\ (\text{false}, x) = (\text{false}, f\ x)$$

Given this function, then $(\text{PREFN}\ f)^{n+1}(\text{true}, x) = (\text{false}, f^n\ x)$, which is enough to let us define a predecessor function without much difficulty. Here is a suitable definition of 'PREFN':

$$\text{PREFN} \triangleq \lambda f\ p.\ (\text{false}, \text{if fst } p \text{ then snd } p \text{ else } f(\text{snd } p)$$

Now we define:

$$\text{PRE}\ n \triangleq \lambda f\ x.\ \text{snd}(n\ (\text{PREFN}\ f)\ (\text{true}, x))$$

It is left as an exercise to the reader to see that this works correctly.

3.2 Recursive functions

Being able to define functions by recursion is a central feature of functional programming: it is the only general way of performing something comparable to iteration. At first sight, it appears that we have no way of doing this in lambda calculus. Indeed, it would seem that the *naming* of the function is a vital part of making a recursive definition, since otherwise, how can we refer to it on the right hand side of the definition without descending into infinite regress? Rather surprisingly, it can be done, although as with the existence of a predecessor function, this fact was only discovered after considerable effort.

The key is the existence of so-called *fixed point combinators*. A closed lambda term Y is said to be a fixed point (or fixpoint) combinator when for all lambda terms f, we have $f(Y\ f) = Y\ f$. That is, a fixed point combinator, given any term f as an argument, returns a fixed point for f, i.e. a term x such that $f(x) = x$. The first such combinator to be found (by Curry) is usually called Y. It can be motivated by recalling the Russell paradox, and for this reason is often called the 'paradoxical combinator'. We defined:

$$R = \lambda x.\ \neg(x\ x)$$

and found that:

$$R\ R = \neg(R\ R)$$

That is, $R\ R$ is a fixed point of the negation operator. So in order to get a general fixed point combinator, all we need to do is generalize $\neg$ into whatever function is given it as argument. Therefore we set:

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

It is a simple matter to show that it works:

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

Though valid from a mathematical point of view, this is not so attractive from a programming point of view since the above was only valid for lambda equality, not reduction (in the last line we performed a *backwards* beta conversion). For this reason the following alternative, due to Turing, may be preferred:

$$T \triangleq (\lambda x\ y.\ y\ (x\ x\ y))\ (\lambda x\ y.\ y\ (x\ x\ y))$$

(The proof that $T\ f \longrightarrow f(T\ f)$ is left as an exercise.) However it does no harm if we simply allow Yf to be beta-reduced throughout the reduction sequence for a recursively defined function. Now let us see how to use a fixed point combinator (say Y) to implement recursive functions. We will take the factorial function as an example. We want to define a function fact with:

$$\text{fact}(n) = \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n)$$

The first step is to transform this into the following equivalent:

$$\text{fact} = \lambda n.\ \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n)$$

which in its turn is equivalent to:

$$\text{fact} = (\lambda f\ n.\ \text{if ISZERO}\ n\ \text{then}\ 1\ \text{else}\ n * f(\text{PRE}\ n))\ \text{fact}$$

This exhibits fact as the fixed point of a function F, namely:

$$F = \lambda f\ n.\ \text{if ISZERO}\ n\ \text{then}\ 1\ \text{else}\ n * f(\text{PRE}\ n)$$

Consequently we merely need to set fact $= Y\ F$. Similar techniques work for mutually recursive functions, i.e. a set of functions whose definitions are mutually dependent. A definition such as:

$$\begin{array}{rcl} f_1 & = & F_1\ f_1\ \cdots\ f_n \\ f_2 & = & F_2\ f_1\ \cdots\ f_n \\ \ldots & = & \ldots \\ f_n & = & F_n\ f_1\ \cdots\ f_n \end{array}$$

can be transformed, using tuples, into a single equation:

$$(f_1, f_2, \ldots, f_n) = (F_1\ f_1\ \cdots\ f_n, F_2\ f_1\ \cdots\ f_n, \ldots, F_n\ f_1\ \cdots\ f_n)$$

If we now write $t = (f_1, f_2, \ldots, f_n)$, then each of the functions on the right can be recovered by applying the appropriate selector to t: we have $f_i = (t)_i$. After abstracting out t, this gives an equation in the canonical form $t = F\ t$ which can be solved by $t = Y\ F$. Now the individual functions can be recovered from t by the same method of selecting components of the tuple.

3.3 Let expressions

We've touted the ability to write anonymous functions as one of the merits of lambda calculus, and shown that names are not necessary to define recursive functions. Nevertheless it is often useful to be able to give names to expressions, to avoid tedious repetition of large terms. A simple form of naming can be supported as another syntactic sugar on top of pure lambda calculus:

$$\text{let}\ x = s\ \text{in}\ t \triangleq (\lambda x.\ t)\ s$$

For example, as we would expect:

$$(\text{let}\ z = 2 + 3\ \text{in}\ z + z) = (\lambda z.\ z + z)\ (2 + 3) = (2 + 3) + (2 + 3)$$

We can achieve the binding of multiple names to expressions either in a serial or parallel fashion. For the first, we will simply iterate the above construct. For the latter, we will write multiple bindings separated by 'and':

$$\text{let } x_1 = s_1 \text{ and } \cdots \text{ and } x_n = s_n \text{ in } t$$

and regard it as syntactic sugar for:

$$(\lambda(x_1, \ldots, x_n).\, t)\ (s_1, \ldots, s_n)$$

To illustrate the distinction between serial and parallel binding, consider:

$$\text{let } x = 1 \text{ in let } x = 2 \text{ in let } y = x \text{ in } x + y$$

which should yield 4, while:

$$\text{let } x = 1 \text{ in let } x = 2 \text{ and } y = x \text{ in } x + y$$

should yield 3. We will also allow arguments to names bound by '`let`'; again this can be regarded as syntactic sugar, with $f\ x_1\ \cdots\ x_n = t$ meaning $f = \lambda x_1\ \cdots\ x_n.\, t$. Instead of a prefix 'let $x = s$ in t', we will allow a postfix version, which is sometimes more readable:

$$t \text{ where } x = s$$

For example, we might write '$y < y^2$ where $y = 1 + x$'. Normally, the 'let' and 'where' constructs will be interpreted as above, in a non-recursive fashion. For example:

$$\text{let } x = x - 1 \text{ in } \cdots$$

binds x to one less than whatever value it is bound to outside, rather than trying to bind x to the fixed point of $x = x - 1$.[5] However where a recursive interpretation is required, we add the word 'rec', writing 'let rec' or 'where rec', e.g.

$$\text{let rec fact}(n) = \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n)$$

This can be regarded as a shorthand for 'let fact $= Y\ F$' where

$$F = \lambda f\ n.\, \text{if ISZERO } n \text{ then } 1 \text{ else } n * f(\text{PRE } n)$$

as explained above.

[5]There is such a fixed point, but the lambda term involved has no normal form, so is in a sense 'undefined'. The semantics of nonterminating terms is subtle, and we will not consider it at length here. Actually, the salient question is not whether a term has a normal form but whether it has a so-called *head normal form* — see Barendregt (1984) and also Abramsky (1990).

3.4 Steps towards a real programming language

We have established a fairly extensive system of 'syntactic sugar' to support a human-readable syntax on top of pure lambda calculus. It is remarkable that this device is enough to bring us to the point where we can write the definition of the factorial function almost in the way we can in ML. In what sense, though, is the lambda calculus augmented with these notational conventions really a programming language?

Ultimately, a program is just a single expression. However, given the use of `let` to bind various important subexpressions to names, it is natural instead to view the program as a set of *definitions* of auxiliary functions, followed finally by the expression itself, e.g:

$$
\begin{array}{l}
\text{let rec fact}(n) = \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n) \text{ in} \\
\cdots \\
fact(6)
\end{array}
$$

We can read these definitions of auxiliary functions as equations in the mathematical sense. Understood in this way, they do not give any explicit instructions on how to evaluate expressions, or even in which direction the equations are to be used. For this reason, functional programming is often referred to as a *declarative* programming method, as also is logic programming (e.g. PROLOG).[6] The program does not incorporate explicit instructions, merely declares certain properties of the relevant notions, and leaves the rest to the machine.

At the same time, the program is useless, or at least ambiguous, unless the machine somehow reacts sensibly to it. Therefore it must be understood that the program, albeit from a superficial point of view purely declarative, is to be executed in a certain way. In fact, the expression is evaluated by expanding all instances of defined notions (i.e. reading the equations from left to right), and then repeatedly performing β-conversions. That is, although there seems to be no procedural information in the program, *a particular execution strategy is understood*. In a sense then, the term 'declarative' is largely a matter of human psychology.

Moreover, there must be a definite convention for the reduction strategy to use, for we know that different choices of β-redex can lead to different behaviour with respect to termination. Consequently, it is only when we specify this that we really get a definite programming language. We will see in later chapters how different functional languages make different choices here. But first we pause to consider the incorporation of types.

[6] Apparently Landin preferred 'denotative' to 'declarative'.

3.5 Further reading

Many of the standard texts already cited include details about the matters discussed here. In particular Gordon (1988) gives a clear proof that lambda calculus is Turing-complete, and that the lambda calculus analog of the 'halting problem' (whether a term has a normal form) is unsolvable. The influence of lambda calculus on real programming languages and the evolution of functional languages are discussed by Hudak (1989).

Exercises

1. Justify 'generalized β-conversion', i.e. prove that:

$$(\lambda(x_1, \ldots, x_n).\ t[x_1, \ldots, x_n])(t_1, \ldots, t_n) = t[t_1, \ldots, t_n]$$

2. Define $f \circ g \stackrel{\triangle}{=} \lambda x.\ f(g\ x)$, and recall that $I = \lambda x.\ x$. Prove that CURRY $\circ$ UNCURRY $= I$. Is it also true that UNCURRY $\circ$ CURRY $= I$?.

3. What arithmetical operation does $\lambda n\ f\ x.\ f(n\ f\ x)$ perform on Church numerals? What about $\lambda m\ n.\ n\ m$?

4. Prove that the following (due to Klop) is a fixed point combinator:

$$££££££££££££££££££££££££££$$

where:

$$£ \stackrel{\triangle}{=} \lambda abcdefghijklmnopqstuvwxyzr.\ r(thisisafixedpointcombinator)$$

5. Make a recursive definition of natural number subtraction.

6. Consider the following representation of lists, due to Mairson (1991):

$$\begin{array}{rcl}
\text{nil} & = & \lambda c\ n.\ n \\
\text{cons} & = & \lambda x\ l\ c\ n.\ c\ x\ (l\ c\ n) \\
\text{head} & = & \lambda l.\ l(\lambda x\ y.\ x)\ \text{nil} \\
\text{tail} & = & \lambda l.\ \text{snd}\ (l(\lambda x\ p.\ (\text{cons}\ x\ (\text{fst}\ p), \text{fst}\ p))\ (\text{nil}, \text{nil})) \\
\text{append} & = & \lambda l_1\ l_2.\ l_1\ \text{cons}\ l_2 \\
\text{append_lists} & = & \lambda L.\ L\ \text{append}\ \text{nil}
\end{array}$$

$$
\begin{aligned}
\text{map} &= \lambda f\ l.\ l\ (\lambda x.\ \text{cons}\ (f\ x))\ \text{nil} \\
\text{length} &= \lambda l.\ l(\lambda x.\ \text{SUC})0 \\
\text{tack} &= \lambda x\ l.\ l\ \text{cons}\ (\text{cons}\ x\ \text{nil}) \\
\text{reverse} &= \lambda l.\ l\ \text{tack}\ \text{nil} \\
\text{filter} &= \lambda l\ test.\ l\ (\lambda x.\ (test\ x)(\text{cons}\ x)(\lambda y.\ y))\ \text{nil}
\end{aligned}
$$

Why does Mairson call them 'Church lists'? What do all these functions do?