Title:	*Introduction to Functional Programming*
Lecturer:	**Mike Gordon** (`http://www.cl.cam.ac.uk/users/mjcg/`)
Class:	**Computer Science Tripos, Part II(General) & Diploma**
Term:	**Lent term 1996**
First Lecture:	**Friday January 19 1996 at 12am**
Location:	**Heycock Lecture Room**
Duration:	**Twelve lectures (M. W. F. 12)**

Preface

This course aims to teach both the theory and practice of functional programming. The theory consists of the λ-calculus and the practice will be illustrated using the programming language Standard ML.

The field of Functional Programming splits into those who prefer 'lazy' languages like Haskell and those who prefer 'strict' languages like ML. The practical parts of this course almost exclusively emphasise the latter, but the material on the λ-calculus underlies both approaches.

The chapters on the λ-calculus have been largely condensed from Part II of the book:

> M.J.C. Gordon, *Programming Language Theory and its Implementation*, Prentice Hall International Series in Computer Science, 1988 (currently out of print).

The introduction to ML in Chapter 4 started life as part of:

> Gordon, M.J.C., Milner, A.J.R.G. and Wadsworth, C.P., *Edinburgh LCF: a mechanized logic of computation*, Springer Lecture Notes in Computer Science, Springer-Verlag, 1979.

The ML parts of this were updated substantially in the technical report:

> G. Cousineau, M. Gordon, G. Huet, R. Milner, L. Paulson, and C. Wadsworth, *The ML handbook*, INRIA (1986).

I translated the introduction of this report into Standard ML and added some new material to get Chapter 4. The case studies were written by me at great speed, and so are bound to contain numerous mistakes! They aim to show how ML-based functional programming can be used in practice.

The following people have contributed in various ways to the material cited above or to these notes: Graham Birtwistle, Shiu Kai Chin, Avra Cohn, Jan van Eijck, Mike Fourman, Elsa Gunter, Peter Hancock, Martin Hyland, Tom Melham, Allan C. Milne, Nicholas Ourusoff, David Shepherd and Roger Stokes.

Contents

Chapter 1

Introduction to the λ-calculus

The λ-calculus (or lambda-calculus) is a theory of functions that was originally developed by the logician Alonzo Church as a foundation for mathematics. This work was done in the 1930s, several years before digital computers were invented. A little earlier (in the 1920s) Moses Schönfinkel developed another theory of functions based on what are now called 'combinators'. In the 1930s, Haskell Curry rediscovered and extended Schönfinkel's theory and showed that it was equivalent to the λ-calculus. About this time Kleene showed that the λ-calculus was a universal computing system; it was one of the first such systems to be rigorously analysed. In the 1950s John McCarthy was inspired by the λ-calculus to invent the programming language LISP. In the early 1960s Peter Landin showed how the meaning of imperative programming languages could be specified by translating them into the λ-calculus. He also invented an influential prototype programming language called ISWIM [24]. This introduced the main notations of functional programming and influenced the design of both functional and imperative languages. Building on this work, Christopher Strachey laid the foundations for the important area of denotational semantics [13, 33]. Technical questions concerning Strachey's work inspired the mathematical logician Dana Scott to invent the theory of domains, which is now one of the most important parts of theoretical computer science. During the 1970s Peter Henderson and Jim Morris took up Landin's work and wrote a number of influential papers arguing that functional programming had important advantages for software engineering [17, 16]. At about the same time David Turner proposed that Schönfinkel and Curry's combinators could be used as the machine code of computers for executing functional programming languages. Such computers could exploit mathematical properties of the λ-calculus for the parallel evaluation of programs. During the 1980s several research groups took up Henderson's and Turner's ideas and started working on making functional programming practical by designing special architectures to support it, some of them with many processors.

We thus see that an obscure branch of mathematical logic underlies important developments in programming language theory, such as:

(i) The study of fundamental questions of computation.

(ii) The design of programming languages.

(iii) The semantics of programming languages.

(iv) The architecture of computers.

1.1 Syntax and semantics of the λ-calculus

The λ-calculus is a notation for defining functions. The expressions of the notation are called *λ-expressions* and each such expression denotes a function. It will be seen later how functions can be used to represent a wide variety of data and data-structures including numbers, pairs, lists etc. For example, it will be demonstrated how an arbitrary pair of numbers (x, y) can be represented as a λ-expression. As a notational convention, mnemonic names are assigned in **bold** or <u>underlined</u> to particular λ-expressions; for example $\underline{1}$ is the λ-expression (defined in Section 2.3) which is used to represent the number one.

There are just three kinds of λ-expressions:

(i) **Variables:** x, y, z etc. The functions denoted by variables are determined by what the variables are bound to in the *environment*. Binding is done by abstractions (see 3 below). We use V, V_1, V_2 etc. for arbitrary variables.

(ii) **Function applications** or **combinations:** if E_1 and E_2 are λ-expressions, then so is $(E_1\ E_2)$; it denotes the result of applying the function denoted by E_1 to the function denoted by E_2. E_1 is called the *rator* (from 'operator') and E_2 is called the *rand* (from 'operand'). For example, if $(\underline{m}, \underline{n})$ denotes a function representing the pair of numbers m and n (see Section 2.2) and **sum** denotes the addition function[1] λ-calculus (see Section 2.5), then the application $(\mathbf{sum}(\underline{m}, \underline{n}))$ denotes $\underline{m+n}$.

(iii) **Abstractions:** if V is a variable and E is a λ-expression, then $\lambda V.\ E$ is an abstraction with *bound variable* V and *body* E. Such an abstraction denotes the function that takes an argument a and returns as result the function denoted by E in an environment in which the bound variable V denotes a. More specifically, the abstraction $\lambda V.\ E$ denotes a function which takes an argument E' and transforms it into the thing denoted by $E[E'/V]$ (the result of substituting E' for V in E, see Section 1.8). For example, $\lambda x.\ \mathbf{sum}(x, \underline{1})$ denotes the add-one function.

Using BNF, the syntax of λ-expressions is just:

$$\begin{array}{rcl} <\lambda\textit{-expression}> & ::= & <\textit{variable}> \\ & | & (<\lambda\textit{-expression}>\ <\lambda\textit{-expression}>) \\ & | & (\lambda\ <\textit{variable}>\ .\ <\lambda\textit{-expression}>) \end{array}$$

If V ranges over the syntax class $<\textit{variable}>$ and E, E_1, E_2, … etc. range over the syntax class $<\lambda\textit{-expression}>$, then the BNF simplifies to:

$$E ::= \underset{\text{variables}}{V} \mid \underbrace{(E_1\ E_2)}_{\substack{\text{applications}\\ \text{(combinations)}}} \mid \underbrace{\lambda V.\ E}_{\text{abstractions}}$$

The description of the meaning of λ-expressions just given above is vague and intuitive. It took about 40 years for logicians (Dana Scott, in fact [32]) to make it rigorous in a useful way. We shall not be going into details of this.

Example: $(\lambda x.\ x)$ denotes the 'identity function': $((\lambda x.\ x)\ E) = E$. □

Example: $(\lambda x.\ (\lambda f.\ (f\ x)))$ denotes the function which when applied to E yields $(\lambda f.\ (f\ x))[E/x]$, i.e. $(\lambda f.\ (f\ E))$. This is the function which when applied to E' yields $(f\ E)[E'/f]$ i.e. $(E'\ E)$. Thus

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

and

$$((\lambda f.\ (f\ E))\ E') = (E'\ E)$$

□

[1] Note that **sum** is a λ-expression, whereas $+$ is a mathematical symbol in the 'metalanguage' (i.e. English) that we are using for talking about the λ-calculus.

Exercise 1
Describe the function denoted by $(\lambda x.\ (\lambda y.\ y))$. $\square$

Example: Section 2.3 describes how numbers can be represented by λ-expressions. Assume that this has been done and that $\underline{0}$, $\underline{1}$, $\underline{2}$, ... are λ-expressions which represent 0, 1, 2, ..., respectively. Assume also that **add** is a λ-expression denoting a function satisfying:

$$((\mathbf{add}\ \underline{m})\ \underline{n}) = \underline{m+n}.$$

Then $(\lambda x.\ ((\mathbf{add}\ \underline{1})\ x))$ is a λ-expression denoting the function that transforms $\underline{n}$ to $\underline{1+n}$, and $(\lambda x.\ (\lambda y.\ ((\mathbf{add}\ x) y)))$ is a λ-expression denoting the function that transforms $\underline{m}$ to the function which when applied to $\underline{n}$ yields $\underline{m+n}$, namely $\lambda y.\ ((\mathbf{add}\ \underline{m}) y))$. $\square$

The relationship between the function **sum** in (ii) at the beginning of this section (page 2) and the function **add** in the previous example is explained in Section 2.5.

1.2 Notational conventions

The following conventions help minimize the number of brackets one has to write.

1. Function application associates to the left, i.e. $E_1\ E_2\ \cdots\ E_n$ means $((\cdots(E_1\ E_2)\ \cdots)\ E_n)$. For example:

$E_1\ E_2$	means	$(E_1\ E_2)$
$E_1\ E_2\ E_3$	means	$((E_1\ E_2) E_3)$
$E_1\ E_2\ E_3\ E_4$	means	$(((E_1\ E_2) E_3) E_4)$

2. $\lambda V.\ E_1\ E_2\ \ldots\ E_n$ means $(\lambda V.\ (E_1\ E_2\ \ldots\ E_n))$. Thus the scope of '$\lambda V$' extends as far to the right as possible.

3. $\lambda V_1\ \cdots\ V_n.\ E$ means $(\lambda V_1.\ (\ \cdots\ .\ (\lambda V_n.\ E)\ \cdots\ ))$. For example:

$\lambda x\ y.\ E$	means	$(\lambda x.\ (\lambda y.\ E))$
$\lambda x\ y\ z.\ E$	means	$(\lambda x.\ (\lambda y.\ (\lambda z.\ E)))$
$\lambda x\ y\ z\ w.\ E$	means	$(\lambda x.\ (\lambda y.\ (\lambda z.\ (\lambda w.\ E))))$

Example: $\lambda x\ y.\ \mathbf{add}\ y\ x$ means $(\lambda x.\ (\lambda y.\ ((\mathbf{add}\ y)\ x)))$. $\square$

1.3 Free and bound variables

An occurrence of a variable V in a λ-expression is *free* if it is not within the scope of a 'λV', otherwise it is *bound*. For example

$$(\lambda x.\ \underset{\text{free}}{y}\ \underset{\text{bound}}{x})(\lambda y.\ \underset{\text{free}}{x}\ \underset{\text{bound}}{y})$$

1.4 Conversion rules

In Chapter 2 it is explained how λ-expressions can be used to represent data objects like numbers, strings etc. For example, an arithmetic expression like $(2+3) \times 5$ can be represented as a λ-expression and its 'value' 25 can also be represented as a λ-expression. The process of 'simplifying' $(2+3) \times 5$ to 25 will be represented by a process called *conversion* (or *reduction*). The rules of λ-conversion described below are very general, yet when they are applied to λ-expressions representing arithmetic expressions they simulate arithmetical evaluation.

There are three kinds of λ-conversion called α-conversion, β-conversion and η-conversion (the original motivation for these names is not clear). In stating the conversion rules the notation $E\texttt{[}E'/V\texttt{]}$ is used to mean the result of substituting E' for each *free* occurrence of V in E. The substitution is called *valid* if and only if no free variable in E' becomes bound in $E\texttt{[}E'/V\texttt{]}$. Substitution is described in more detail in Section 1.8.

The rules of λ-conversion

- **α-conversion**.

  Any abstraction of the form $\lambda V.\ E$ can be converted to $\lambda V'.\ E\texttt{[}V'/V\texttt{]}$ provided the substitution of V' for V in E is valid.

- **β-conversion**.

  Any application of the form $(\lambda V.\ E_1)\ E_2$ can be converted to $E_1\texttt{[}E_2/V\texttt{]}$, provided the substitution of E_2 for V in E_1 is valid.

- **η-conversion**.

  Any abstraction of the form $\lambda V.\ (E\ V)$ in which V has no free occurrence in E can be reduced to E.

The following notation will be used:

- $E_1 \underset{\alpha}{\longrightarrow} E_2$ means E_1 α-converts to E_2.
- $E_1 \underset{\beta}{\longrightarrow} E_2$ means E_1 β-converts to E_2.
- $E_1 \underset{\eta}{\longrightarrow} E_2$ means E_1 η-converts to E_2.

In Section 1.4.4 below this notation is extended.

The most important kind of conversion is β-conversion; it is the one that can be used to simulate arbitrary evaluation mechanisms. α-conversion is to do with the technical manipulation of bound variables and η-conversion expresses the fact that two functions that always give the same results on the same arguments are equal (see Section 1.7). The next three subsections give further explanation and examples of the three kinds of conversion (note that 'conversion' and 'reduction' are used below as synonyms).

1.4.1 α-conversion

A λ-expression (necessarily an abstraction) to which α-reduction can be applied is called an *α-redex*. The term 'redex' abbreviates 'reducible expression'. The rule of α-conversion just says that bound variables can be renamed provided no 'name-clashes' occur.

Examples

$$\lambda x.\ x \underset{\alpha}{\longrightarrow} \lambda y.\ y$$

$$\lambda x.\ f\ x \underset{\alpha}{\longrightarrow} \lambda y.\ f\ y$$

It is *not* the case that

$$\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y \underset{\alpha}{\longrightarrow} \lambda y.\ \lambda y.\ \mathbf{add}\ y\ y$$

because the substitution $(\lambda y.\ \mathbf{add}\ x\ y)[y/x]$ is not valid since the y that replaces x becomes bound. $\square$

1.4.2 β-conversion

A λ-expression (necessarily an application) to which β-reduction can be applied is called a *β-redex*. The rule of β-conversion is like the evaluation of a function call in a programming language: the body E_1 of the function $\lambda V.\ E_1$ is evaluated in an environment in which the 'formal parameter' V is bound to the 'actual parameter' E_2.

Examples

$$(\lambda x.\ f\ x)\ E \underset{\beta}{\longrightarrow} f\ E$$
$$(\lambda x.\ (\lambda y.\ \mathbf{add}\ x\ y))\ \underline{3} \underset{\beta}{\longrightarrow} \lambda y.\ \mathbf{add}\ \underline{3}\ y$$
$$(\lambda y.\ \mathbf{add}\ \underline{3}\ y)\ \underline{4} \underset{\beta}{\longrightarrow} \mathbf{add}\ \underline{3}\ \underline{4}$$

It is *not* the case that

$$(\lambda x.\ (\lambda y.\ \mathbf{add}\ x\ y))\ (\mathbf{square}\ y) \underset{\beta}{\longrightarrow} \lambda y.\ \mathbf{add}\ (\mathbf{square}\ y)\ y$$

because the substition $(\lambda y.\ \mathbf{add}\ x\ y)[(\mathbf{square}\ y)/x]$ is not valid, since y is free in $(\mathbf{square}\ y)$ but becomes bound after substitution for x in $(\lambda y.\ \mathbf{add}\ x\ y)$. $\square$

It takes some practice to parse λ-expressions according to the conventions of Section 1.2 so as to identify the β-redexes. For example, consider the application:

$$(\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y)\ \underline{3}\ \underline{4}.$$

Putting in brackets according to the conventions expands this to:

$$(((\lambda x.\ (\lambda y.\ ((\mathbf{add}\ x)\ y)))\ \underline{3})\ \underline{4})$$

which has the form:

$$((\lambda x.\ E)\ \underline{3})\ \underline{4}$$

where

$$E = (\lambda y.\ \mathbf{add}\ x\ y)$$

$(\lambda x.\ E)\ \underline{3}$ is a β-redex and could be reduced to $E[\underline{3}/x]$.

1.4.3 η-conversion

A λ-expression (necessarily an abstraction) to which η-reduction can be applied is called an *η-redex*. The rule of η-conversion expresses the property that two functions are equal if they give the same results when applied to the same arguments. This property is called *extensionality* and is discussed further in Section 1.7. For example, η-conversion ensures that $\lambda x.\ (\mathbf{sin}\ x)$ and $\mathbf{sin}$ denote the same function. More generally, $\lambda V.\ (E\ V)$ denotes the function which when applied to an argument E' returns $(E\ V)[E'/V]$. If V does not occur free in E then $(E\ V)[E'/V] = (E\ E')$. Thus $\lambda V.\ E\ V$ and E both yield the same result, namely $E\ E'$, when applied to the same arguments and hence they denote the same function.

Examples

$$\lambda x.\ \mathbf{add}\ x \xrightarrow[\eta]{} \mathbf{add}$$

$$\lambda y.\ \mathbf{add}\ x\ y \xrightarrow[\eta]{} \mathbf{add}\ x$$

It is *not* the case that

$$\lambda x.\ \mathbf{add}\ x\ x \xrightarrow[\eta]{} \mathbf{add}\ x$$

because x is free in $\mathbf{add}\ x$. □

1.4.4 Generalized conversions

The definitions of $\xrightarrow[\alpha]{}$, $\xrightarrow[\beta]{}$ and $\xrightarrow[\eta]{}$ can be generalized as follows:

- $E_1 \xrightarrow[\alpha]{} E_2$ if E_2 can be got from E_1 by α-converting any subterm.
- $E_1 \xrightarrow[\beta]{} E_2$ if E_2 can be got from E_1 by β-converting any subterm.
- $E_1 \xrightarrow[\eta]{} E_2$ if E_2 can be got from E_1 by η-converting any subterm.

Examples

$$((\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y)\ \underline{3})\ \underline{4} \xrightarrow[\beta]{} (\lambda y.\ \mathbf{add}\ \underline{3}\ y)\ \underline{4}$$

$$(\lambda y.\ \mathbf{add}\ \underline{3}\ y)\ \underline{4} \xrightarrow[\beta]{} \mathbf{add}\ \underline{3}\ \underline{4}$$

□

The first of these is a β-conversion in the generalized sense because $(\lambda y.\ \mathbf{add}\ \underline{3}\ y)\underline{4}$ is obtained from $((\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y)\underline{3})\underline{4}$ (which is not itself a β-redex) by reducing the subexpression $(\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y)\underline{3}$. We will sometimes write a sequence of conversions like the two above as:

$$((\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y)\ \underline{3})\ \underline{4} \xrightarrow[\beta]{} (\lambda y.\ \mathbf{add}\ \underline{3}\ y)\ \underline{4} \xrightarrow[\beta]{} \mathbf{add}\ \underline{3}\ \underline{4}$$

Exercise 2
Which of the three β-reductions below are generalized conversions (i.e. reductions of subexpressions) and which are conversions in the sense defined on page 4? □

(i) $(\lambda x.\ x)\ \underline{1} \xrightarrow[\beta]{} \underline{1}$

(ii) $(\lambda y.\ y)\ ((\lambda x.\ x)\ \underline{1}) \xrightarrow[\beta]{} (\lambda y.\ y)\underline{1} \xrightarrow[\beta]{} \underline{1}$

(iii) $(\lambda y.\ y)\ ((\lambda x.\ x)\ \underline{1}) \xrightarrow[\beta]{} (\lambda x.\ x)\ \underline{1} \xrightarrow[\beta]{} \underline{1}$

In reductions (ii) and (iii) in the exercise above one starts with the same λ-expression, but reduce redexes in different orders.

An important property of β-reductions is that no matter in which order one does them, one always ends up with equivalent results. If there are several disjoint redexes in an expression, one can reduce them in parallel. Note, however, that some reduction sequences may never terminate. This is discussed further in connection with the normalization theorem of Chapter 2.9. It is a current hot research topic in 'fifth-generation computing' to design processors which exploit parallel evaluation to speed up the execution of functional programs.

1.5 Equality of λ-expressions

The three conversion rules preserve the meaning of λ-expressions, i.e. if E_1 can be converted to E_2 then E_1 and E_2 denote the same function. This property of conversion should be intuitively clear. It is possible to give a mathematical definition of the function denoted by a λ-expression and then to prove that this function is unchanged by α-, β- or η-conversion. Doing this is surprisingly difficult [33] and is beyond the scope of this book.

We will simply *define* two λ-expressions to be equal if they can be transformed into each other by a sequence of (forwards or backwards) λ-conversions. It is important to be clear about the difference between *equality* and *identity*. Two λ-expressions are identical if they consist of *exactly* the same sequence of characters; they are equal if one can be converted to the other. For example, $\lambda x.\ x$ is equal to $\lambda y.\ y$, but not identical to it. The following notation is used:

- $E_1 \equiv E_2$ means E_1 and E_2 are identical.
- $E_1 = E_2$ means E_1 and E_2 are equal.

Equality ($=$) is defined in terms of identity ($\equiv$) and conversion ($\xrightarrow[\alpha]{}$, $\xrightarrow[\beta]{}$ and $\xrightarrow[\eta]{}$) as follows.

Equality of λ-expressions

If E and E' are λ-expressions then $E = E'$ if $E \equiv E'$ or there exist expressions $E_1, E_2, \ldots, E_n$ such that:

1. $E \equiv E_1$
2. $E' \equiv E_n$
3. For each i either
   (a) $E_i \xrightarrow[\alpha]{} E_{i+1}$ or $E_i \xrightarrow[\beta]{} E_{i+1}$ or $E_i \xrightarrow[\eta]{} E_{i+1}$ or
   (b) $E_{i+1} \xrightarrow[\alpha]{} E_i$ or $E_{i+1} \xrightarrow[\beta]{} E_i$ or $E_{i+1} \xrightarrow[\eta]{} E_i$.

Examples

$$(\lambda x.\ x)\ \underline{1}\ =\ \underline{1}$$
$$(\lambda x.\ x)\ ((\lambda y.\ y)\ \underline{1})\ =\ \underline{1}$$
$$(\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y)\ \underline{3}\ \underline{4}\ =\ \mathbf{add}\ \underline{3}\ \underline{4}$$

□

From the definition of $=$ it follows that:

(i) For any E it is the case that $E = E$ (equality is *reflexive*).

(ii) If $E = E'$, then $E' = E$ (equality is *symmetric*).

(iii) If $E = E'$ and $E' = E''$, then $E = E''$ (equality is *transitive*).

If a relation is reflexive, symmetric and transitive then it is called an *equivalence relation*. Thus $=$ is an equivalence relation.

Another important property of $=$ is that if $E_1 = E_2$ and if E_1' and E_2' are two λ-expressions that only differ in that where one contains E_1 the other contains E_2, then $E_1' = E_2'$. This property is called *Leibnitz's law*. It holds because the same sequence of reduction for getting from E_1 to E_2 can be used for getting from E_1' to E_2'. For example, if $E_1 = E_2$, then by Leibnitz's law $\lambda V.\ E_1\ =\ \lambda V.\ E_2$.

It is essential for the substitutions in the α- and β-reductions to be valid. The validity requirement disallows, for example, $\lambda x.\ (\lambda y.\ x)$ being α-reduced to $\lambda y.\ (\lambda y.\ y)$ (since y becomes bound after substitution for x in $\lambda y.\ x$). If this invalid substitution were permitted, then it would follow by the definition of $=$ that:

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

But then since:

$$(\lambda x.\ (\lambda y.\ x))\ \underline{1}\ \underline{2} \underset{\beta}{\longrightarrow} (\lambda y.\ \underline{1})\ \underline{2} \underset{\beta}{\longrightarrow} \underline{1}$$

and

$$(\lambda y.\ (\lambda y.\ y))\ \underline{1}\ \underline{2} \underset{\beta}{\longrightarrow} (\lambda y.\ y)\ \underline{2} \underset{\beta}{\longrightarrow} \underline{2}$$

one would be forced to conclude that $\underline{1} = \underline{2}$. More generally by replacing $\underline{1}$ and $\underline{2}$ by any two expressions, it could be shown that any two expressions are equal!

Exercise 3
Find an example which shows that if substitutions in β-reductions are allowed to be invalid, then it follows that any two λ-expressions are equal. □

Example: If V_1, V_2, ..., V_n are all distinct and none of them occur free in any of E_1, E_2,..., E_n, then

$$\begin{array}{ll} (\lambda V_1\ V_2 \cdots V_n.\ E)\ E_1\ E_2 \cdots E_n & \\ = & ((\lambda V_1.\ (\lambda V_2 \cdots V_n.\ E))E_1)\ E_2 \cdots E_n \\ \underset{\beta}{\longrightarrow} & ((\lambda V_2 \cdots V_n.\ E)[E_1/V_1])\ E_2 \cdots E_n \\ = & (\lambda V_2 \ldots V_n.\ E[E_1/V_1])E_2 \cdots E_n \\ & \vdots \\ = & E[E_1/V_1][E_2/V_2] \cdots [E_n/V_n] \end{array}$$

□

Exercise 4
In the last example, where was the assumption used that V_1, V_2,..., V_n are all distinct and that none of them occur free in any of E_1, E_2,..., E_n? $\Box$

Exercise 5
Find an example to show that if $V_1 = V_2$, then even if V_2 is not free in E_1, it is not necessarily the case that:

$$(\lambda V_1 V_2.E)\ E_1\ E_2 = E\,[E_1/V_1]\,[E_2/V_2]$$

$\Box$

Exercise 6
Find an example to show that if $V_1 \neq V_2$, but V_2 occurs free in E_1, then it is not necessarily the case that:

$$(\lambda V_1 V_2.\ E)\ E_1\ E_2 = E\,[E_1/V_1]\,[E_2/V_2]$$

$\Box$

1.6 The $\longrightarrow$ relation

In the previous section $E_1 = E_2$ was defined to mean that E_2 could be obtained from E_1 by a sequence of forwards *or backwards* conversions. A special case of this is when E_2 is got from E_1 using only forwards conversions. This is written $E_1 \longrightarrow E_2$.

Definition of $\longrightarrow$

If E and E' are λ-expressions, then $E \longrightarrow E'$ if $E \equiv E'$ or there exist expressions E_1, E_2, ...,E_n such that:

1. $E \equiv E_1$
2. $E' \equiv E_n$
3. For each i either $E_i \underset{\alpha}{\longrightarrow} E_{i+1}$ or $E_i \underset{\beta}{\longrightarrow} E_{i+1}$ or $E_i \underset{\eta}{\longrightarrow} E_{i+1}$.

Notice that the definition of $\longrightarrow$ is just like the definition of $=$ on page 7 except that part (b) of 3 is missing.

Exercise 7
Find E, E' such that $E = E'$ but it is not the case that $E \longrightarrow E'$. $\Box$

Exercise 8
[very hard!] Show that if $E_1 = E_2$, then there exists E such that $E_1 \longrightarrow E$ and $E_2 \longrightarrow E$. (This property is called the Church-Rosser theorem. Some of its consequences are discussed in Chapter 2.9.) $\Box$

1.7 Extensionality

Suppose V does not occur free in E_1 or E_2 and

$$E_1\ V = E_2\ V$$

Then by Leibnitz's law (see page 8)

$$\lambda V.\ E_1\ V = \lambda V.\ E_2\ V$$

so by η-reduction applied to both sides

$$E_1 = E_2$$

It is often convenient to prove that two λ-expressions are equal using this property, i.e. to prove $E_1 = E_2$ by proving $E_1\ V = E_2\ V$ for some V not occuring free in E_1 or E_2. We will refer to such proofs as being *by extensionality*.

Exercise 9
Show that

$$(\lambda f\ g\ x.\ f\ x\ (g\ x))\ (\lambda x\ y.\ x)\ (\lambda x\ y.\ x)\ =\ \lambda x.\ x$$

□

1.8 Substitution

At the beginning of Section 1.4 $E[E'/V]$ was defined to mean the result of substituting E' for each *free* occurrence of V in E. The substitution was said to be valid if no free variable in E' became bound in $E[E'/V]$. In the definitions of α- and β-conversion, it was stipulated that the substitutions involved must be valid. Thus, for example, it was only the case that

$$(\lambda V.\ E_1)\ E_2 \underset{\beta}{\longrightarrow} E_1[E_2/V]$$

as long as the substitution $E_1[E_2/V]$ was valid.
It is very convenient to extend the meaning of $E[E'/V]$ so that we don't have to worry about validity. This is achieved by the definition below which has the property that for *all* expressions E, E_1 and E_2 and *all* variables V and V':

$$(\lambda V.\ E_1)\ E_2 \longrightarrow E_1[E_2/V] \quad \text{and} \quad \lambda V.\ E \longrightarrow \lambda V'.\ E[V'/V]$$

To ensure this property holds, $E[E'/V]$ is defined recursively on the structure of E as follows:

E	$E\,[E'/V]$
V	E'
V' (where $V \neq V'$)	V'
$E_1\ E_2$	$E_1\,[E'/V]\ E_2\,[E'/V]$
$\lambda V.\ E_1$	$\lambda V.\ E_1$
$\lambda V'.\ E_1$ (where $V \neq V'$ and V' is not free in E')	$\lambda V'.\ E_1\,[E'/V]$
$\lambda V'.\ E_1$ (where $V \neq V'$ and V' is free in E')	$\lambda V''.\ E_1\,[V''/V']\,[E'/V]$ where V'' is a variable not free in E' or E_1

This particular definition of $E\,[E'/V]$ is based on (but not identical to) the one in Appendix C of [2].

To illustrate how this works consider $(\lambda y.\ y\ x)\,[y/x]$. Since y is free in y, the last case of the table above applies. Since z does not occur in $y\ x$ or y,

$$(\lambda y.\ y\ x)\,[y/x] \equiv \lambda z.\ (y\ x)\,[z/y]\,[y/x] \equiv \lambda z.\ (z\ x)\,[y/x] \equiv \lambda z.\ z\ y$$

In the last line of the table above, the particular choice of V'' is not specified. Any variable not occurring in E' or E_1 will do.

A good discussion of substitution can be found in the book by Hindley and Seldin [19] where various technical properties are stated and proved. The following exercise is taken from that book.

Exercise 10
Use the table above to work out

(i) $(\lambda y.\ x\ (\lambda x.\ x))\,[(\lambda y.\ y\ x)/x]$.

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

□

It is straightforward, but rather tedious, to prove from the definition of $E\,[E'/V]$ just given that indeed

$$(\lambda V.\ E_1)\ E_2 \longrightarrow E_1\,[E_2/V] \quad \text{and} \quad \lambda V.\ E \longrightarrow \lambda V'.\ E\,[V'/V]$$

for all expressions E, E_1 and E_2 and all variables V and V'.

In Chapter 3 it will be shown how the theory of combinators can be used to decompose the complexities of substitution into simpler operations. Instead of combinators it is possible to use the so-called *nameless terms* of De Bruijn [6]. De Bruijn's idea is that variables can be thought of as 'pointers' to the λs that bind them. Instead of 'labelling' λs with names (i.e. bound variables) and then pointing to them via these names, one can point to the appropriate λ by giving the number of levels 'upwards' needed to reach it. For example, $\lambda x.\ \lambda y.\ x\ y$ would be represented by $\lambda\lambda 2\ 1$. As a more complicated example, consider the expression below in which we indicate the number of levels separating a variable from the λ that binds it.

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

In De Bruijn's notation this is $\lambda\lambda 2\ 1\ \lambda 3\ 1\ 1$.

A free variable in an expression is represented by a number bigger than the depth of λs above it; different free variables being assigned different numbers. For example,

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

would be represented by

$$\lambda(\lambda 1\ 2\ 3)\ 1\ 2\ 4$$

Since there are only two λs above the occurrence of 3, this number must denote a free variable; similarly there is only one λ above the second occurrence of 2 and the occurrence of 4, so these too must be free variables. Note that 2 could not be used to represent w since this had already been used to represent the free y; we thus chose the first available number bigger than 2 (3 was already in use representing z).

Care must be taken to assign big enough numbers to free variables. For example, the first occurrence of z in $\lambda x.\ z\ (\lambda y.\ z)$ could be represented by 2, but the second occurrence requires 3; since they are the same variable we must use 3.

Example: With De Bruijn's scheme $\lambda x.\ x\ (\lambda y.\ x\ y\ y)$ would be represented by $\lambda 1(\lambda 2\ 1\ 1)$. □

Exercise 11
What λ-expression is represented by $\lambda 2(\lambda 2)$? □

Exercise 12
Describe an algorithm for computating the De Bruijn representation of the expression $E[E'/V]$ from the representations of E and E'. □

Chapter 2

Representing Things in the λ-calculus

The λ-calculus appears at first sight to be a very primitive language. However, it can be used to represent most of the objects and structures needed for modern programming. The idea is to code these objects and structures in such a way that they have the required properties. For example, to represent the truth values *true* and *false* and the Boolean function $\neg$ ('not'), λ-expressions **true**, **false** and **not** are devised with the properties that:

$$\begin{array}{l}\textbf{not true} = \textbf{false}\\ \textbf{not false} = \textbf{true}\end{array}$$

To represent the Boolean function $\wedge$ ('and') a λ-expression **and** is devised such that:

$$\begin{array}{l}\textbf{and true true} = \textbf{true}\\ \textbf{and true false} = \textbf{false}\\ \textbf{and false true} = \textbf{false}\\ \textbf{and false false} = \textbf{false}\end{array}$$

and to represent $\vee$ ('or') an expression **or** such that:

$$\begin{array}{l}\textbf{or true true} = \textbf{true}\\ \textbf{or true false} = \textbf{true}\\ \textbf{or false true} = \textbf{true}\\ \textbf{or false false} = \textbf{false}\end{array}$$

The λ-expressions used to represent things may appear completely unmotivated at first. However, the definitions are chosen so that they work together in unison.

We will write

$$\texttt{LET} \sim = \lambda\textit{-expression}$$

to introduce $\sim$ as a new notation. Usually $\sim$ will just be a name such as **true** or **and**. Such names are written in **bold** face, or underlined, to distinguish them from variables. Thus, for example, *true* is a variable but **true** is the λ-expression $\lambda x.\ \lambda y.\ x$ (see Section 2.1 below) and 2 is a number but $\underline{2}$ is the λ-expression $\lambda f\ x.\ f(f\ x)$ (see Section 2.3).

Sometimes $\sim$ will be a more complicated form like the conditional notation $(E \rightarrow E_1 \mid E_2)$.

2.1 Truth-values and the conditional

This section defines λ-expressions **true**, **false**, **not** and $(E \rightarrow E_1 \mid E_2)$ with the following properties:

$$\begin{array}{l}\textbf{not true} = \textbf{false}\\ \textbf{not false} = \textbf{true}\\ \\ (\textbf{true} \to E_1 \mid E_2) = E_1\\ (\textbf{false} \to E_1 \mid E_2) = E_2\end{array}$$

The λ-expressions **true** and **false** represent the truth-values *true* and *false*, **not** represents the negation function $\neg$ and $(E \to E_1 \mid E_2)$ represents the conditional 'if E then E_1 else E_2'.

There are infinitely many different ways of representing the truth-values and negation that work; the ones used here are traditional and have been developed over the years by logicians.

$$\texttt{LET}\ \textbf{true} = \lambda x.\ \lambda y.\ x$$

$$\texttt{LET}\ \textbf{false} = \lambda x.\ \lambda y.\ y$$

$$\texttt{LET}\ \textbf{not} = \lambda t.\ t\ \textbf{false}\ \textbf{true}$$

It is easy to use the rules of λ-conversion to show that these definitions have the desired properties. For example:

$$\begin{array}{rll}\textbf{not true} & = (\lambda t.\ t\ \textbf{false true})\ \textbf{true} & \text{(definition of } \textbf{not}\text{)}\\ & = \textbf{true false true} & (\beta\text{-conversion})\\ & = (\lambda x.\ \lambda y.\ x)\ \textbf{false true} & \text{(definition of } \textbf{true}\text{)}\\ & = (\lambda y.\ \textbf{false})\ \textbf{true} & (\beta\text{-conversion})\\ & = \textbf{false} & (\beta\text{-conversion})\end{array}$$

Similarly **not false** $=$ **true**.

Conditional expressions $(E \to E_1 \mid E_2)$ can be defined as follows:

$$\texttt{LET}\ (E \to E_1 \mid E_2) = (E\ E_1\ E_2)$$

This means that for any λ-expressions E, E_1 and E_2, $(E \to E_1 \mid E_2)$ stands for $(E\ E_1\ E_2)$.

The conditional notation behaves as it should:

$$\begin{array}{rl}(\textbf{true} \to E_1 \mid E_2) & = \textbf{true}\ E_1\ E_2\\ & = (\lambda x\ y.\ x)\ E_1\ E_2\\ & = E_1\end{array}$$

and

$$\begin{array}{rl}(\textbf{false} \to E_1 \mid E_2) & = \textbf{false}\ E_1\ E_2\\ & = (\lambda x\ y.\ y)\ E_1\ E_2\\ & = E_2\end{array}$$

Exercise 13
Let **and** be the λ-expression $\lambda x\ y.\ (x \rightarrow y \mid \mathbf{false})$. Show that:

$$\begin{aligned}\mathbf{and\ true\ true} &= \mathbf{true}\\ \mathbf{and\ true\ false} &= \mathbf{false}\\ \mathbf{and\ false\ true} &= \mathbf{false}\\ \mathbf{and\ false\ false} &= \mathbf{false}\end{aligned}$$

$\square$

Exercise 14
Devise a λ-expression **or** such that:

$$\begin{aligned}\mathbf{or\ true\ true} &= \mathbf{true}\\ \mathbf{or\ true\ false} &= \mathbf{true}\\ \mathbf{or\ false\ true} &= \mathbf{true}\\ \mathbf{or\ false\ false} &= \mathbf{false}\end{aligned}$$

$\square$

2.2 Pairs and tuples

The following abbreviations represent pairs and n-tuples in the λ-calculus.

$$\texttt{LET}\ \mathbf{fst} = \lambda p.\ p\ \mathbf{true}$$

$$\texttt{LET}\ \mathbf{snd} = \lambda p.\ p\ \mathbf{false}$$

$$\texttt{LET}\ (E_1, E_2) = \lambda f.\ f\ E_1\ E_2$$

(E_1, E_2) is a λ-expression representing an ordered pair whose first component (i.e. E_1) is accessed with the function **fst** and whose second component (i.e. E_2) is accessed with **snd**. The following calculation shows how the various definitions co-operate together to give the right results.

$$\begin{aligned}\mathbf{fst}\ (E_1, E_2) &= (\lambda p.\ p\ \mathbf{true})\ (E_1, E_2)\\ &= (E_1, E_2)\ \mathbf{true}\\ &= (\lambda f.\ f\ E_1\ E_2)\ \mathbf{true}\\ &= \mathbf{true}\ E_1\ E_2\\ &= (\lambda x\ y.\ x)\ E_1\ E_2\\ &= E_1\end{aligned}$$

Exercise 15
Show that $\mathbf{snd}(E_1, E_2) = E_2$.
$\square$

A pair is a data-structure with two components. The generalization to n components is called an *n-tuple* and is easily defined in terms of pairs.

$$\texttt{LET}\ (E_1, E_2, \ldots, E_n) = (E_1, (E_2, (\cdots (E_{n-1}, E_n) \cdots)))$$

$(E_1, \ldots, E_n)$ is an *n-tuple* with *components* $E_1, \ldots, E_n$ and *length* n. Pairs are 2-tuples. The abbreviations defined next provide a way of extracting the components of n-tuples.

$$
\begin{array}{l}
\texttt{LET}\ E \overset{n}{\downarrow} 1 = \mathbf{fst}\ E \\[2ex]
\texttt{LET}\ E \overset{n}{\downarrow} 2 = \mathbf{fst}(\mathbf{snd}\ E) \\
\quad\vdots \\
\texttt{LET}\ E \overset{n}{\downarrow} i = \mathbf{fst}(\underbrace{\mathbf{snd}(\mathbf{snd}(\cdots(\mathbf{snd}}_{i-1\ \mathrm{snds}}\ E)\cdots))) \qquad (\text{if } i < n) \\
\quad\vdots \\
\texttt{LET}\ E \overset{n}{\downarrow} n = \underbrace{\mathbf{snd}(\mathbf{snd}(\ldots(\mathbf{snd}}_{n-1\ \mathrm{snds}}\ E)\ldots))
\end{array}
$$

It is easy to see that these definitions work, for example:

$$
\begin{aligned}
(E_1, E_2, \ldots, E_n) \overset{n}{\downarrow} 1 &= (E_1, (E_2, (\ldots))) \overset{n}{\downarrow} 1 \\
&= \mathbf{fst}\ (E_1, (E_2, (\ldots))) \\
&= E_1
\end{aligned}
$$

$$
\begin{aligned}
(E_1, E_2, \ldots, E_n) \overset{n}{\downarrow} 2 &= (E_1, (E_2, (\ldots))) \overset{n}{\downarrow} 2 \\
&= \mathbf{fst}\ (\mathbf{snd}\ (E_1, (E_2, (\ldots)))) \\
&= \mathbf{fst}\ (E_2, (\ldots)) \\
&= E_2
\end{aligned}
$$

In general $(E_1, E_2, \ldots, E_n) \overset{n}{\downarrow} i = E_i$ for all i such that $1 \leq i \leq n$.

Convention

We will usually just write $E \downarrow i$ instead of $E \overset{n}{\downarrow} i$ when it is clear from the context what n should be. For example,

$$(E_1, \ldots, E_n) \downarrow i \;=\; E_i \quad (\text{where } 1 \leq i \leq n)$$

2.3 Numbers

There are many ways to represent numbers by λ-expressions, each with their own advantages and disadvantages [38, 22]. The goal is to define for each number n a λ-expression $\underline{n}$ that represents it. We also want to define λ-expressions to represent the primitive arithmetical operations. For example, we will need λ-expressions **suc**, **pre**, **add** and **iszero** representing the successor function $(n \mapsto n + 1)$, the predecessor function $(n \mapsto n{-}1)$, addition and a test for zero, respectively. These λ-expressions will represent the numbers correctly if they have the following properties:

$$\textbf{suc}\ \underline{n} = \underline{n{+}1} \quad \text{(for all numbers } n\text{)}$$

$$\textbf{pre}\ \underline{n} = \underline{n{-}1} \quad \text{(for all numbers } n\text{)}$$

$$\textbf{add}\ \underline{m}\ \underline{n} = \underline{m{+}n} \quad \text{(for all numbers } m \text{ and } n\text{)}$$

$$\textbf{iszero}\ \underline{0} = \textbf{true}$$

$$\textbf{iszero}\ (\textbf{suc}\ \underline{n}) = \textbf{false}$$

The representation of numbers described here is the original one due to Church. In order to explain this it is convenient to define $f^n\ x$ to mean n applications of f to x. For example,

$$f^5\ x \ = \ f(f(f(f(f\ x))))$$

By convention $f^0\ x$ is defined to mean x. More generally:

$$\texttt{LET}\ E^0\ E' \ = \ E'$$

$$\texttt{LET}\ E^n\ E' \ = \ \underbrace{E(E(\cdots(E}_{n\ E\text{s}}\ E')\cdots))$$

Note that $E^n(EE') \ = \ E^{n+1}\ E' \ = E(E^n\ E')$; we will use the fact later.

Example:

$$f^4 x \ = \ f(f(f(f\ x))) \ = \ f(f^3 x) \ = \ f^3(f\ x)$$

□

Using the notation just introduced we can now define Church's numerals. Notice how the definition of the λ-expression $\underline{n}$ below encodes a unary representation of n.

$$\texttt{LET}\ \underline{0} = \lambda f\ x.\ x$$

$$\texttt{LET}\ \underline{1} = \lambda f\ x.\ f\ x$$

$$\texttt{LET}\ \underline{2} = \lambda f\ x.\ f(f\ x)$$

$$\vdots$$

$$\texttt{LET}\ \underline{n} = \lambda f\ x.\ f^n\ x$$

$$\vdots$$

The representations of **suc**, **add** and **iszero** are now magically pulled out of a hat. The best way to see how they work is to think of them as operating on unary representations of numbers. The exercises that follow should help.

$$\texttt{LET}\ \textbf{suc} = \lambda n\ f\ x.\ n\ f(f\ x)$$

$$\texttt{LET}\ \textbf{add} = \lambda m\ n\ f\ x.\ m\ f\ (n\ f\ x)$$

$$\texttt{LET}\ \textbf{iszero} = \lambda n.\ n\ (\lambda x.\ \textbf{false})\ \textbf{true}$$

Exercise 16
Show:

(i) $\textbf{suc}\ \underline{0} = \underline{1}$

(ii) $\textbf{suc}\ \underline{5} = \underline{6}$

(iii) $\textbf{iszero}\ \underline{0} = \textbf{true}$

(iv) $\textbf{iszero}\ \underline{5} = \textbf{false}$

(v) $\textbf{add}\ \underline{0}\ \underline{1} = \underline{1}$

(vi) $\textbf{add}\ \underline{2}\ \underline{3} = \underline{5}$

□

Exercise 17
Show for all numbers m and n:

(i) $\textbf{suc}\ \underline{n} = \underline{n{+}1}$

(ii) $\textbf{iszero}\ (\textbf{suc}\ \underline{n}) = \textbf{false}$

(iii) $\textbf{add}\ \underline{0}\ \underline{n} = \underline{n}$

(iv) $\textbf{add}\ \underline{m}\ \underline{0} = \underline{m}$

(v) $\textbf{add}\ \underline{m}\ \underline{n} = \underline{m+n}$

□

The predecesor function is harder to define than the other primitive functions. The idea is that the predecessor of $\underline{n}$ is defined by using $\lambda f\ x.\ f^n\ x$ (i.e. $\underline{n}$) to obtain a function that applies f only $n{-}1$ times. The trick is to 'throw away' the first application of f in f^n. To achieve this, we first define a function **prefn** that operates on pairs and has the property that:

(i) $\textbf{prefn}\ f\ (\textbf{true}, x) = (\textbf{false}, x)$

(ii) $\textbf{prefn}\ f\ (\textbf{false}, x) = (\textbf{false}, f\ x)$

From this it follows that:

(iii) $(\textbf{prefn}\ f)^n\ (\textbf{false}, x) = (\textbf{false}, f^n\ x)$

(iv) $(\textbf{prefn}\ f)^n\ (\textbf{true}, x) = (\textbf{false}, f^{n-1}\ x) \quad (\text{if } n > 0)$

Thus n applications of **prefn** to $(\textbf{true}, x)$ result in $n{-}1$ applications of f to x. With this idea, the definition of the predecessor function **pre** is straightforward. Before giving it, here is the definition of **prefn**:

$$\texttt{LET}\ \textbf{prefn} = \lambda f\ p.\ (\textbf{false},\ (\textbf{fst}\ p \rightarrow \textbf{snd}\ p \mid (f(\textbf{snd}\ p))))$$

Exercise 18
Show $\textbf{prefn}\ f\ (b, x) = (\textbf{false}, (b \rightarrow x \mid f\ x))$ and hence:

(i) $\textbf{prefn}\ f\ (\textbf{true}, x) = (\textbf{false}, x)$

(ii) $\textbf{prefn}\ f\ (\textbf{false}, x) = (\textbf{false}, f\ x)$

(iii) $(\textbf{prefn}\ f)^n\ (\textbf{false}, x) = (\textbf{false}, f^n\ x)$

(iv) $(\textbf{prefn}\ f)^n\ (\textbf{true}, x) = (\textbf{false}, f^{n-1}\ x) \quad (\text{if } n > 0)$

□

The predecessor function **pre** can now be defined.

$$\texttt{LET}\ \textbf{pre}\ = \lambda n\ f\ x.\ \textbf{snd}\ (n\ (\textbf{prefn}\ f)\ (\textbf{true}, x))$$

It follows that if $n > 0$

$$\begin{array}{rcll} \textbf{pre}\ \underline{n}\ f\ x & = & \textbf{snd}\ (\underline{n}\ (\textbf{prefn}\ f)\ (\textbf{true}, x)) & (\text{definition of } \textbf{pre}) \\ & = & \textbf{snd}\ ((\textbf{prefn}\ f)^n\ (\textbf{true}, x)) & (\text{definition of } \underline{n}) \\ & = & \textbf{snd}(\textbf{false}, f^{n-1}\ x) & (\text{by (v) above}) \\ & = & f^{n-1}\ x & \end{array}$$

hence by extensionality (Section 1.7 on page 10)

$$\begin{array}{rcll} \textbf{pre}\ \underline{n} & = & \lambda f\ x.\ f^{n-1}\ x & \\ & = & \underline{n{-}1} & (\text{definition of } \underline{n{-}1}) \end{array}$$

Exercise 19
Using the results of the previous exercise (or otherwise) show that

(i) $\textbf{pre}\ (\textbf{suc}\ \underline{n}) = \underline{n}$

(ii) $\textbf{pre}\ \underline{0} = \underline{0}$

□

The numeral system in the next exercise is the one used in [2] and has some advantages over Church's (e.g. the predecessor function is easier to define).

Exercise 20

$$\texttt{LET}\ \widehat{\underline{0}} = \lambda x.x$$

$$\texttt{LET}\ \widehat{\underline{1}} = (\textbf{false}, \widehat{\underline{0}})$$

$$\texttt{LET}\ \widehat{\underline{2}} = (\textbf{false}, \widehat{\underline{1}})$$

$$\vdots$$

$$\texttt{LET}\ \widehat{\underline{n{+}1}} = (\textbf{false}, \widehat{\underline{n}})$$

$$\vdots$$

Devise λ-expressions $\widehat{\textbf{suc}}$, $\widehat{\textbf{iszero}}$, $\widehat{\textbf{pre}}$ such that for all n:

(i) $\widehat{\textbf{suc}}\ \widehat{\underline{n}} = \widehat{\underline{n{+}1}}$

(ii) $\widehat{\textbf{iszero}}\ \widehat{\underline{0}} = \textbf{true}$

(iii) $\widehat{\textbf{iszero}}\ (\widehat{\textbf{suc}}\ \widehat{\underline{n}}) = \textbf{false}$

(iv) $\widehat{\textbf{pre}}\ (\widehat{\textbf{suc}}\ \widehat{\underline{n}}) = \widehat{\underline{n}}$

□

2.4 Definition by recursion

To represent the multiplication function in the λ-calculus we would like to define a λ-expression, **mult** say, such that:

$$\textbf{mult}\ m\ n = \underbrace{\textbf{add}\ n\ (\textbf{add}\ n\ (\ \cdots\ (\textbf{add}\ n\ \underline{0})\ \cdots\ ))}_{m\ \textbf{add}\text{s}}$$

This would be achieved if **mult** could be defined to satisfy the equation:

$$\textbf{mult}\ m\ n = (\textbf{iszero}\ m \to \underline{0} \mid \textbf{add}\ n\ (\textbf{mult}\ (\textbf{pre}\ m)\ n))$$

If this held then, for example,

$$\begin{aligned}
\textbf{mult}\ \underline{2}\ \underline{3} &= (\textbf{iszero}\ \underline{2} \to \underline{0} \mid \textbf{add}\ \underline{3}\ (\textbf{mult}\ (\textbf{pre}\ \underline{2})\ \underline{3})) \\
&\qquad\qquad \text{(by the equation)} \\
&= \textbf{add}\ \underline{3}\ (\textbf{mult}\ \underline{1}\ \underline{3}) \\
&\qquad\qquad \text{(by properties of } \textbf{iszero}\text{, the conditional and } \textbf{pre}\text{)} \\
&= \textbf{add}\ \underline{3}\ (\textbf{iszero}\ \underline{1} \to \underline{0} \mid \textbf{add}\ \underline{3}\ (\textbf{mult}\ (\textbf{pre}\ \underline{1})\ \underline{3})) \\
&\qquad\qquad \text{(by the equation)} \\
&= \textbf{add}\ \underline{3}\ (\textbf{add}\ \underline{3}\ (\textbf{mult}\ \underline{0}\ \underline{3})) \\
&\qquad\qquad \text{(by properties of } \textbf{iszero}\text{, the conditional and } \textbf{pre}\text{)} \\
&= \textbf{add}\ \underline{3}\ (\textbf{add}\ \underline{3}\ (\textbf{iszero}\ \underline{0} \to \underline{0} \mid \textbf{add}\ \underline{3}\ (\textbf{mult}\ (\textbf{pre}\ \underline{0})\ \underline{3}))) \\
&\qquad\qquad \text{(by the equation)} \\
&= \textbf{add}\ \underline{3}\ (\textbf{add}\ \underline{3}\ \underline{0}) \\
&\qquad\qquad \text{(by properties of } \textbf{iszero} \text{ and the conditional)}
\end{aligned}$$

The equation above suggests that **mult** be defined by:

$$\textbf{mult} = \lambda m\ n.\ (\textbf{iszero}\ m \to \underline{0} \mid \textbf{add}\ n\ (\underset{\uparrow\ \text{N.B.}}{\textbf{mult}}\ (\textbf{pre}\ m)\ n))$$

Unfortunately, this cannot be used to define **mult** because, as indicated by the arrow, **mult** must already be defined for the λ-expression to the right of the equals to make sense.

Fortunately, there is a technique for constructing λ-expressions that satisfy arbitrary equations. When this technique is applied to the equation above it gives the desired definition of **mult**. First define a λ-expression **Y** that, for any expression E, has the following odd property:

$$\textbf{Y}\ E = E\ (\textbf{Y}\ E)$$

This says that **Y** E is unchanged when the function E is applied to it. In general, if $E\ E' = E'$ then E' is called a *fixed point* of E. A λ-expression **Fix** with the property that $\textbf{Fix}\ E = E(\textbf{Fix}\ E)$ for any E is called a *fixed-point operator*. There are known to be infinitely many different fixed-point operators [28]; **Y** is the most famous one, and its definition is:

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

It is straightforward to show that $\mathbf{Y}$ is indeed a fixed-point operator:

$$\begin{array}{rlr}
\mathbf{Y}\ E & = (\lambda f.\ (\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x)))\ E & \text{(definition of } \mathbf{Y}\text{)} \\
 & = (\lambda x.\ E(x\ x))\ (\lambda x.\ E(x\ x)) & (\beta\text{-conversion}) \\
 & = E\ ((\lambda x.\ E(x\ x))\ (\lambda x.\ E(x\ x))) & (\beta\text{-conversion}) \\
 & = E\ (\mathbf{Y}\ E) & \text{(the line before last)}
\end{array}$$

This calculation shows that every λ-expression E has a fixed point.
Armed with $\mathbf{Y}$, we can now return to the problem of solving the equation for **mult**. Suppose **multfn** is defined by

$$\texttt{LET}\ \mathbf{multfn} = \lambda f\ m\ n.\ (\mathbf{iszero}\ m \rightarrow \underline{0} \mid \mathbf{add}\ n\ (f\ (\mathbf{pre}\ m)\ n))$$

and then define **mult** by:

$$\texttt{LET}\ \mathbf{mult} = \mathbf{Y}\ \mathbf{multfn}$$

Then:

$$\begin{array}{rlr}
\mathbf{mult}\ m\ n & = (\mathbf{Y}\ \mathbf{multfn})\ m\ n & \text{(definition of } \mathbf{mult}\text{)} \\
 & = \mathbf{multfn}\ (\mathbf{Y}\ \mathbf{multfn})\ m\ n & \text{(fixed-point property of } \mathbf{Y}\text{)} \\
 & = \mathbf{multfn}\ \mathbf{mult}\ m\ n & \text{(definition of } \mathbf{mult}\text{)} \\
 & = (\lambda f\ m\ n.\ (\mathbf{iszero}\ m \rightarrow \underline{0} \mid \mathbf{add}\ n\ (f\ (\mathbf{pre}\ m)\ n)))\ \mathbf{mult}\ m\ n & \\
 & & \text{(definition of } \mathbf{multfn}\text{)} \\
 & = (\mathbf{iszero}\ m \rightarrow \underline{0} \mid \mathbf{add}\ n\ (\mathbf{mult}\ (\mathbf{pre}\ m)\ n)) & (\beta\text{-conversion})
\end{array}$$

An equation of the form $f\ x_1\ \cdots\ x_n = E$ is called *recursive* if f occurs free in E. $\mathbf{Y}$ provides a general way of solving such equations. Start with an equation of the form:

$$\mathbf{f}\ x_1 \ldots x_n = \frown\!\!\mathbf{f}\!\!\frown$$

where $\frown\!\!\mathbf{f}\!\!\frown$ is some λ-expression containing $\mathbf{f}$. To obtain an $\mathbf{f}$ so that this equation holds define:

$$\texttt{LET}\ \mathbf{f} = \mathbf{Y}\ (\lambda f\ x_1 \ldots x_n. \frown f \frown)$$

The fact that the equation is satisfied can be shown as follows:

$$\begin{array}{rlr}
\mathbf{f}\ x_1 \ldots x_n & = \mathbf{Y}\ (\lambda f\ x_1 \ldots x_n. \frown f \frown)\ x_1 \ldots x_n & \text{(definition of } \mathbf{f}\text{)} \\
 & = (\lambda f\ x_1 \ldots x_n. \frown f \frown)\ (\mathbf{Y}\ (\lambda f\ x_1 \ldots x_n. \frown f \frown))\ x_1 \ldots x_n & \\
 & & \text{(fixed-point property)} \\
 & = (\lambda f\ x_1 \ldots x_n. \frown f \frown)\ \mathbf{f}\ x_1 \ldots x_n & \text{(definition of } \mathbf{f}\text{)} \\
 & = \frown\!\!\mathbf{f}\!\!\frown & (\beta\text{-conversion})
\end{array}$$

Exercise 21
Construct a λ-expression **eq** such that

$$\begin{array}{rl}
\mathbf{eq}\ m\ n = & (\mathbf{iszero}\ m \rightarrow \mathbf{iszero}\ n \mid \\
 & \quad (\mathbf{iszero}\ n \rightarrow \mathbf{false} \mid \mathbf{eq}\ (\mathbf{pre}\ m)\ (\mathbf{pre}\ n)))
\end{array}$$

□

Exercise 22
Show that if $\mathbf{Y}_1$ is defined by:

$$\texttt{LET } \mathbf{Y}_1 = \mathbf{Y}\ (\lambda y\ f.\ f(y\ f))$$

then $\mathbf{Y}_1$ is a fixed-point operator, i.e. for any E:

$$\mathbf{Y}_1\ E = E\ (\mathbf{Y}_1\ E)$$

□

The fixed-point operator in the next exercise is due to Turing (Barendregt [2], page 132).

Exercise 23
Show that $(\lambda x\ y.\ y\ (x\ x\ y))\ (\lambda x\ y.\ y\ (x\ x\ y))$ is a fixed-point operator. □

The next exercise also comes from Barendregt's book, where it is attributed to Klop.

Exercise 24
Show that $\mathbf{Y}_2$ is a fixed-point operator, where:

$$\begin{array}{l} \texttt{LET } \pounds = \lambda abcdefghijklmnopqstuvwxyzr. \\ \qquad\qquad r(thisisafixedpointcombinator) \\ \texttt{LET } \mathbf{Y}_2 = \pounds \end{array}$$

□

Exercise 25
Is it the case that $\mathbf{Y}\ f \longrightarrow f\ (\mathbf{Y}\ f)$? If so prove it; if not find a λ-expression $\widehat{\mathbf{Y}}$ such that $\widehat{\mathbf{Y}}\ f \longrightarrow f\ (\widehat{\mathbf{Y}}\ f)$. □

In the pure λ-calculus as defined on page 1, λ-expressions could only be applied to a single argument; however, this argument could be a *tuple* (see page 16). Thus one can write:

$$E(E_1, \ldots, E_n)$$

which actually abbreviates:

$$E(E_1, (E_2, (\cdots (E_{n-1}, E_n) \cdots)))$$

For example, $E(E_1, E_2)$ abbreviates $E(\lambda f.\ f\ E_1\ E_2)$.

2.5 Functions with several arguments

In conventional mathematical usage, the application of an n-argument function f to arguments $x_1, \ldots, x_n$ would be written as $f(x_1, \ldots, x_n)$. There are two ways of representing such applications in the λ-calculus:

(i) as $(f\ x_1\ \ldots\ x_n)$, or

(ii) as the application of f to an n-tuple $(x_1, \ldots, x_n)$.

In case (i), f expects its arguments 'one at a time' and is said to be *curried* after a logician called Curry (the idea of currying was actually invented by Schönfinkel [31]). The functions **and**, **or** and **add** defined earlier were all curried. One advantage of curried functions is that they can be 'partially applied'; for example, **add** $\underline{1}$ is the result of partially applying **add** to $\underline{1}$ and denotes the function $n \mapsto n{+}1$.

Although it is often convenient to represent n-argument functions as curried, it is also useful to be able to represent them, as in case (ii) above, by λ-expressions expecting a single n-tuple argument. For example, instead of representing $+$ and $\times$ by λ-expressions **add** and **mult** such that

$$\begin{aligned} \mathbf{add}\ \underline{m}\ \underline{n} &= \underline{m{+}n} \\ \mathbf{mult}\ \underline{m}\ \underline{n} &= \underline{m{\times}n} \end{aligned}$$

it might be more convenient to represent them by functions, **sum** and **prod** say, such that

$$\begin{aligned} \mathbf{sum}\ (\underline{m}, \underline{n}) &= \underline{m{+}n} \\ \mathbf{prod}\ (\underline{m}, \underline{n}) &= \underline{m{\times}n} \end{aligned}$$

This is nearer to conventional mathematical usage and has applications that will be encountered later. One might say that **sum** and **prod** are *uncurried* versions of **add** and **mult** respectively.

Define:

$$\begin{aligned} &\texttt{LET}\ \mathbf{curry} = \lambda f\ x_1\ x_2.\ f\ (x_1, x_2) \\ &\texttt{LET}\ \mathbf{uncurry} = \lambda f\ p.\ f\ (\mathbf{fst}\ p)\ (\mathbf{snd}\ p) \end{aligned}$$

then defining

$$\begin{aligned} \mathbf{sum} &= \ \mathbf{uncurry\ add} \\ \mathbf{prod} &= \ \mathbf{uncurry\ mult} \end{aligned}$$

results in **sum** and **prod** having the desired properties; for example:

$$\begin{aligned} \mathbf{sum}\ (\underline{m}, \underline{n}) &= \mathbf{uncurry\ add}\ (\underline{m}, \underline{n}) \\ &= (\lambda f\ p.\ f\ (\mathbf{fst}\ p)\ (\mathbf{snd}\ p))\mathbf{add}\ (\underline{m}, \underline{n}) \\ &= \mathbf{add}\ (\mathbf{fst}\ (\underline{m}, \underline{n}))\ (\mathbf{snd}\ (\underline{m}, \underline{n})) \\ &= \mathbf{add}\ \underline{m}\ \underline{n} \\ &= \underline{m{+}n} \end{aligned}$$

Exercise 26

Show that for any E:

$$\begin{aligned} \mathbf{curry}\ (\mathbf{uncurry}\ E) = E \\ \mathbf{uncurry}\ (\mathbf{curry}\ E) = E \end{aligned}$$

hence show that:

$$\begin{aligned} \mathbf{add} &= \mathbf{curry\ sum} \\ \mathbf{mult} &= \mathbf{curry\ prod} \end{aligned}$$

□

We can define n-ary functions for currying and uncurrying. For $n > 0$ define:

$$\texttt{LET}\ \mathbf{curry}_n = \lambda f\ x_1 \cdots x_n.\ f\ (x_1, \ldots, x_n)$$

$$\texttt{LET}\ \mathbf{uncurry}_n = \lambda f\ p.\ f\ (p \overset{n}{\downarrow} 1)\ \cdots\ (p \overset{n}{\downarrow} n)$$

If E represents a function expecting an n-tuple argument, then $\mathbf{curry}_n\ E$ represents the curried function which takes its arguments one at a time. If E represents a curried function of n arguments, then $\mathbf{uncurry}_n\ E$ represents the uncurried version which expects a single n-tuple as argument.

Exercise 27
Show that:

(i) $\mathbf{curry}_n\ (\mathbf{uncurry}_n\ E) = E$

(ii) $\mathbf{uncurry}_n\ (\mathbf{curry}_n\ E) = E$

□

Exercise 28
Devise λ-expressions E_1^n and E_2^n built out of **curry** and **uncurry** such that $\mathbf{curry}_n = E_1^n$ and $\mathbf{uncurry}_n = E_2^n$. □

The following notation provides a convenient way to write λ-expressions which expect tuples as arguments.

Generalized λ-abstractions

$$\texttt{LET}\ \lambda(V_1, \ldots, V_n).\ E = \mathbf{uncurry}_n\ (\lambda V_1\ \ldots\ V_n.\ E)$$

Example: $\lambda(x, y).\ \mathbf{mult}\ x\ y$ abbreviates:

$$\begin{aligned}\mathbf{uncurry}_2\ (\lambda x\ y.\ \mathbf{mult}\ x\ y) &= (\lambda f\ p.\ f\ (p \overset{2}{\downarrow} 1)\ (p \overset{2}{\downarrow} 2))\ (\lambda x\ y.\ \mathbf{mult}\ x\ y) \\ &= (\lambda f\ p.\ f\ (\mathbf{fst}\ p)\ (\mathbf{snd}\ p))\ (\lambda x\ y.\ \mathbf{mult}\ x\ y) \\ &= \lambda p.\ \mathbf{mult}\ (\mathbf{fst}\ p)(\mathbf{snd}\ p)\end{aligned}$$

Thus:

$$\begin{aligned}(\lambda(x, y).\ \mathbf{mult}\ x\ y)\ (E_1, E_2) &= (\lambda p.\ \mathbf{mult}\ (\mathbf{fst}\ p)\ (\mathbf{snd}\ p))\ (E_1, E_2) \\ &= \mathbf{mult}\ (\mathbf{fst}(E_1, E_2))\ (\mathbf{snd}(E_1, E_2)) \\ &= \mathbf{mult}\ E_1\ E_2\end{aligned}$$

□

This example illustrates the rule of *generalized β-conversion* in the box below. This rule can be derived from ordinary β-conversion and the definitions of tuples and generalized λ-abstractions. The idea is that a tuple of arguments is passed to each argument position in the body of the generalized abstraction; then each individual argument can be extracted from the tuple without affecting the others.

Generalized β-conversion

$$(\lambda(V_1,\ldots,V_n).\ E)\ (E_1,\ldots,E_n) = E\,[E_1,\ldots,E_n/V_1,\ldots,V_n]$$

where $E\,[E_1,\ldots,E_n/V_1,\ldots,V_n]$ is the *simultaneous substitution* of $E_1,\ldots,E_n$ for $V_1,\ldots,V_n$ respectively and none of these variables occur free in any of $E_1,\ldots,E_n$.

It is convenient to extend the notation $\lambda V_1\ V_2 \ldots V_n.\ E$ described on page **3** so that each V_i can either be an identifier or a tuple of identifiers. The meaning of $\lambda V_1\ V_2 \ldots V_n.\ E$ is still $\lambda V_1.(\lambda V_2.(\cdots(\lambda V_n.\ E)\cdots))$, but now if V_i is a tuple of identifiers then the expression is a generalized abstraction.

Example: $\lambda f\ (x,y).\ f\ x\ y$ means $\lambda f.\ (\lambda(x,y).\ f\ x\ y)$ which in turn means $\lambda f.$ **uncurry** $(\lambda x\ y.\ f\ x\ y)$ which equals $\lambda f.\ (\lambda p.\ f\ (\mathbf{fst}\ p)\ (\mathbf{snd}\ p))$. $\Box$

Exercise 29
Show that if the only free variables in E are $x_1,\ \ldots,\ x_n$ and f, then if:

$$\mathbf{f}\ =\ \mathbf{Y}\ (\lambda f\ (x_1,\ldots,x_n).\ E)$$

then

$$\mathbf{f}\,(x_1,\ldots,x_n)\ =\ E\,[\mathbf{f}/f]$$

$\Box$

Exercise 30
Define a λ-expression **div** with the property that:

$$\mathbf{div}\ (\underline{m},\underline{n}) = (\underline{q},\underline{r})$$

where q and r are the quotient and remainder of dividing n into m. $\Box$

2.6 Mutual recursion

To solve a set of mutually recursive equations like:

$$\begin{aligned}
\mathbf{f}_1 &= F_1\ \mathbf{f}_1 \cdots \mathbf{f}_n \\
\mathbf{f}_2 &= F_2\ \mathbf{f}_1 \cdots \mathbf{f}_n \\
&\ \vdots \\
\mathbf{f}_n &= F_n\ \mathbf{f}_1 \cdots \mathbf{f}_n
\end{aligned}$$

we simply define for $1 \le i \le n$

$$\mathbf{f}_i = \mathbf{Y}\ (\lambda(f_1,\ldots f_n).\ (F_1\ f_1\cdots f_n,\ \ldots\ ,F_n\ f_1\cdots f_n))\downarrow i$$

This works because if

$$\vec{\mathbf{f}} = \mathbf{Y}\ (\lambda(f_1,\ldots f_n).\ (F_1\ f_1\cdots f_n,\ \ldots\ ,F_n\ f_1\cdots f_n))$$

then $\mathbf{f}_i = \vec{\mathbf{f}}\downarrow i$ and hence:

$$\begin{aligned}
\vec{\mathbf{f}} &= (\lambda(f_1,\ldots,f_n).\ (F_1\ f_1\cdots f_n,\ \ldots\ ,F_n\ f_1\cdots f_n))\vec{\mathbf{f}} \\
&= (F_1(\vec{\mathbf{f}}\downarrow 1)\cdots(\vec{\mathbf{f}}\downarrow n),\ \ldots\ ,F_n(\vec{\mathbf{f}}\downarrow 1)\cdots(\vec{\mathbf{f}}\downarrow n)) \\
&= (F_1\ \mathbf{f}_1\cdots\mathbf{f}_n,\ \ldots\ ,F_n\ \mathbf{f}_1\cdots\mathbf{f}_n) \qquad (\text{since } \vec{\mathbf{f}}\downarrow i = \mathbf{f}_i).
\end{aligned}$$

Hence:

$$\mathbf{f}_i = F_i\ \mathbf{f}_1\cdots\mathbf{f}_n$$

2.7 Representing the recursive functions

The *recursive functions* form an important class of numerical functions. Shortly after Church invented the λ-calculus, Kleene proved that every recursive function could be represented in it. This provided evidence for *Church's thesis*, the hypothesis that any intuitively computable function could be represented in the λ-calculus. It has been shown that many other models of compution define the same class of functions that can be defined in the λ-calculus.

In this section it is described what it means for an arithmetic function to be represented in the λ-calculus. Two classes of functions, the *primitive recursive* functions and the *recursive* functions, are defined and it is shown that all the functions in these classes can be represented in the λ-calculus.

In Section 2.3 it was explained how a number n is represented by the λ-expression $\underline{n}$. A λ-expression $\underline{f}$ is said to represent a mathematical function f if for all numbers $x_1, \ldots, x_n$:

$$\underline{f}(\underline{x_1}, \ldots, \underline{x_n}) = \underline{y} \quad \text{if} \quad f(x_1, \ldots, x_n) = y$$

2.7.1 The primitive recursive functions

A function is called *primitive recursive* if it can be constructed from 0 and the functions S and U_n^i (defined below) by a finite sequence of applications of the operations of substitution and primitive recursion (also defined below).

The successor function S and projection functions U_n^i (where n and i are numbers) are defined by:

(i) $S(x) = x + 1$

(ii) $U_n^i(x_1, x_2, \ldots, x_n) = x_i$

Substitution

Suppose g is a function of r arguments and $h_1, \ldots, h_r$ are r functions each of n arguments. We say f is defined from g and $h_1, \ldots, h_r$ by substitution if:

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_r(x_1, \ldots, x_n))$$

Primitive recursion

Suppose g is a function of $n-1$ arguments and h is a function of $n+1$ arguments. We say f is defined from g and h by primitive recursion if:

$$\begin{aligned} f(0, x_2, \ldots, x_n) &= g(x_2, \ldots, x_n) \\ f(S(x_1), x_2, \ldots, x_n) &= h(f(x_1, x_2, \ldots, x_n), x_1, x_2, \ldots, x_n) \end{aligned}$$

g is called the *base function* and h is called the *step function*. It can proved that for any base and step function there always exists a unique function defined from them by primitive recursion. This result is called the primitive recursion theorem; proofs of it can be found in textbooks on mathematical logic.

Example: The addition function *sum* is primitive recursive because:

$$\begin{aligned} sum(0, x_2) &= x_2 \\ sum(S(x_1), x_2) &= S(sum(x_1, x_2)) \end{aligned}$$

□

It is now shown that every primitive recursive function can be represented by λ-expressions.

It is obvious that the λ-expressions $\underline{0}$, **suc**, $\lambda p.\ p \overset{n}{\downarrow} i$ represent the initial functions 0, S and U^i_n respectively.

Suppose function g of r variables is represented by $\mathbf{g}$ and functions h_i $(1 \leq i \leq r)$ of n variables are represented by $\mathbf{h}_i$. Then if a function f of n variables is defined by substitution by:

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_r(x_1, \ldots, x_n))$$

then f is represented by $\mathbf{f}$ where:

$$\mathbf{f} = \lambda(x_1, \ldots, x_n).\ \mathbf{g}(\mathbf{h}_1(x_1, \ldots, x_n), \ldots, \mathbf{h}_r(x_1, \ldots, x_n))$$

Suppose function f of n variables is defined inductively from a base function g of $n-1$ variables and an inductive step function h of $n+1$ variables. Then

$$\begin{aligned} f(0, x_2, \ldots, x_n) &= g(x_2, \ldots, x_n) \\ f(S(x_1), x_2, \ldots, x_n) &= h(f(x_1, x_2, \ldots, x_n), x_1, x_2, \ldots, x_n) \end{aligned}$$

Thus if $\mathbf{g}$ represents g and $\mathbf{h}$ represents h then $\mathbf{f}$ will represent f if

$$\begin{aligned} &\mathbf{f}\ (x_1, x_2, \ldots, x_n) = \\ &\quad (\mathbf{iszero}\ x_1 \to \mathbf{g}(x_2, \ldots, x_n) \mid \\ &\qquad\qquad \mathbf{h}(\mathbf{f}\ (\mathbf{pre}\ x_1, x_2, \ldots, x_n), \mathbf{pre}\ x_1, x_2, \ldots, x_n)) \end{aligned}$$

Using the fixed-point trick, an $\mathbf{f}$ can be constructed to satisfy this equation by defining $\mathbf{f}$ to be:

$$\begin{aligned} &\mathbf{Y}(\lambda f.\ \lambda(x_1, x_2, \ldots, x_n). \\ &\qquad (\mathbf{iszero}\ x_1 \to \mathbf{g}(x_2, \ldots, x_n) \mid \\ &\qquad\qquad \mathbf{h}(f(\mathbf{pre}\ x_1, x_2, \ldots, x_n), \mathbf{pre}\ x_1, x_2, \ldots, x_n))) \end{aligned}$$

Thus any primitive recursive function can be represented by a λ-expression.

2.7.2 The recursive functions

A function is called *recursive* if it can be constructed from 0, the successor function and the projection functions (see page 26) by a sequence of substitutions, primitive recursions and *minimizations*.

Minimization

Suppose g is a function of n arguments. We say f is defined from g by minimization if:

$$f(x_1, x_2, \ldots, x_n) \quad = \quad \text{`the smallest } y \text{ such that } g(y, x_2, \ldots, x_n){=}x_1\text{'}$$

The notation MIN(f) is used to denote the minimization of f. Functions defined by minimization may be undefined for some arguments. For example, if *one* is the function that always returns 1, i.e. $one(x) = 1$ for every x, then MIN(one) is only defined for arguments with value 1. This is obvious because if $f(x) = \mathrm{MIN}(one)(x)$, then:

$$f(x) \quad = \quad \text{`the smallest } y \text{ such that } one(y){=}x\text{'}$$

and clearly this is only defined if $x = 1$. Thus

$$\mathrm{MIN}(one)(x) = \begin{cases} \underline{0} & \text{if } x = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

To show that any recursive function can be represented in the λ-calculus it is necessary to show how to represent the minimization of an arbitrary function. Suppose **g** represents a function g of n variables and f is defined by:

$$f = \mathrm{MIN}(g)$$

Then if a λ-expression **min** can be devised such that **min** $\underline{x}$ **f** $(\underline{x}_1, \ldots, \underline{x}_n)$ represents the least number y greater than x such that

$$f(y, x_2, \ldots, x_n) = x_1$$

then **g** will represent g where:

$$\mathbf{g} \;=\; \lambda(x_1, x_2, \ldots, x_n).\ \mathbf{min}\ \underline{0}\ \mathbf{f}\ (x_1, x_2, \ldots, x_n)$$

min will clearly have the desired property if:

$$\begin{array}{l} \mathbf{min}\ x\ f\ (x_1, x_2, \ldots, x_n) = \\ \quad (\mathbf{eq}\ (f(x, x_2, \ldots, x_n))\ x_1) \rightarrow x \mid \mathbf{min}\ (\mathbf{suc}\ x)\ f\ (x_1, x_2, \ldots, x_n)) \end{array}$$

where **eq** $\underline{m}$ $\underline{n}$ is equal to **true** if $m = n$ and **false** otherwise (a suitable definition of **eq** occurs on page 21). Thus **min** can simply be defined to be:

$$\begin{array}{l} \mathbf{Y}(\lambda m. \\ \quad \lambda x\ f\ (x_1, x_2, \ldots, x_n). \\ \quad\quad (\mathbf{eq}\ (f(x, x_2, \ldots, x_n))\ x_1\ \rightarrow x \mid m\ (\mathbf{suc}\ x)\ f\ (x_1, x_2, \ldots, x_n))) \end{array}$$

Thus any recursive function can be represented by a λ-expression.

Higher-order primitive recursion

There are functions which are recursive but not primitive recursive. Here is a version of Ackermann's function, ψ, defined by:

$$\begin{array}{rcl} \psi(0, n) & = & n{+}1 \\ \psi(m{+}1, 0) & = & \psi(m, 1) \\ \psi(m{+}1, n{+}1) & = & \psi(m, \psi(m{+}1, n)) \end{array}$$

However, if one allows functions as arguments, then many more recursive functions can be defined by a primitive recursion. For example, if the higher-order function *rec* is defined by primitive recursion as follows:

$$\begin{array}{rcl} \mathit{rec}(0, x_2, x_3) & = & x_2 \\ \mathit{rec}(S(x_1), x_2, x_3) & = & x_3(\mathit{rec}(x_1, x_2, x_3)) \end{array}$$

then ψ can be defined by:

$$\psi(m, n) \;=\; \mathit{rec}\ (m,\ S,\ f \mapsto (x \mapsto \mathit{rec}(x, f(1), f)))\ (n)$$

where $x \mapsto \theta(x)$ denotes the function[1] that maps x to $\theta(x)$. Notice that the third argument of *rec* , viz. x_3, must be a function. In the definition of ψ we also took x_2 to be a function, viz. S.

[1]Note that $\lambda x.\ \theta(x)$ is an expression of the λ-calculus whereas $x \mapsto \theta(x)$ is a notation of informal mathematics.

Exercise 31
Show that the definition of ψ in terms of *rec* works, i.e. that with ψ defined as above:

$$\begin{array}{rcl}\psi(0,n) & = & n{+}1\\ \psi(m{+}1,0) & = & \psi(m,1)\\ \psi(m{+}1,n{+}1) & = & \psi(m,\psi(m{+}1,n))\end{array}$$

□

A function which takes another function as an argument, or returns another function as a result, is called *higher-order*. The example ψ shows that higher-order primitive recursion is more powerful than ordinary primitive recursion[2]. The use of operators like *rec* is one of the things that makes functional programming very powerful.

2.7.3 The partial recursive functions

A partial function is one that is not defined for all arguments. For example, the function MIN(*one*) described above is partial. Another example is the division function, since division by 0 is not defined. Functions that are defined for all arguments are called *total*.

A partial function is called *partial recursive* if it can be constructed from 0, the successor function and the projection functions by a sequence of substitutions, primitive recursions and minimizations. Thus the recursive functions are just the partial recursive functions which happen to be total. It can be shown that every partial recursive function f can be represented by a λ-expression $\underline{f}$ in the sense that

(i) $\underline{f}(\underline{x_1},\ldots,\underline{x_n}) = \underline{y}$ if $f(x_1,\ldots,x_n) = y$

(ii) If $f(x_1,\ldots,x_n)$ is undefined then $\underline{f}(\underline{x_1},\ldots,\underline{x_n})$ has no normal form.

Note that despite (ii) above, it is not in general correct to regard expressions with no normal form as being 'undefined'.

Exercise 32
Write down the λ-expression that represents MIN(f), where $f(x) = 0$ for all x. □

2.8 Extending the λ-calculus

Although it is possible to represent data-objects and data-structures with λ-expressions, it is often inefficient to do so. For example, most computers have hardware for arithmetic and it is reasonable to use this, rather than λ-conversion, to compute with numbers. A mathematically clean way of 'interfacing' computation rules to the λ-calculus is via so called *δ-rules*.

The idea is to add a set of new constants and then to specify rules, called a δ-rules, for reducing applications involving these constants. For example, one might add numerals and + as new constants, together with the δ-rule:

$$+\ m\ n \underset{\delta}{\longrightarrow} m{+}n$$

($E_1 \underset{\delta}{\longrightarrow} E_2$ means E_2 results by applying a δ-rule to some subexpression of E_1).

When adding such constants and rules to the λ-calculus one must be careful not to destroy its nice properties, e.g. the Church-Rosser theorem (see page 31).

[2]The kind of primitive recursion defined in Section 2.7.1 is *first-order* primitive recursion.

It can be shown that δ-rules are safe if they have the form:

$$c_1\ c_2\ \cdots\ c_n \underset{\delta}{\longrightarrow} e$$

where c_1, ..., c_n are constants and e is either a constant or a closed abstraction (such λ-expressions are sometimes called *values*).
For example, one might add as constants **Suc**, **Pre**, **IsZero**, Δ_0, Δ_1, Δ_2, $\cdots$ with the δ-rules:

$$\begin{array}{l}
\mathbf{Suc}\ \Delta_n \underset{\delta}{\longrightarrow} \Delta_{n+1} \\
\mathbf{Pre}\ \Delta_{n+1} \underset{\delta}{\longrightarrow} \Delta_n \\
\mathbf{IsZero}\ \Delta_0 \underset{\delta}{\longrightarrow} \mathbf{true} \\
\mathbf{IsZero}\ \Delta_{n+1} \underset{\delta}{\longrightarrow} \mathbf{false}
\end{array}$$

Here Δ_n represents the number n, **Suc**, **Pre**, **IsZero** are new constants (not defined λ-expressions like **suc**, **pre**, **iszero**), and **true** and **false** are the previously defined expressions (which are both closed abstractions).

2.9 Theorems about the λ-calculus

If $E_1 \longrightarrow E_2$ then E_2 can be thought of as having been got from E_1 by 'evaluation'. If there are no (β- or η-) redexes in E_2 then it can be thought of as 'fully evaluated'.
A λ-expression is said to be *in normal form* if it contains no β- or η-redexes (i.e. if the only conversion rule that can be applied is α-conversion). Thus a λ-expression in normal form is 'fully evaluated'.

Examples

(i) The representations of numbers are all in normal form.

(ii) $(\lambda x.\ x)\ \underline{0}$ is not in normal form.

□

Suppose an expression E is 'evaluated' in two different ways by applying two different sequences of reductions until two normal forms E_1 and E_2 are obtained. The Church-Rosser theorem stated below shows that E_1 and E_2 will be the same except for having possibly different names of bound variables.
Because the results of reductions do not depend on the order in which they are done, separate redexes can be evaluated in parallel. Various research projects are currently trying to exploit this fact by designing multiprocessor architectures for evaluating λ-expressions. It is too early to tell how successful this work will be. There is a possibility that the communication overhead of distributing redexes to different processors and then collecting together the results will cancel out the theoretical advantages of the approach. Let us hope this pessimistic possibility can be avoided. It is a remarkable fact that the Church-Rosser theorem, an obscure mathematical result dating from before computers were invented, may underpin the design of the next generation of computing systems.
Here is the statement of the Church-Rosser theorem. It is an example of something that is intuitively obvious, but very hard to prove. Many properties of the λ-calculus share this property.

The Church-Rosser theorem

If $E_1 = E_2$ then there exists an E such that $E_1 \longrightarrow E$ and $E_2 \longrightarrow E$.

It is now possible to see why the Chuch-Rosser theorem shows that λ-expressions can be evaluated in any order. Suppose an expression E is 'evaluated' in two different ways by applying two different sequences of reductions until two normal forms E_1 and E_2 are obtained. Since E_1 and E_2 are obtained from E by sequences of conversions, it follows by the definition of $=$ that $E = E_1$ and $E = E_2$ and hence $E_1 = E_2$. By the Church-Rosser theorem there exists an expression, E' say, such that $E_1 \longrightarrow E'$ and $E_2 \longrightarrow E'$. Now if E_1 and E_2 are in normal form, then the only redexes they can contain are α-redexes and so the only way that E_1 and E_2 can be reduced to E' is by changing the names of bound variables. Thus E_1 and E_2 must be the same up to renaming of bound variables (i.e. α-conversion).

Another application of the Church-Rosser theorem is to show that if $m \neq n$ then the λ-expressions representing m and n are not equal, i.e. $\underline{m} \neq \underline{n}$. Suppose $m \neq n$ but $\underline{m} = \underline{n}$; by the Church-Rosser theorem $\underline{m} \longrightarrow E$ and $\underline{n} \longrightarrow E$ for some E. But it is obvious from the definitions of $\underline{m}$ and $\underline{n}$, namely

$$\underline{m} = \lambda f\ x.\ f^m\ x$$
$$\underline{n} = \lambda f\ x.\ f^n\ x$$

that no such E can exist. The only conversions that are applicable to $\underline{\mathrm{m}}$ and $\underline{\mathrm{n}}$ are α-conversions and these cannot change the number of function applications in an expression ($\underline{m}$ contains m applications and $\underline{n}$ contains n applications).

A λ-expression E *has a normal form* if $E = E'$ for some E' in normal form. The following corollary relates expressions *in* normal form to those that *have* a normal form; it summarizes some of the statements made above.

Corollary to the Church-Rosser theorem

(i) If E has a normal form then $E \longrightarrow E'$ for some E' in normal form.

(ii) If E has a normal form and $E = E'$ then E' has a normal form.

(iii) If $E = E'$ and E and E' are both in normal form, then E and E' are identical up to α-conversion.

Proof

(i) If E has a normal form then $E = E'$ for some E' in normal form. By the Church-Rosser theorem there exists E'' such that $E \longrightarrow E''$ and $E' \longrightarrow E''$. As E' is in normal form the only redexes it can have are α-redexes, so the reduction $E' \longrightarrow E''$ must consist of a sequence of α-conversions. Thus E'' must be identical to E' except for some renaming of bound variables; it must thus be in normal form as E' is.

(ii) Suppose E has a normal form and $E = E'$. As E has a normal form, $E = E''$ where E'' is in normal form. Hence $E' = E''$ by the transitivity of $=$ (see page 8) and so E' has a normal form.

(iii) This was proved above.

□

Exercise 33
For each of the following λ-expressions *either* find its normal form *or* show that it has no normal form:

(i) $\mathbf{add}\ \underline{3}$

(ii) $\mathbf{add}\ \underline{3}\ \underline{5}$

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

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

(v) $\mathbf{Y}$

(vi) $\mathbf{Y}\ (\lambda y.\ y)$

(vii) $\mathbf{Y}\ (\lambda f\ x.\ (\mathbf{iszero}\ x \rightarrow \underline{0}\ |\ \ f\ (\mathbf{pre}\ x)))\ \underline{7}$

□

Notice that a λ-expression E might have a normal form even if there exists an infinite sequence $E \longrightarrow E_1 \longrightarrow E_2\ \cdots$. For example $(\lambda x.\ \underline{1})\ (\mathbf{Y}\ f)$ has a normal form $\underline{1}$ even though:

$$(\lambda x.\ \underline{1})\ (\mathbf{Y}\ f) \longrightarrow (\lambda x.\ \underline{1})\ (f\ (\mathbf{Y}\ f)) \longrightarrow \cdots (\lambda x.\ \underline{1})\ (f^n\ (\mathbf{Y}\ f)) \longrightarrow \cdots$$

The normalization theorem stated below tells us that such blind alleys can always be avoided by reducing the *leftmost* β- or η-redex, where by 'leftmost' is meant the redex whose beginning λ is as far to the left as possible.

Another important point to note is that E_1 may not have a normal form even though $E_1\ E_2$ does have one. For example, $\mathbf{Y}$ has no normal form, but $\mathbf{Y}\ (\lambda x.\ \underline{1}) \longrightarrow \underline{1}$. It is a common mistake to think of λ-expressions without a normal form as denoting 'undefined' functions; $\mathbf{Y}$ has no normal form but it denotes a perfectly well defined function[3]. Analysis beyond the scope of this book (see Wadsworth's paper [37]) shows that a λ-expression denotes an undefined function if and only if it *cannot* be converted to an expression in *head normal form*, where E is in head normal form if it has the form

$$\lambda V_1\ \cdots\ V_m.\ V\ E_1\ \cdots\ E_n$$

where $V_1, \ldots, V_m$ and V are variables and $E_1, \ldots, E_n$ are λ-expressions (V can either be equal to V_i, for some i, or it can be distinct from all of them). It follows that the fixed-point operator $\mathbf{Y}$ is not undefined because it can be converted to

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

which is in head normal form.

It can be shown that an expression E has a head normal form if and only if there exist expressions $E_1, \ldots, E_n$ such that $E\ E_1\ \ldots\ E_n$ has a normal form. This supports the interpretation of expressions without head normal forms as denoting undefined functions: E being undefined means that $E\ E_1\ \ldots\ E_n$ never terminates for *any* $E_1, \ldots, E_n$. Full details on head normal forms and their relation to definedness can be found in Barendregt's book [2].

[3]The mathematical characterization of the function denoted by $\mathbf{Y}$ can be found in Stoy's book [33].

The normalization theorem

If E has a normal form, then repeatedly reducing the leftmost β- or η-redex (possibly after an α-conversion to avoid invalid substitutions) will terminate with an expression in normal form.

The remark about α-conversion in the statement of the theorem is to cover cases like:

$$(\lambda x.\ (\lambda y.\ x\ y))\ y \longrightarrow \lambda y'.\ y\ y'$$

where $\lambda y.\ x\ y \longrightarrow \lambda y'.\ x\ y'$ has been α-converted so as to avoid the invalid substitution $(\lambda y.\ x\ y)[y/x] = \lambda y.\ y\ y$.

A sequence of reductions in which the leftmost redex is always reduced is called a *normal order reduction sequence.*

The normalization theorem says that if E has a normal form (i.e. for some E' in normal form $E = E'$) then it can be found by normal order reduction. This, however, is not usually the 'most efficient' way to find it. For example, normal order reduction requires

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

to be reduced to

$$\frown E \frown E \frown$$

If E is not in normal form then it would be more efficient to first reduce E to E' say (where E' is in normal form) and then to reduce

$$(\lambda x. \frown x \frown x \frown)\ E'$$

to

$$\frown E' \frown E' \frown$$

thereby avoiding having to reduce E twice.

Note, however, that this 'call-by-value' scheme is disastrous in cases like

$$(\lambda x.\underline{1})\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))$$

It is a difficult problem to find an optimal algorithm for choosing the next redex to reduce. For recent work in this area see Levy's paper [25].

Because normal order reduction appears so inefficient, some programming languages based on the λ-calculus, e.g. LISP, have used call by value even though it doesn't always terminate. Actually, call by value has other advantages besides efficiency, especially when the language is 'impure', i.e. has constructs with side effects (e.g. assignments). On the other hand, recent research suggests that maybe normal order evaluation is not as inefficient as was originally thought if one uses cunning implementation tricks like graph reduction (see page 40). Whether functional programming languages should use normal order or call by value is still a controversial issue.

2.10 Call-by-value and Y

Recall **Y**:

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

Unfortunately $\mathbf{Y}$ doesn't work with call-by-value, because applicative order causes it to go into a loop.

$$\begin{array}{ll} \mathbf{Y}\ f & \longrightarrow f(\mathbf{Y}\ f) \\ & \longrightarrow f(f(\mathbf{Y}\ f)) \\ & \longrightarrow f(f(f(\mathbf{Y}\ f))) \\ & \vdots \end{array}$$

To get around this, define:

$$\texttt{LET}\ \hat{\mathbf{Y}} = \lambda f.\ (\lambda x.\ f(\lambda y.\ x\ x\ y))\ (\lambda x.\ f(\lambda y.\ x\ x\ y))$$

Note that $\hat{\mathbf{Y}}$ is $\mathbf{Y}$ with "$x\ x$" η-converted to "$\lambda y.\ x\ x\ y$". $\hat{\mathbf{Y}}$ doesn't goes into a loop with call-by-value:

$$\hat{\mathbf{Y}}\ f \quad \longrightarrow f(\lambda y.\ \hat{\mathbf{Y}}\ f\ y)$$

Call-by-value doesn't evaluate λs, hence the looping is avoided.

Chapter 3

Combinators

Combinators provide an alternative theory of functions to the λ-calculus. They were originally introduced by logicians as a way of studying the process of substitution. More recently, Turner has argued that combinators provide a good 'machine code' into which functional programs can be compiled [34]. Several experimental computers have been built based on Turner's ideas (see e.g. [8]) and the results are promising. How these machines work is explained in Section 3.3. Combinators also provide a good intermediate code for conventional machines; several of the best compilers for functional languages are based on them (e.g. [11, 1]).

There are two equivalent ways of formulating the theory of combinators:

(i) within the λ-calculus, or

(ii) as a completely separate theory.

The approach here is to adopt (i) as it is slightly simpler, but (ii) was how it was done originally[1]. It will be shown that *any* λ-expression is equal to an expression built from variables and two particular expressions, **K** and **S**, using only function application. This is done by mimicking λ-abstractions using combinations of **K** and **S**. It will be demonstrated how β-reductions can be simulated by simpler operations involving **K** and **S**. It is these simpler operations that combinator machines implement directly in hardware. The definitions of **K** and **S** are

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

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

From these definitions it is clear by β-reduction that for all E_1, E_2 and E_3:

$$\mathbf{K}\ E_1\ E_2 = E_1$$

$$\mathbf{S}\ E_1\ E_2\ E_3 = (E_1\ E_3)\ (E_2\ E_3)$$

Any expression built by application (i.e. combination) from **K** and **S** is called a *combinator*; **K** and **S** are the *primitive combinators*.

In BNF, combinators have the following syntax:

<*combinator*> ::= **K** | **S** | (<*combinator*> <*combinator*>)

A *combinatory expression* is an expression built from **K**, **S** and zero or more variables. Thus a combinator is a combinatory expression not containing variables. In

[1] The two-volume treatise *Combinatory Logic* [9, 10] is the definitive reference, but the more recent textbooks [19, 2] are better places to start.

BNF, the syntax of combinatory expressions is:

<*combinatory expression*>
  ::= **K** | **S**
   | <*variable*>
   | (<*combinatory expression*> <*combinatory expression*>)

Exercise 34
Define **I** by:

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

Show that $\mathbf{I} = \mathbf{S}\ \mathbf{K}\ \mathbf{K}$. □

The identity function **I** defined in the last exercise is often taken as a primitive combinator, but as the exercise shows this is not necessary as it can be defined from **K** and **S**.

3.1 Combinator reduction

If E and E' are combinatory expressions then the notation $E \underset{c}{\longrightarrow} E'$ is used if $E \equiv E'$ or if E' can be got from E by a sequence of rewritings of the form:

(i) $\mathbf{K}\ E_1\ E_2 \underset{c}{\longrightarrow} E_1$

(ii) $\mathbf{S}\ E_1\ E_2\ E_3 \underset{c}{\longrightarrow} (E_1\ E_3)\ (E_2\ E_3)$

(iii) $\mathbf{I}\ E \underset{c}{\longrightarrow} E$

Note that the reduction $\mathbf{I}\ E \underset{c}{\longrightarrow} E$ is derivable from (i) and (ii).

Example

$$\begin{array}{rll} \mathbf{S}\ \mathbf{K}\ \mathbf{K}\ x & \underset{c}{\longrightarrow} \mathbf{K}\ x\ (\mathbf{K}\ x) & \text{by (ii)} \\ & \underset{c}{\longrightarrow} x & \text{by (i)} \end{array}$$

□

This example shows that for any E: $\mathbf{I}\ E \underset{c}{\longrightarrow} E$.

Any sequence of combinatory reductions, i.e. reductions via $\underset{c}{\longrightarrow}$, can be expanded into a sequence of β-conversions. This is clear because $\mathbf{K}\ E_1\ E_2$ and $\mathbf{S}\ E_1\ E_2\ E_3$ reduce to E_1 and $(E_1\ E_3)\ (E_2\ E_3)$, respectively, by sequences of β-conversions.

3.2 Functional completeness

A surprising fact is that any λ-expression can be translated to an equivalent combinatory expression. This result is called the functional completeness of combinators and is the basis for compilers for functional languages to the machine code of combinator machines.

The first step is to define, for an arbitrary variable V and combinatory expression E, another combinatory expression $\lambda^* V.\ E$ that simulates $\lambda V.\ E$ in the sense that $\lambda^* V.\ E = \lambda V.\ E$. This provides a way of using **K** and **S** to simulate adding 'λV' to an expression.

If V is a variable and E is a combinatory expression, then the combinatory expression $\lambda^* V.\ E$ is defined inductively on the structure of E as follows:

(i) $\lambda^* V.\ V = \mathbf{I}$

(ii) $\lambda^* V.\ V' = \mathbf{K}\ V' \quad$ (if $V \neq V'$)

(iii) $\lambda^* V.\ C = \mathbf{K}\ C \quad$ (if C is a combinator)

(iv) $\lambda^* V.\ (E_1\ E_2) = \mathbf{S}\ (\lambda^* V.\ E_1)\ (\lambda^* V.\ E_2)$

Note that $\lambda^* V.\ E$ is a combinatory expression not containing V.

Example: If f and x are variables and $f \neq x$, then:

$$
\begin{aligned}
\lambda^* x.\ \ f\ x &= \mathbf{S}\ (\lambda^* x.\ f)\ (\lambda^* x.\ x) \\
&= \mathbf{S}\ (\mathbf{K}\ f)\ \mathbf{I}
\end{aligned}
$$

□

The following theorem shows that $\lambda^* V.\ E$ simulates λ-abstraction.

Theorem $\quad (\lambda^* V.\ E) = \lambda V.\ E$

Proof

We show that $(\lambda^* V.\ E)\ V = E$. It then follows immediately that $\lambda V.\ (\lambda^* V.\ E)\ V = \lambda V.E$ and hence by η-reduction that $\lambda^* V.\ E = \lambda V.\ E$.

The proof that $(\lambda^* V.\ E)\ V\ =\ E$ is by mathematical induction on the 'size' of E. The argument goes as follows:

(i) If $E = V$ then:

$$(\lambda^* V.\ E)\ V \ = \ \mathbf{I}\ V \ = \ (\lambda x.\ x)\ V \ = \ V \ = \ E$$

(ii) If $E = V'$ where $V' \neq V$ then:

$$(\lambda^* V.\ E)\ V \ = \ \mathbf{K}\ V'\ V \ = \ (\lambda x\ y.\ x)\ V'\ V \ = \ V' \ = \ E$$

(iii) If $E = C$ where C is a combinator, then:

$$(\lambda^* V.\ E)\ V \ = \ \mathbf{K}\ C \ = \ (\lambda x\ y.\ x)\ C\ V \ = \ C \ = \ E$$

(iv) If $E = (E_1\ E_2)$ then we can assume by induction that:

$$
\begin{aligned}
(\lambda^* V.\ E_1)\ V = E_1 \\
(\lambda^* V.\ E_2)\ V = E_2
\end{aligned}
$$

and hence

$$
\begin{aligned}
(\lambda^* V.\ E)\ V &= (\lambda^* V.\ (E_1\ E_2))\ V \\
&= (\mathbf{S}\ (\lambda^* V.\ E_1)\ (\lambda^* V.\ E_2))\ V \\
&= (\lambda f\ g\ x.\ f\ x\ (g\ x))\ (\lambda^* V.\ E_1)\ (\lambda^* V.\ E_2)\ V \\
&= (\lambda^* V.\ E_1)\ V\ ((\lambda^* V.\ E_2)\ V) \\
&= E_1\ E_2 \qquad\qquad \text{(by induction assumption)} \\
&= E
\end{aligned}
$$

□

The notation

$$\lambda^* V_1\ V_2\ \cdots\ V_n.\ E$$

is used to mean

$$\lambda^* V_1.\ \lambda^* V_2.\ \ \cdots\ \lambda^* V_n.\ E$$

Now define the translation of an arbitrary λ-expression E to a combinatory expression $(E)_{\mathtt{C}}$:

(i) $(V)_{\mathtt{C}} = V$

(ii) $(E_1\ E_2)_{\mathtt{C}} = (E_1)_{\mathtt{C}}\ (E_2)_{\mathtt{C}}$

(iii) $(\lambda V.\ E)_{\mathtt{C}} = \lambda^* V.\ (E)_{\mathtt{C}}$

Theorem For every λ-expression E we have: $E = (E)_{\mathtt{C}}$

Proof
The proof is by induction on the size of E.

(i) If $E = V$ then $(E)_{\mathtt{C}} = (V)_{\mathtt{C}} = V$

(ii) If $E = (E_1\ E_2)$ we can assume by induction that

$$\begin{aligned} E_1 &= (E_1)_{\mathtt{C}} \\ E_2 &= (E_2)_{\mathtt{C}} \end{aligned}$$

hence

$$(E)_{\mathtt{C}}\ =\ (E_1\ E_2)_{\mathtt{C}}\ =\ (E_1)_{\mathtt{C}}\ (E_2)_{\mathtt{C}}\ =\ E_1\ E_2\ =\ E$$

(iii) If $E = \lambda V.\ E'$ then we can assume by induction that

$$(E')_{\mathtt{C}} = E'$$

hence

$$\begin{aligned} (E)_{\mathtt{C}} &= (\lambda V.\ E')_{\mathtt{C}} \\ &= \lambda^* V.\ (E')_{\mathtt{C}} && \text{(by translation rules)} \\ &= \lambda^* V.\ E' && \text{(by induction assumption)} \\ &= \lambda V.\ E' && \text{(by previous theorem)} \\ &= E \end{aligned}$$

□

This theorem shows that any λ-expression is equal to a λ-expression built up from **K** and **S** and variables by application, i.e. the class of λ-expressions E defined by the BNF:

$$E\ ::=\ V\ \mid\ \mathbf{K}\ \mid\ \mathbf{S}\ \mid\ E_1\ E_2$$

is equivalent to the full λ-calculus.

A collection of n combinators $C_1, \ldots, C_n$ is called an n-element *basis* (Barendregt [2], Chapter 8) if every λ-expression E is equal to an expression built from C_is and variables by function applications. The theorem above shows that **K** and **S** form a 2-element basis. The exercise below (from Section 8.1.5. of Barendregt) shows that there exists a 1-element basis.

Exercise 35
Find a combinator, $\mathbf{X}$ say, such that any λ-expression is equal to an expression built from $\mathbf{X}$ and variables by application. *Hint*: Let $\langle E_1, E_2, E_3\rangle = \lambda p.\ p\ E_1\ E_2\ E_3$ and consider $\langle \mathbf{K},\mathbf{S},\mathbf{K}\rangle\ \langle \mathbf{K},\mathbf{S},\mathbf{K}\rangle\ \langle \mathbf{K},\mathbf{S},\mathbf{K}\rangle$ and $\langle \mathbf{K},\mathbf{S},\mathbf{K}\rangle\ \langle\langle \mathbf{K},\mathbf{S},\mathbf{K}\rangle\ \langle \mathbf{K},\mathbf{S},\mathbf{K}\rangle\rangle$ □

Examples:

$$\begin{aligned}
\lambda^* f.\ \lambda^* x.\ f\ (x\ x) &= \lambda^* f.\ (\lambda^* x.\ f\ (x\ x))\\
&= \lambda^* f.\ (\mathbf{S}\ (\lambda^* x.\ f)\ (\lambda^* x.\ x\ x))\\
&= \lambda^* f.\ (\mathbf{S}\ (\mathbf{K} f)\ (\mathbf{S}(\lambda^* x.\ x)\ (\lambda^* x.\ x)))\\
&= \lambda^* f.\ (\mathbf{S}\ (\mathbf{K} f)\ (\mathbf{S\ I\ I}))\\
&= \mathbf{S}\ (\lambda^* f.\ \mathbf{S}\ (\mathbf{K} f))\ (\lambda^* f.\ \mathbf{S\ I\ I})\\
&= \mathbf{S}\ (\mathbf{S}\ (\lambda^* f.\ \mathbf{S})\ (\lambda^* f.\ \mathbf{K}\ f))\ (\mathbf{K}\ (\mathbf{S\ I\ I}))\\
&= \mathbf{S}\ (\mathbf{S}\ (\mathbf{K\ S})\ (\mathbf{S}\ (\lambda^* f.\ \mathbf{K})\ (\ \lambda^* f.\ f)))\ (\mathbf{K}\ (\mathbf{S\ I\ I}))\\
&= \mathbf{S}\ (\mathbf{S}\ (\mathbf{K\ S})\ (\mathbf{S}\ (\mathbf{K\ K})\ \mathbf{I}))\ (\mathbf{K}\ (\mathbf{S\ I\ I}))
\end{aligned}$$

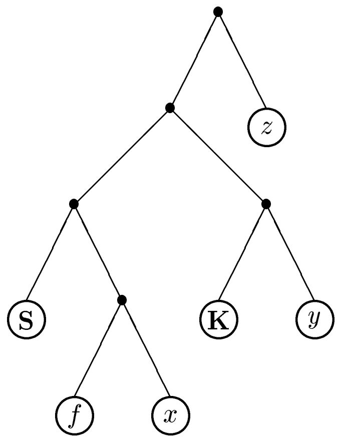
$$\begin{aligned}
(\mathbf{Y})_{\mathtt{C}} &= (\lambda f.\ (\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x)))_{\mathtt{C}}\\
&= \lambda^* f.\ ((\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x)))_{\mathtt{C}}\\
&= \lambda^* f.\ ((\lambda x.\ f(x\ x))_{\mathtt{C}}\ (\lambda x.\ f(x\ x))_{\mathtt{C}})\\
&= \lambda^* f.\ (\lambda^* x.\ (f(x\ x))_{\mathtt{C}})\ (\lambda^* x.\ (f(x\ x))_{\mathtt{C}})\\
&= \lambda^* f.\ (\lambda^* x.\ f(x\ x))\ (\lambda^* x.\ f(x\ x))\\
&= \mathbf{S}\ (\lambda^* f.\ \lambda^* x.\ f(x\ x))\ (\lambda^* f.\ \lambda^* x.\ f(x\ x))\\
&= \mathbf{S}(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{I}))(\mathbf{K}(\mathbf{SII})))(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{I}))(\mathbf{K}(\mathbf{SII})))
\end{aligned}$$

□

3.3 Reduction machines

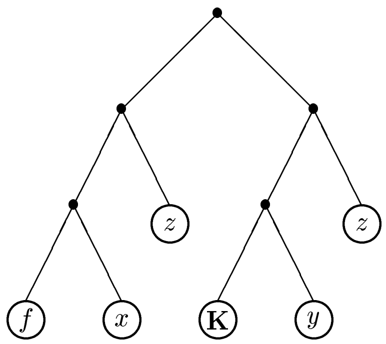
Until David Turner published his paper [34], combinators were regarded as a mathematical curiosity. In his paper Turner argued that translating functional languages, i.e. languages based on the λ-calculus, to combinators and then reducing the resulting expressions using the rewrites given on page 36 is a *practical* way of implementing these languages.

Turner's idea is to represent combinatory expressions by trees. For example, $\mathbf{S}\ (f\ x)\ (\mathbf{K}\ y)\ z$ would be represented by:

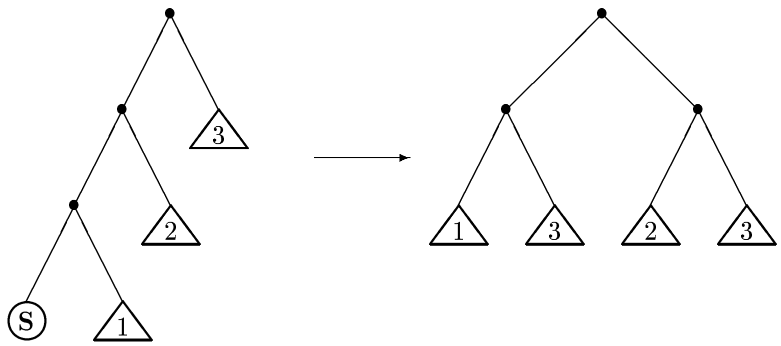


Such trees are represented as pointer structures in memory. Special hardware or firmware can then be implemented to transform such trees according to the rules of combinator reduction defining $\underset{c}{\longrightarrow}$.

For example, the tree above could be transformed to:



using the transformation

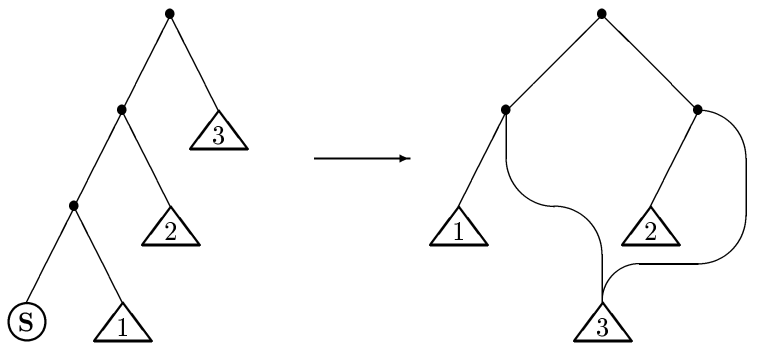


which corresponds to the reduction $\mathbf{S}\ E_1\ E_2\ E_3 \underset{c}{\longrightarrow} (E_1\ E_3)\ (E_2\ E_3)$.

Exercise 36

What tree transformation corresponds to $\mathbf{K}\ E_1\ E_2 \underset{c}{\longrightarrow} E_1$? How would this transformation change the tree above? □

Notice that the tree transformation for **S** just given duplicates a subtree. This wastes space; a better transformation would be to generate one subtree with two pointers to it, i.e.



This generates a *graph* rather than a tree. For further details of such *graph reductions* see Turner's paper [34].

It is clear from the theorem above that a valid way of reducing λ-expressions is:

(i) Translating to combinators (i.e. $E \mapsto (E)_{\mathtt{C}}$).

(ii) Applying the rewrites

$$\begin{array}{l} \mathbf{K}\ E_1\ E_2 \underset{\mathrm{c}}{\longrightarrow} E_1 \\ \mathbf{S}\ E_1\ E_2\ E_3 \underset{\mathrm{c}}{\longrightarrow} (E_1\ E_3)\ (E_2\ E_3) \end{array}$$

until no more rewriting is possible.

An interesting question is whether this process will 'fully evaluate' expressions. If some expression E is translated to combinators, then reduced using $\underset{\mathrm{c}}{\longrightarrow}$, is the resulting expression as 'fully evaluated' as the result of λ-reducing E directly, or is it only partially evaluated? Surprisingly, there doesn't seem to be anything in the literature on this important question[2]. However, combinator machines have been built and they appear to work [8]!

It is well known that if $E_1 \longrightarrow E_2$ in the λ-calculus, then it is *not* necessarily the case that $(E_1)_{\mathtt{C}} \underset{\mathrm{c}}{\longrightarrow} (E_2)_{\mathtt{C}}$. For example, take

$$\begin{array}{l} E_1 = \lambda y.\ (\lambda z.\ y)\ (x\ y) \\ E_2 = \lambda y.\ y \end{array}$$

Exercise 37

With E_1 and E_2 as above show that $E_1 \longrightarrow E_2$ in the λ-calculus, but it is not the case that $(E_1)_{\mathtt{C}} \underset{\mathrm{c}}{\longrightarrow} (E_2)_{\mathtt{C}}$. □

A combinatory expression is defined to be in *combinatory normal form* if it contains no subexpressions of the form $\mathbf{K}\ E_1\ E_2$ or $\mathbf{S}\ E_1\ E_2\ E_3$. Then the normalization theorem holds for combinatory expressions, i.e. always reducing the leftmost combinatory redex will find a combinatory normal form if it exists.

Note that if E is in combinatory normal form, then it does not necessarily follow that it is a λ-expression in normal form.

Example: $\mathbf{S}\ \mathbf{K}$ is in combinatory normal form, but it contains a β-redex, namely:

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

□

Exercise 38

Construct a combinatory expression E which is in combinatory normal form, but has no normal form. □

3.4 Improved translation to combinators

The examples on page 39 show that simple λ-expressions can translate to quite complex combinatory expressions via the rules on page 38.

To make the 'code' executed by reduction machines more compact, various optimizations have been devised.

Examples

[2]The most relevant paper I could find is one by Hindley [18]. This compares λ-reduction with combinatory reduction, but not in a way that is prima facie relevant to the termination of combinator machines.

(i) Let E be a combinatory expression and x a variable not occurring in E. Then:

$$\mathbf{S}\ (\mathbf{K}\ E)\ \mathbf{I}\ x \underset{\mathrm{c}}{\longrightarrow} (\mathbf{K}\ E\ x)\ (\mathbf{I}\ x) \underset{\mathrm{c}}{\longrightarrow} E\ x$$

hence $\mathbf{S}\ (\mathbf{K}E)\ \mathbf{I}\ x = E\ x$ (because $E_1 \underset{\mathrm{c}}{\longrightarrow} E_2$ implies $E_1 \longrightarrow E_2$), so by extensionality (Section 1.7, see on page 10):

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

(ii) Let E_1, E_2 be combinatory expressions and x a variable not occurring in either of them. Then:

$$\mathbf{S}\ (\mathbf{K}\ E_1)\ (\mathbf{K}\ E_2)\ x \underset{\mathrm{c}}{\longrightarrow} \mathbf{K}\ E_1\ x\ (\mathbf{K}\ E_2)\ x \underset{\mathrm{c}}{\longrightarrow} E_1\ E_2$$

Thus

$$\mathbf{S}\ (\mathbf{K}\ E_1)\ (\mathbf{K}\ E_2)\ x \ = \ E_1\ E_2$$

Now

$$\mathbf{K}\ (E_1\ E_2)\ x \ \underset{\mathrm{c}}{\longrightarrow}\ E_1\ E_2$$

hence $\mathbf{K}\ (E_1\ E_2)\ x = E_1\ E_2$. Thus

$$\mathbf{S}\ (\mathbf{K}\ E_1)\ (\mathbf{K}\ E_2)\ x \ = \ E_1\ E_2 \ = \ \mathbf{K}\ (E_1\ E_2)\ x$$

It follows by extensionality that:

$$\mathbf{S}\ (\mathbf{K}\ E_1)\ (\mathbf{K}\ E_2) \ = \ \mathbf{K}\ (E_1\ E_2)$$

□

Since $\mathbf{S}\ (\mathbf{K}\ E)\ \mathbf{I} = E$ for any E, whenever a combinatory expression of the form $\mathbf{S}\ (\mathbf{K}\ E)\ \mathbf{I}$ is generated, it can be 'peephole optimized' to just E. Similarly, whenever an expression of the form $\mathbf{S}\ (\mathbf{K}\ E_1)\ (\mathbf{K}\ E_2)$ is generated, it can be optimized to $\mathbf{K}\ (E_1\ E_2)$.

Example: On page 39 it was shown that:

$$\lambda^* f.\ \lambda^* x.\ f(x\ x) = \mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ \mathbf{S})\ (\mathbf{S}\ (\mathbf{K}\ \mathbf{K})\ \mathbf{I}))\ (\mathbf{K}\ (\mathbf{S}\ \mathbf{I}\ \mathbf{I}))$$

Using the optimization $\mathbf{S}\ (\mathbf{K}\ E)\ \mathbf{I} = E$ this simplifies to:

$$\lambda^* f.\ \lambda^* x.\ f(x\ x) = \mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ \mathbf{S})\ \mathbf{K})\ (\mathbf{K}\ (\mathbf{S}\ \mathbf{I}\ \mathbf{I}))$$

□

3.5 More combinators

It is easier to recognize the applicability of the optimization $\mathbf{S}\ (\mathbf{K}\ E)\ \mathbf{I} = E$ if $\mathbf{I}$ has not been expanded to $\mathbf{S}\ \mathbf{K}\ \mathbf{K}$, i.e. if $\mathbf{I}$ is taken as a primitive combinator. Various other combinators are also useful in the same way; for example, $\mathbf{B}$ and $\mathbf{C}$ defined by:

$$\texttt{LET}\ \mathbf{B} = \lambda f\ g\ x.\ f\ (g\ x)$$

$$\texttt{LET}\ \mathbf{C} = \lambda f\ g\ x.\ f\ x\ g$$

These have the following reduction rules:

$$\mathbf{B}\ E_1\ E_2\ E_3 \underset{c}{\longrightarrow} E_1\ (E_2\ E_3)$$
$$\mathbf{C}\ E_1\ E_2\ E_3 \underset{c}{\longrightarrow} E_1\ E_3\ E_2$$

Exercise 39
Show that with **B**, **C** defined as above:

$$\mathbf{S}\ (\mathbf{K}\ E_1)\ E_2 = \mathbf{B}\ E_1\ E_2$$
$$\mathbf{S}\ E_1\ (\mathbf{K}\ E_2) = \mathbf{C}\ E_1\ E_2$$

(where E_1, E_2 are any two combinatory expressions). □

Using **B** and **C**, one can further optimize the translation of λ-expressions to combinators by replacing expressions of the form $\mathbf{S}\ (\mathbf{K}\ E_1)\ E_2$ and $\mathbf{S}\ E_1\ (\mathbf{K}\ E_2)$ by $\mathbf{B}\ E_1\ E_2$ and $\mathbf{C}\ E_1\ E_2$.

3.6 Curry's algorithm

Combining the various optimizations described in the previous section leads to *Curry's algorithm* for translating λ-expressions to combinatory expressions. This algorithm consists in using the definition of $(E)_{\mathsf{C}}$ given on page 38, but whenever an expression of the form $\mathbf{S}\ E_1\ E_2$ is generated one tries to apply the following rewrite rules:

1. $\mathbf{S}\ (\mathbf{K}\ E_1)\ (\mathbf{K}\ E_2) \longrightarrow \mathbf{K}\ (E_1\ E_2)$
2. $\mathbf{S}\ (\mathbf{K}\ E)\ \mathbf{I} \longrightarrow E$
3. $\mathbf{S}\ (\mathbf{K}\ E_1)\ E_2 \longrightarrow \mathbf{B}\ E_1\ E_2$
4. $\mathbf{S}\ E_1\ (\mathbf{K}\ E_2) \longrightarrow \mathbf{C}\ E_1\ E_2$

If more than one rule is applicable, the *earlier* one is used. For example, $\mathbf{S}\ (\mathbf{K}\ E_1)\ (\mathbf{K}\ E_2)$ is translated to $\mathbf{K}\ (E_1\ E_2)$, not to $\mathbf{B}\ E_1\ (\mathbf{K}\ E_2)$.

Exercise 40
Show that using Curry's algorithm, **Y** is translated to the combinator:

$$\mathbf{S}\ (\mathbf{C}\ \mathbf{B}\ (\mathbf{S}\ \mathbf{I}\ \mathbf{I}))\ (\mathbf{C}\ \mathbf{B}\ (\mathbf{S}\ \mathbf{I}\ \mathbf{I}))$$

□

Exercise 41
Show that:

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

□

3.7 Turner's algorithm

In a second paper, Turner proposed that Curry's algorithm be extended to use another new primitive combinator called $\mathbf{S}'$ [35]. This is defined by:

$$\texttt{LET}\ \mathbf{S}' = \lambda c\ f\ g\ x.\ c\ (f\ x)\ (g\ x)$$

and has the reduction rule:

$$\mathbf{S}'\ C\ E_1\ E_2\ E_3 \underset{\mathrm{c}}{\longrightarrow} C\ (E_1\ E_3)\ (E_2\ E_3)$$

where C, E_1, E_2, E_3 are arbitrary combinatory expressions. The reason why 'C' is used is that $\mathbf{S}'$ has the property that *if C is a combinator* (i.e. contains no variables), then for any E_1 and E_2:

$$\lambda^* x.\ C\ E_1\ E_2 = \mathbf{S}'\ C\ (\lambda^* x.\ E_1)\ (\lambda^* x.\ E_2)$$

This can be shown using extensionality. Clearly x is a variable not occurring in $\lambda^* x.\ C\ E_1\ E_2$ or $\mathbf{S}'\ C\ (\lambda^* x.\ E_1)\ (\lambda^* x.\ E_2)$ (exercise: why?), so it is sufficient to show:

$$(\lambda^* x.\ C\ E_1\ E_2)\ x = (\mathbf{S}'\ C\ (\lambda^* x.\ E_1)\ (\lambda^* x.\ E_2))\ x$$

From the definition of $\lambda^* x$ it easily follows that:

$$\lambda^* x.\ C\ E_1\ E_2 = \mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ C)\ (\lambda^* x.\ E_1))\ (\lambda^* x.\ E_2)$$

hence

$$\begin{aligned}(\lambda^* x.\ C\ E_1\ E_2)\ x &= (\mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ C)\ (\lambda^* x.\ E_1))\ (\lambda^* x.\ E_2)\ x \\ &= \mathbf{S}\ (\mathbf{K}\ C)\ (\lambda^* x.\ E_1)\ x\ ((\lambda^* x.\ E_2))\ x) \\ &= \mathbf{K}\ C\ x\ ((\lambda^* x.\ E_1)\ x)\ ((\lambda^* x.\ E_2)\ x) \\ &= C\ ((\lambda^* x.\ E_1)\ x)\ ((\lambda^* x.\ E_2))\ x)\end{aligned}$$

But $(\mathbf{S}'\ C\ (\lambda^* x.\ E_1)\ (\lambda^* x.\ E_2)\ x = C\ ((\lambda^* x.\ E_1)\ x)\ ((\lambda^* x.\ E_2))\ x)$ also, and so:

$$(\lambda^* x.\ C\ E_1\ E_2)\ x = (\mathbf{S}'\ C\ (\lambda^* x.\ E_1)\ (\lambda^* x.\ E_2))\ x$$

Exercise 42
Where in the argument above did we use the assumption that C is a combinator?
□

Turner's combinator $\mathbf{S}'$ is useful when translating λ-expressions of the form $\lambda V_n\ \cdots\ V_2\ V_1.\ E_1\ E_2$ (it will be seen shortly why it is convenient to number the bound variables in descending order). To see this, following Turner [35], temporarily define

E'	to mean	$\lambda^* V_1.\ E$
E''	to mean	$\lambda^* V_2.\ (\lambda^* V_1.\ E)$
E'''	to mean	$\lambda^* V_3.\ (\lambda^* V_2.\ (\lambda^* V_1.\ E))$
$\vdots$		

Recall that:

$$(\lambda V_n\ \cdots\ V_2\ V_1.\ E_1\ E_2)_{\mathtt{C}} = \lambda^* V_n.\ (\ \cdots\ (\lambda^* V_2.\ (\lambda^* V_1.\ (E_1\ E_2)_{\mathtt{C}})))\ \cdots\ )$$

The next exercise shows that:

$$\lambda^* V_n.\ \ldots \lambda^* V_2.\ \lambda^* V_1.\ (E_1\ E_2)$$

gets very complicated as n increases.

Exercise 43
Show that:

(i) $\lambda^* x_1.\ E_1\ E_2 = \mathbf{S}\ E_1'\ E_2'$

(ii) $\lambda^* x_2.\ (\lambda^* x_1.\ E_1\ E_2) = \mathbf{S}\ (\mathbf{B}\ \mathbf{S}\ E_1'')\ E_2''$

(iii) $\lambda^* x_3.\ (\lambda^* x_2.\ (\lambda^* x_1.\ E_1\ E_2)) = \mathbf{S}\ (\mathbf{B}\ \mathbf{S}\ (\mathbf{B}\ (\mathbf{B}\ \mathbf{S})\ E_1'''))\ E_2'''$

(iv) $\lambda^* x_4.\ (\lambda^* x_3.\ (\lambda^* x_2.\ (\lambda^* x_1.\ E_1\ E_2))) =$
$\mathbf{S}\ (\mathbf{B}\ \mathbf{S}\ (\mathbf{B}\ (\mathbf{B}\ \mathbf{S})\ (\mathbf{B}\ (\mathbf{B}\ (\mathbf{B}\ \mathbf{S})))\ E_1''''))\ E_2''''$

□

The size of $\lambda^* V_n.\ \ldots \lambda^* V_2.\ \lambda^* V_1.\ (E_1\ E_2)$ is proportional to the *square* of n. Using $\mathbf{S}'$, the size can be made to grow *linearly* with n:

$$\begin{array}{rl}
\lambda^* x_2.\ (\lambda^* x_1.\ E_1\ E_2) & = \lambda^* x_2.\ \mathbf{S}\ E_1'\ E_2' \\
& = \mathbf{S}'\ \mathbf{S}\ (\lambda^* x_2.\ E_1')\ (\lambda^* x_2.\ E_2') \\
& = \mathbf{S}'\ \mathbf{S}\ E_1''\ E_2'' \\[1ex]
\lambda^* x_3.\ (\lambda^* x_2.\ (\lambda^* x_1.\ E_1\ E_2)) & = \lambda^* x_3.\ \mathbf{S}'\ \mathbf{S}\ E_1''\ E_2'' \\
& = \mathbf{S}'\ (\mathbf{S}'\ \mathbf{S})\ (\lambda^* x_3.\ E_1'')\ (\lambda^* x_3.\ E_2'') \\
& = \mathbf{S}'\ (\mathbf{S}'\ \mathbf{S})\ E_1'''\ E_2''' \\[1ex]
\lambda^* x_4.\ (\lambda^* x_3.\ (\lambda^* x_2.\ (\lambda^* x_1.\ E_1\ E_2))) & = \lambda^* x_4.\ \mathbf{S}'\ (\mathbf{S}'\ \mathbf{S})\ E_1'''\ E_2''' \\
& = \mathbf{S}'\ (\mathbf{S}'\ (\mathbf{S}'\ \mathbf{S}))\ (\lambda^* x_4.\ E_1''')\ (\lambda^* x_4.\ E_2''') \\
& = \mathbf{S}'\ (\mathbf{S}'\ (\mathbf{S}'\ \mathbf{S}))\ E_1''''\ E_2''''
\end{array}$$

Just as $\mathbf{B}$ and $\mathbf{C}$ were introduced to simplify combinatory expressions of the form $\mathbf{S}\ (\mathbf{K}\ E_1)\ E_2$ and $\mathbf{S}\ E_1\ (\mathbf{K}\ E_2)$ respectively, Turner also devised $\mathbf{B}'$ and $\mathbf{C}'$ with an analogous role for $\mathbf{S}'$. The properties required are:

$$\mathbf{S}'\ C\ (\mathbf{K}\ E_1)\ E_2 = \mathbf{B}'\ C\ E_1\ E_2$$
$$\mathbf{S}'\ C\ E_1\ (\mathbf{K}\ E_2) = \mathbf{C}'\ C\ E_1\ E_2$$

(where C is any combinator, and E_1, E_2 are arbitrary combinatory expressions). This is achieved if $\mathbf{B}'$ and $\mathbf{C}'$ are defined by:

$$\texttt{LET}\ \mathbf{B}' = \lambda c\ f\ g\ x.\ c\ f\ (g\ x)$$

$$\texttt{LET}\ \mathbf{C}' = \lambda c\ f\ g\ x.\ c\ (f\ x)\ g$$

Clearly $\mathbf{B}'$ and $\mathbf{C}'$ will have the property that for arbitrary λ-expressions C, E_1, E_2 and E_3:

$$\mathbf{B}'\ C\ E_1\ E_2\ E_3 \underset{c}{\longrightarrow} C\ E_1\ (E_2\ E_3)$$
$$\mathbf{C}'\ C\ E_1\ E_2\ E_3 \underset{c}{\longrightarrow} C\ (E_1\ E_3)\ E_2$$

Exercise 44
Show that for arbitrary λ-expressions E_1, E_2 and E_3:

(i) $\mathbf{S}'\ E_1\ (\mathbf{K}\ E_2)\ E_3 = \mathbf{B}'\ E_1\ E_2\ E_3$

(ii) $\mathbf{S}'\ E_1\ E_2\ (\mathbf{K}\ E_3) = \mathbf{C}'\ E_1\ E_2\ E_3$

(iii) $\mathbf{S}\ (\mathbf{B}\ E_1\ E_2)\ E_3 = \mathbf{S}'\ E_1\ E_2\ E_3$

(iv) $\mathbf{B}\ (E_1\ E_2)\ E_3 = \mathbf{B}'\ E_1\ E_2\ E_3$

(v) $\mathbf{C}\ (\mathbf{B}\ E_1\ E_2)\ E_3 = \mathbf{C}'\ E_1\ E_2\ E_3$

□

Turner's algorithm for translating λ-expressions to combinatory expressions is described by him [35] as follows:

> Use the algorithm of Curry but whenever a term beginning in **S**, **B** or **C** is formed use one of the following transformations if it is possible to do so
>
> $$\mathbf{S}\ (\mathbf{B}\ K\ A)\ B \longrightarrow \mathbf{S}'\ K\ A\ B,$$
>
> $$\mathbf{B}\ (K\ A)\ B \longrightarrow \mathbf{B}'\ K\ A\ B,$$
>
> $$\mathbf{C}\ (\mathbf{B}\ K\ A)\ B \longrightarrow \mathbf{C}'\ K\ A\ B.$$
>
> Here A and B stand for arbitrary terms as usual and K is any term composed entirely of constants. The correctness of the new algorithm can be inferred from the correctness of the Curry algorithm by demonstrating that in each of the above transformations the left- and right-hand sides are extensionally equal. In each case this follows directly from the definitions of the combinators involved.

Since Turner's pioneering papers appeared, many people have worked on improving the basic idea. For example, John Hughes has devised a scheme for dynamically generating an 'optimal' set of primitive combinators (called *supercombinators*) for each program [20]. The idea is that the compiler will generate combinatory expressions built out of the supercombinators for the program being compiled. It will also dynamically produce 'microcode' to implement the reduction rules for these supercombinators. The result is that each program runs on a reduction machine tailored specially for it. Most current high-performance implementations of functional languages use supercombinators [1, 11]. Another avenue of research is to use combinators based on the De Bruijn notation briefly described on page 11. The 'Categorical Abstract Machine' [26] uses this approach.

Chapter 4

A Quick Overview of ML

There are two widely use descendents of the original ML: Standard ML and Caml[1]. These notes[2] describe the former. Several implementations of Standard ML exist. These all support the same core language, but differ in extensions, error message details etc. AT&T's public domain "Standard ML of New Jersey" (SML/NJ) and the commercial system PolyML[3] are used for research applications in the Cambridge Computer Laboratory. The ML implementation on Thor for teaching is "Edinburgh ML" from the University of Edinburgh (with enhancements due to Arthur Norman of Cambridge). The different outputs produced by SML/NJ and Edinburgh ML will be sometimes shown, but the examples that follow are presented in the system neutral style of Paulson's book (which is closer to Edinburgh ML than SML/NJ).

4.1 Interacting with ML

ML is an interactive language. A common way to run it is inside a shell window from emacs. The programs are then tested by 'cutting and pasting' from the text window to the shell window.

The two main things one does in ML are evaluate expressions and perform declarations.

What follows is a session in which simple uses of various ML constructs are illustrated. To make the session easier to follow, it is split into a sequence of boxed sub-sessions.

4.2 Expressions

The top-level ML prompt is "`-`". As ML reads a phrase it prompts with "`=`" until a complete expression or declaration is found. Neither the initial prompt `-` nor the intermediate prompt `=` will normally be shown here, except in sessions which are included to illustrate the behaviour of particular ML implementations (e.g. the next two boxes).

SML/NJ is called "sml" on Computer Lab machines. The following seesion shows it being run and the expression `2+3` being evaluated.

[1]Readers interested in Caml should consult the Web page `http://pauillac.inria.fr/caml/`. Caml is a lightweight language better suited than Standard ML for use on small machines. All the constructs described in this course are in Caml, though the syntactic details differ slightly from Standard ML

[2]This overview has evolved from the description of the original ML in Section 2.1 of:

> M.J.C. Gordon, A.J.R.G Milner and C.P. Wadsworth *Edinburgh LCF: A Mechanized Logic of Computation*, Lecture Notes in Computer Science **78**, Springer-Verlag 1979.

[3]PolyML was originally developed at the Cambridge Computer Laboratory and then licenced first to Imperial Software Technology and then to Abstract Hardware Limited. It has an integrated persistant storage system (database) and is less memory hungry than Standard ML of New Jersey.

```
woodcock% sml                                                          1
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit

-2+3;
val it = 5 : int

-it;
val it = 5 : int
```

After SML/NJ starts up it prints a message followed by `val it = () : unit`, (this will be explained later). It then prompts for user input with `-`, the user then input `2+3;` followed by a carriage return; ML then responded with `val it = 5 : int`, a new line, and then prompted again. This output shows that `2+3` evaluates to the value `5` of type `int`.

The user then input `it;` followed by a carriage return, and the system responded with `val it = 5 : int` again. In general, to evaluate an expression e one inputs e followed by a semi-colon and then a carriage return; the system then prints e's value and type in the format shown. The value of the last expression evaluated at top level is remembered in the identifier `it`. This is shown explicitly in the output from SML/NJ, but not in the output from Edinburgh ML shown in the following box (which, after Edinburgh ML has been run, has the same input as the preceding one).

```
hammer.thor.cam.ac.uk% /group/clteach/acn/ml/unix/cml                  2
FAM /group/clteach/acn/ml/unix/fam started on 02-Jan-1996 16:03:07
   (version 4.2.01 of Jan 25 1995)
Image file /group/clteach/acn/ml/unix/cml.exp
   (written on 25-Jan-1995 15:42:47 by FAM version 4.2.01)
[Loading Generic Heap...resexing...relocating by efff1ff8 (bytes)]

Edinburgh ML for DOS/Win32s/Unix  (C) Edinburgh University & A C Norman

- 2+3;
> 5 : int

- it;
> 5 : int
```

Unless explicity indicated otherwise, the boxed sessions that follow use the format illustrated by:

```
2+3;                                                                   3
> val it = 5 : int

it;
> val it = 5 : int
```

Prompts (`-`) are not shown, system output is indicated by `>` and the values of expressions are shown explicitly bound to `it`. Sometimes part of the output will be omitted (e.g. the type).

4.3 Declarations

The declaration `val` x`=`e evaluates e and binds the resulting value to x.

```
val x=2*3;                                                    4
> val x = 6 : int

it=x;
> val it = false : bool
```

Notice that declarations do not affect `it`.

Inputting e`;` at top level is actually treated as inputting the declaration `let it = e;`. The ML system (both SML/NJ and Edinburgh ML) initially binds `it` to a special value `()`, which is the only value of the one-element type `unit`.

To bind the variables $x_1, \ldots, x_n$ simultaneously to the values of the expressions $e_1, \ldots, e_n$ one can perform:

- either the declaration `val` x_1`=`e_1 `and` x_2`=`$e_2 \ldots$ `and` x_n`=`e_n
- or `val (`$x_1, x_2, \ldots, x_n$`)=(`$e_1, e_2, \ldots, e_n$`)`.

These two declarations are equivalent.

```
val y=10 and z=x;                                             5
> val y = 10 : int
> val z = 6 : int

val (x,y) = (y,x);
> val x = 10 : int
> val y = 6 : int
```

A declaration d can be made local to the evaluation of an expression e by evaluating the expression `let` d `in` e `end`.

```
let val x=2 in x*y end;                                       6
> val it = 12 : int

x;
> val it = 10 : int
```

4.4 Comments

Comments start with `(*` and end with `*)`. They nest like parentheses, can extend over many lines and can be inserted wherever spaces are allowed.

```
tr(* comments can't go in the middle of names *)ue;           7
> Error: unbound variable or constructor: tr
> Error: unbound variable or constructor: ue

1 (* this comment is ignored *) < 2;
> val it = true : bool

(* Inside this comment (* another one is nested *) ! *)
```

4.5 Functions

To define a function f with formal parameter x and body e one performs the declaration: `fun` f x `=` e. To apply the function f to an actual parameter e one evaluates the expression: f e.

```
fun f x = 2*x;                                                        8
> val f = fn : int -> int

f 4;
> val it = 8 : int
```

Functions are printed as `fn` in SML/NJ and `Fn` in Edinburgh ML, since a function as such is not printable. After `fn` or `Fn` is printed, the type of the function is also printed. Functions are printed as `fn` in these notes.

Applying a function to an argument of the wrong type results in a typechecking error. The particular error message depends on the ML system used. In SML/NJ:

```
- f true;                                                             9
std_in:12.1-12.6 Error: operator and operand don't agree (tycon mismatch)
  operator domain: int
  operand:         bool
  in expression:
    f true
```

In Edinburgh ML:

```
- f true;                                                             10
Type clash  in:  (f true)
Looking  for a:  int
I have found a:  bool
```

Application binds more tightly than anything else in the language; thus, for example, f `3 + 4` means `(`f `3)+4` not f`(3+4)`. Functions of several arguments can be defined:

```
fun add (x:int) (y:int) = x+y;                                        11
> val add = fn : int -> int -> int

add 3 4;
> val it = 7 : int

val f = add 3;
> val f = fn : int -> int

f 4;
> val it = 7 : int
```

Application associates to the left, so `add 3 4` means `(add 3)4`. In the expression `add 3`, the function `add` is applied to `3`; the resulting value is the function of type `int -> int` which adds `3` to its argument. Thus `add` takes its arguments 'one at a time'.

Without the explicit typing of the formal parameters, ML cannot tell whether the `+` is addition of integers or reals. The symbol `+` is overloaded. If the extra type information is omitted, an error results. In SML/NJ:

```
- fun add x y = x+y;                                                  12
std_in:5.16 Error: overloaded variable "+" cannot be resolved
```

In Edinburgh ML:

```
- fun add x y = x+y;                                                  13
Type checking error in: (syntactic context unknown)
Unresolvable overloaded identifier:  +
Definition cannot be found for the type:  ('a * 'a) -> 'a
```

This kind of typechecking error is relatively rare. Much more common are errors resulting from applying functions to arguments of the wrong type.

The function `add` could alternatively have been defined to take a single argument of the product type `int * int`:

```
fun add(x,y):int = x+y;                                                    14
> val add = fn : int * int -> int

add(3,4);
> val it = 7 : int

let val z = (3,4) in add z end;
> val it = 7 : int

add 3;
> std_in:2.1-2.5 Error: operator and operand don't agree (tycon mismatch)
>   operator domain: int * int
>   operand:         int
>   in expression:
>     add 3
```

The error message shown here is the one generated by SML/NJ. Notice that this time the result of the function has had its type given explicitly. In general, it is sufficient to explicitly type any subexpression as long as this disambiguates all overloaded operators.

As well as taking structured arguments (e.g. `(3,4)`) functions may also return structured results.

```
fun sumdiff(x:int,y:int) = (x+y,x-y);                                      15
> val sumdiff = fn : int * int -> int * int

sumdiff(3,4);
> val it = (7,~1) : int * int
```

4.6 Type abbreviations

Types can be given names:

```
type intpair = int * int;                                                  16
> type intpair defined

fun addpair ((x,y):intpair) = x+y;
> val addpair = fn : intpair -> int

(3,5);
> val it = (3,5) : int * int

(3,5):intpair;
> val it = (3,5) : intpair

addpair(3,5);
> val it = 8 : int
```

The new name is simply an abbreviation; `intpair` and `int*int` are completely equivalent.

4.7 Operators

`+` (addition) and `*` are built-in infix operators. Users can define their own infixes using `infix` (for left associative operators) and `infixr` for right associative ones.

```
infix op1;                                                              17
infixr op2;
> infix op1
> infixr op2
```

This merely tells the parser to parse e_1 `op1` e_2 as `op1(`e_1`,`e_2`)` and e_1 `op2` e_2 as `op2(`e_1`,`e_2`)`.

```
fun (x:int) op1 (y:int) = x + y;                                        18
> val op1 = fn : int * int -> int

1 op1 2;
> val it = 3 : int

fun (x:int) op2 (y:int) = x * y;
> val op2 = fn : int * int -> int

2 op2 3;
> val it = 6 : int
```

An infix of precedence n can be created by using `infix` n instead of just `infix` (and `infixr` n instead of just `infixr`). If the n is omitted a default precedence of `0` is assumed.
The ML parser can be told to ignore the infix status of an occurrence of an identifier by preceding the occurrence with `op`.

```
op1;                                                                    19
> Error: nonfix identifier required

op op1;
> val it = fn : int * int -> int
```

The infix status of an operator can be permanently removed using the directive `nonfix`.

```
1 + 2;                                                                  20
> val it = 3 : int

nonfix +;
> nonfix +

1 + 2;
> Error: operator is not a function
>   operator: int
>   in expression:
>     1 + : overloaded
```

Removing the infix status of built-in operators is not recommended. Let's restore it before chaos results: `+` is left-associative with precedence `6`.

```
infix 6 +;                                                              21
> infix 6 +
```

4.8 Lists

If $e_1, \ldots, e_n$ all have type ty then the ML expression `[`e_1`,...,`e_n`]` has type `(`ty `list)`. The standard functions on lists are `hd` (head), `tl` (tail), `null` (which tests whether a list is empty—i.e. is equal to `[]`), and the infixed operators `::` (cons) and `@` (append, or concatenation).

```
val m = [1,2,(2+1),4];                                                    22
> val m = [1,2,3,4] : int list

(hd m , tl m);
> val it = (1,[2,3,4]) : int * int list

(null m , null []);
> val it = (false,true) : bool * bool

0::m;
> val it = [0,1,2,3,4] : int list

[1, 2] @ [3, 4, 5, 6];
> val it = [1,2,3,4,5,6] : int list

[1,true,2];
> std_in:3.1-3.10 Error: operator and operand don't agree (tycon mismatch)
>   operator domain: bool * bool list
>   operand:         bool * int list
>   in expression:
>     true :: 2 :: nil
```

All the members of a list must have the same type (the error message shown is from SML/NJ).

4.9 Strings

A sequence of characters enclosed between quotes (`"`) is a string.

```
"this is a string";                                                       23
> val it = "this is a string" : string

"";
> val it = "" : string
```

The empty string is `""`. A string can be 'exploded' into a list of single-character strings with the function `explode`. The inverse of this is `implode`, which concatenates a list of single-character strings into a single string.

```
explode;                                                                  24
> val it = fn : string -> string list

explode "this is a string";
> val it =
>   ["t","h","i","s"," ","i","s"," ","a"," ","s","t","r","i","n","g"]
>   : string list

implode it;
> val it = "this is a string" : string
```

4.10 Records

Records are data-structures with named components. They can be contrasted with tuples whose components are determined by position.

A record with fields $x_1, \ldots, x_n$ whose values are $v_1, \ldots, v_n$ is created by evaluating the expression: {x_1=v_1, ... , x_n=v_n}.

```
val MikeData =                                                          25
 {userid = "mjcg", sex = "male", married = true, children = 2};
> val MikeData = {children=2,married=true,sex="male",userid="mjcg"}
>   : {children:int, married:bool, sex:string, userid:string}
```

The type of {x_1=v_1, ... , x_n=v_n} is {x_1:σ_1, ... , x_n:σ_n}, where σ_i is the type of v_i. The order in which record components are named does not matter:

```
val MikeData' =                                                         26
 {sex = "male", userid = "mjcg", children = 2, married = true};
> val MikeData' = {children=2,married=true,sex="male",userid="mjcg"}
>   : {children:int, married:bool, sex:string, userid:string}

MikeData = MikeData';
> val it = true : bool
```

The component named x of a record can be extracted using the special operator #x.

```
#children MikeData;                                                     27
> val it = 2 : int
```

Functions which access record components need to be explicitly told the type of the record they are accessing, since there may be several types of records around with the same field names.

```
fun Sex p = #sex p;                                                     28
> Error: unresolved flex record in let pattern

type persondata = {userid:string, children:int, married:bool, sex:string};
> type persondata = {children:int, married:bool, sex:string, userid:string}

fun Sex(p:persondata) = #sex p;
> val Sex = fn : persondata -> string
```

A tuple (v_1, ... , v_n) is equivalent to the record {1=v_1, ... , n=v_n} (i.e. tuples in ML are special cases of records).

```
{1 = "Hello", 2 = true, 3 = 0};                                         29
> val it = ("Hello",true,0) : string * bool * int

#2 it;
> val it = true : bool
```

4.11 Polymorphism

The list processing functions `hd`, `tl` etc. can be used on all types of lists.

```
hd [1,2,3];                                                        30
> val it = 1 : int

hd [true,false,true];
> val it = true : bool

hd [(1,2),(3,4)];
> val it = (1,2) : int * int
```

Thus `hd` has several types: above it is used with types `(int list) -> int`, `(bool list) -> bool` and `(int * int) list -> (int * int)`. In fact if *ty* is any type then `hd` has the type `(`*ty* `list) ->` *ty*. Functions, like `hd`, with many types are called *polymorphic*, and ML uses type variables `'a`, `'b`, `'c` etc. to represent their types.

```
hd;                                                                31
> val it = fn : 'a list -> 'a
```

The ML function `map` takes a function f (with argument type `'a` and result type `'b`), and a list l (of elements of type `'a`), and returns the list obtained by applying f to each element of l (which is a list of elements of type `'b`).

```
map;                                                               32
> val map = fn : ('a -> 'b) -> 'a list -> 'b list

fun add1 (x:int) = x+1;
> val add1 = fn : int -> int

map add1 [1,2,3,4,5];
> val it = [2,3,4,5,6] : int list
```

`map` can be used at any instance of its type: above, both `'a` and `'b` were instantiated to `int`; below, `'a` is instantiated to `(int list)` and `'b` to `bool`. Notice that the instance need not be specified; it is determined by the type checker.

```
map null [[1,2], [], [3], []];                                     33
> val it = [false,true,false,true] : bool list
```

A useful built-in operator is function composition `o`

```
op o;                                                              34
> val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b

fun add1 n = n+1
and add2 n = n+2;
> val add1 = fn : int -> int
> val add2 = fn : int -> int

(add1 o add2) 5;
> val it = 8 : int
```

4.12 fn-expressions

The expression `fn` x `=>` e evaluates to a function with formal parameter x and with body e. Thus the declaration `fun` f x `=` e is equivalent to `val` f `= fn` x `=>` e. Similarly `fun` f`(`x`,`y`)`z `=` e is equivalent to `val` f `= fn (`x`,`y`) => fn` z `=>` e. In the theory of functions, the symbol λ is used instead of `fn`; expressions like `fn` x `=>` e are sometimes called λ-expressions, because they correspond to λ-calulus abstractions $\lambda x.e$ (see Chapter 1).

```
fn x => x+1;                                                    35
> val it = fn : int -> int

it 3;
> val it = 4 : int
```

The higher order function `map` applies a function to each element of a list in turn and returns the list of results.

```
map (fn x => x*x) [1,2,3,4];                                    36
> val it = [1,4,9,16] : int list

val doubleup = map (fn x => x@x);
> val doubleup = fn : 'a list list -> 'a list list

doubleup [ [1,2], [3,4,5] ];
> val it = [[1,2,1,2],[3,4,5,3,4,5]] : int list list

doubleup [];
> val it = [] : 'a list list
```

4.13 Conditionals

ML has conditionals with syntax `if` e `then` e_1 `else` e_2 with the expected meaning. The truthvalues are `true` and `false`, both of type `bool`.

```
if true then 1 else 2;                                          37
> val it = 1 : int

if 2<1 then 1 else 2;
> val it = 2 : int
```

e_1 `orelse` e_2 abbreviates `if` e_1 `then true else` e_2 and e_1 `andalso` e_2 abbreviates `if` e_1 `then` e_2 `else false`.

4.14 Recursion

The following defines the factorial function:

```
fun fact n = if n=0 then 1 else n*fact(n-1);                    38
> val fact = fn : int -> int

fact 5;
> val it = 120 : int
```

Notice that the compiler automatically detects recursive calls. In earlier versions of ML, recursion had to be explicitly indicated.
Consider:

```
fun f n : int = n+1;                                            39
> val f = fn : int -> int

fun f n = if n=0 then 1 else n*f(n-1);
> val f = fn : int -> int

f 3;
> val it = 6 : int
```

Here `f 3` results in the evaluation of `3*f(2)`. In earlier versions of ML, the first `f` would have been used, so that `f(2)` would have evaluated to `2+1=3`, hence the expression `f 3` would have evaluated to `3*3=9`.

An alternative style of defining functions in Standard ML that avoids enforced recursion uses `val` and `fn`.

40

```
fun f n : int = n+1;
> val f = fn : int -> int

val f = fn n => if n=0 then 1 else n*f(n-1);
> val f = fn : int -> int

f 3;
> val it = 9 : int
```

Here, the occurrence of `f` in `n*f(n-1)` is interpreted as the previous version of `f`.

The keyword `rec` after `val` can be used to force a recursive interpretation:

41

```
fun f n : int = n + 1;
> val f = fn : int -> int

val rec f = fn n => if n=0 then 1 else n*f(n-1);
> val f = fn : int -> int

f 3;
> val it = 6 : int
```

With `val rec` the occurrence of `f` in `n*f(n-1)` is interpreted recursively.

4.15 Equality types

Simple 'concrete' values like integers, booleans and strings are easy to test for equality. Values of simple datatypes, like pairs and records, whose components have concrete types are also easy to test for equality. For example, (v_1,v_2) is equal to (v_1',v_2') if and only if $v_1 = v_1'$ and $v_2 = v_2'$. There is thus a large class of types whoses values can be tested for equality. However, in general it is undecidable to test the equality of functions. It is thus not possible to overload `=` to work properly on all types. In old versions of ML, `=` was interpreted on functions by testing the equality of the addresses in memory of the data-structure representing the functions. If such a test yielded `true` then the functions were certainly equal, but many mathematically (i.e. extensionally) equal functions were different using this interpretation of `=`.

In Standard ML, those types whose values can be tested for equality are called "equality types" and are treated specially. Special type variables that are constrained only to range over equality types are provided. These have the form `''`α, whereas ordinary type variables have the form `'`α. The built-in function `=` has type `''a * ''a -> bool`. Starting from this, the ML typechecker can infer types containing equality type variables.

42

```
fun Eq x y = (x = y);
> val Eq = fn : ''a -> ''a -> bool

fun EqualHd l1 l2 = (hd l1 = hd l2);
> val EqualHd = fn : ''a list -> ''a list -> bool
```

Trying to instantiate an equality type variable to a functional type results in an error. In SML/NJ:

43
```
hd = hd;
> Error: operator and operand don't agree (equality type required)
>   operator domain: ''Z * ''Z
>   operand:         ('Y list -> 'Y) * ('X list -> 'X)
>   in expression:
>     = (hd,hd)

EqualHd [hd] [hd];
> Error: operator and operand don't agree (tycon mismatch)
>   operator domain: 'Z * 'Z
>   operand:         ''Y list -> ''Y list -> bool
>   in expression:
>     - : overloaded EqualHd
```

The use of equality types in Standard ML is considered controversial: some people think they are too messy for the benefit they provide. It is possible that future versions of ML will drop equality types.

4.16 Pattern matching

Functions can be defined by pattern matching. For example here is another definition of the factorial function.

44
```
fun fact 0 = 1
 |  fact n = n * (fact(n-1));
> val fact = fn : int -> int
```

Here is the Fibonacci function:

45
```
fun fib 0 = 0
 |  fib 1 = 1
 |  fib n = fib(n-1) + fib(n-2);
> val fib = fn : int -> int
```

Suppose function f is defined by:

46
```
fun f p1  =  e1
 |  f p2  =  e2
    ⋮
 |  f pn  =  en
```

An expression $f\ e$ is evaluated by successively matching the value of e with the patterns $p_1, p_2, \ldots, p_n$ (in that order) until a match is found, say with p_i. Then the value of $f\ e$ is the value of e_i. During the match variables in the patterns may be bound to components of e's value and then the variables have these values during the evaluation of e_i. For example, evaluation `fib 8` causes `8` to be matched with `0` then `1`, both of which fail, and then with n which succeeds, binding n to `8`. The result is then the value of `fib(8-1) + fib(8-2)` which (after some recursive calls), evaluates to `21`.

47
```
fib 8;
> val it = 21 : int
```

Patterns can be quite elaborate and are typically composed with 'constructors' (see Section 4.19 below).

The patterns in a function definition need not be exhaustive. In SML/NJ;

```
- fun foo 0 = 0;                                                48
std_in:33.1-33.13 Warning: match nonexhaustive
          0 => ...

val foo = fn : int -> int
```

In Edinburgh ML:

```
- fun foo 0 = 0;                                                49
***Warning:  Patterns in Match not exhaustive:  0=>0
> val foo = Fn : int -> int
```

If a function is defined with a non-exhaustive match, and then applied to an argument whose value doesn't match any pattern a special kind of run-time error called an exception results (see Section 4.18).
In SML/NJ:

```
- foo 0;                                                        50
val it = 0 : int
- foo 1;

uncaught Match exception std_in:33.1-33.13
```

In Edinburgh ML:

```
- foo 1;                                                        51
Exception raised at top level
Warning: optimisations enabled -
         some functions may be missing from the trace
Exception: Match  raised
```

Messages warning that a match is non-exhaustive will sometimes be omitted from the output shown here.
The built-in list-processing functions `hd` and `tl` can be defined by:

```
fun hd(x::l) = x;                                               52
> Warning: match nonexhaustive
> val hd = fn : 'a list -> 'a

fun tl(x::l) = l;
> Warning: match nonexhaustive
> val tl = fn : 'a list -> 'a list
```

These definitions give exactly the same results as the built-in functions except on the empty list `[]`, where they differ in the exceptions raised – exceptions are described in Section 4.18.
if x is a variable and p a pattern, then the pattern x `as` p is a pattern that matches the same things as p, but has the additional effect that when a match succeeds the value matched is bound to x. Consider the function `RemoveDuplicates`:
The wildcard "_" matches anything:

```
fun null [] = true                                              53
 |  null _  = false;
> val null = fn : 'a list -> bool
```

```
fun RemoveDuplicates[]          = []                                      54
 |  RemoveDuplicates[x]         = [x]
 |  RemoveDuplicates(x1::x2::l) =
     if x1=x2 then RemoveDuplicates(x2::l)
              else x1::RemoveDuplicates(x2::l);

>val RemoveDuplicates = fn : ''a list -> ''a list

RemoveDuplicates[1,1,1,2,3,4,5,5,5,5,5,6,7,8,8,8];
> val it = [1,2,3,4,5,6,7,8] : int list
```

The repetition (and extra list conses) of `x2::l` can be avoided as follows:

```
fun RemoveDuplicates[]                = []                                55
 |  RemoveDuplicates(l as [x])        = l
 |  RemoveDuplicates(x1::(l as x2::_)) =
     if x1=x2 then RemoveDuplicates l else x1::RemoveDuplicates l;
```

Incidently, note that (alas) duplicate variables are not allowed in patterns:

```
fun RemoveDuplicates[]              = []                                  56
 |  RemoveDuplicates(l as [x])      = l
 |  RemoveDuplicates(x::(l as x::_)) = RemoveDuplicates l
 |  RemoveDuplicates(x::l)          = x::RemoveDuplicates l;
> Error: duplicate variable in pattern(s): x
```

Anonymous functions (`fn`-expressions) can be defined by pattern matching using the syntax: `fn` p_1 `=>` e_1 `|` ... `|` p_n `=>` e_n

```
fn [] => "none" | [_] => "one" | [_,_] => "two" | _ => "many";            57
> val it = fn : 'a list -> string

(it [], it[true], it[1,2], it[1,2,3]);
> val it = ("none","one","two","many") : string * string * string * string
```

Patterns can be constructed out of records, with "..." as a wildcard.

```
fun IsMale({sex="male",...}:persondata) = true                           58
 |  IsMale _ = false;
> val IsMale = fn : persondata -> bool

IsMale MikeData;
> val it = true : bool
```

An alternative definition is:

```
fun IsMale({sex=x,...}:persondata) = (x = "male");                       59
```

A more compact form of this is allowed.

```
fun IsMale({sex,...}:persondata) = (sex = "male");                       60
```

The field name `sex` doubles as a variable. Think of a pattern `{···,v,···}` as abbreviating `{···,v=v,···}`.

4.17 The `case` construct

The case construct permits one to compute by cases on an expression of a datatype. The expression `case` e `of` p_1 `=>` e_1 `|` ... `|` p_n `=>` e_n, is an equivalent form for the application `(fn` p_1 `=>` e_1 `|` ... `|` p_n `=>` e_n`)` e.

4.18 Exceptions

Some standard functions *raise exceptions* at run-time on certain arguments. When this happens a special kind of value (called an exception packet) is propagated which identifies the cause of the exception. These packets have names which usually reflect the function that raised the exception; they may also contain values.

```
hd(tl[2]);                                                   61
> uncaught exception Hd

1 div 0;
> uncaught exception Div

(1 div 0)+1000;
> uncaught exception Div
```

Exceptions must be declared using the keyword `exception`; they have type `exn`. Exceptions can be explicitly raised by evaluating an expression of the form `raise` *e* where *e* evaluates to an exception value. Exceptions are printed slightly differently in SML/NJ and Edinburgh ML. In SML/NJ:

```
- exception Ex1;exception Ex2;                               62
exception Ex1
exception Ex2

- [Ex1,Ex2];
val it = [Ex1(-),Ex2(-)] : exn list

- raise hd it;
uncaught exception Ex1
```

In Edinburgh ML:

```
exception Ex1;exception Ex2;                                 63
> type exn
  con Ex1 = - : exn
> type exn
  con Ex2 = - : exn

- [Ex1,Ex2];
> [-,-] : exn list

- raise hd it;
Exception raised at top level
Warning: optimisations enabled -
         some functions may be missing from the trace
Exception: Ex1  raised
```

An exception packet constructor called *name* and which constructs packets containing values of type *ty* is declared by `exception` *name* `of` *ty*.

```
exception Ex3 of string;                                     64
> exception Ex3

Ex3;
> val it = fn : string -> exn

raise Ex3 "foo";
> uncaught exception Ex3
```

The type `exn` is a datatype (see Section 4.19 below) whose constructors are the exceptions. It is the only datatype that can be dynamically extended. All other datatypes have to have all their constructors declared at the time when the datatype is declared.

Because `exn` is a datatype, exceptions can be used in patterns like other constructors. This is useful for handling exceptions.

An exception can be trapped (and its contents extracted) using an exception handler. An important special case is unconditional trapping of all exceptions. The value of the expression e_1 `handle _ =>` e_2 is that of e_1, unless e_1 raises an exception, in which case it is the value of e_2.

```
hd[1,2,3] handle _ => 0;                                                  65
> val it = 1 : int

hd[] handle _ => 0;
> val it = 0 : int

hd(tl[2]) handle _ => 0;
> val it = 0 : int

1 div 0 handle _ => 1000;
> val it = 1000 : int
```

The function `half`, defined below, only succeeds (i.e. doesn't raise an exception) on non-zero even numbers; on `0` it raises `Zero`, and on odd numbers it raises `Odd`.

```
exception Zero; exception Odd;                                            66
> exception Zero
> exception Odd

fun half n =
 if n=0 then raise Zero
        else let
              val m = n div 2
             in
              if n=2*m then m else raise Odd
             end;
> val half = fn : int -> int
```

Some examples of using `half`:

```
half 4;                                                                   67
> val it = 2 : int

half 0;
> uncaught exception Zero

half 3;
> uncaught exception Odd

half 3 handle _ => 1000;
> val it = 1000 : int
```

Failures may be trapped selectively by matching the exception packet; this is done by replacing the wildcard `_` by a pattern. For example, if e raises Ex, then the value of e `handle` Ex_1 `=>` e_1 `|` $\ldots$ `|` Ex_n `=>` e_n is the value of e_i if Ex equals Ex_i otherwise the `handle`-expression raises Ex.

```
half(0) handle Zero => 1000;                                          68
> val it = 1000 : int

half(1) handle Zero => 1000;
> uncaught exception Odd

half(0) handle Zero => 1000 | Odd => 1001;
> val it = 1000 : int

half(3) handle Zero => 1000 | Odd => 1001;
> val it = 1001 : int
```

Instead of having the two exceptions `Zero` and `Odd`, one could have a single kind of exception containing a string.

```
exception Half of string;                                             69
> exception Half

fun half n =
 if n=0 then raise Half "Zero"
        else let
              val m = n div 2
             in
              if n=2*m then m else raise Half "Odd"
             end;
> val half = fn : int -> int
```

A disadvantage of this approach is that the kind of exception is not printed when the exceptions are uncaught.

```
half 0;                                                               70
> uncaught exception Half

half 3;
> uncaught exception Half

half(0) handle Half "Zero" => 1000 | Half "Odd" => 1001;
> val it = 1000 : int

half(3) handle Half "Zero" => 1000 | Half "Odd" => 1001;
> val it = 1001 : int
```

Alternatively, one can match the contents of the exception packet to a variable, `s` say, and then branch on the value matched to `s`.

```
half(0) handle Half s => (if s="Zero" then 1000 else 1001);           71
> val it = 1000 : int

half(3) handle Half s => (if s="Zero" then 1000 else 1001);
> val it = 1001 : int
```

4.19 Datatype declarations

New types (rather than mere abbreviations) can also be defined. Datatypes are types defined by a set of constructors which can be used to create objects of that type and also (in patterns) to decompose objects of that type. For example, to define a type `card` one could use the construct `datatype`:

```
datatype card = king | queen | jack | other of int;                72
> datatype  card
> con jack : card
> con king : card
> con other : int -> card
> con queen : card
```

Such a declaration declares `king`, `queen`, `jack` and `other` as constructors and gives them values. The value of a 0-ary constructor such as `king` is the constant value `king`. The value of a constructor such as `other` is a constructor function that given an integer value n produces `other(`n`)`.

```
king;                                                               73
> val it = king : card

other(4+5);
> val it = other 9 : card
```

To define functions that take their argument from a concrete type, `fn`-expressions of the form `fn` p_1 `=>` e_1 `|...|` p_n `=>` e_n can be used. Such an expression denotes a function that given a value v selects the first pattern that matches v, say p_i, binds the variables of p_i to the corresponding components of the value and then evaluates the expression e_i. For example, the values of the different cards can be defined in the following way:

```
val value = fn  king      => 500                                    74
            |   queen     => 200
            |   jack      => 100
            |   (other n) => 5*n;
> val value = fn : card -> int
```

Alternatively, and perhaps more lucidly, this could be defined using a `fun` declaration.

```
fun value king      = 500                                           75
 |  value queen     = 200
 |  value jack      = 100
 |  value (other n) = 5*n;
> val value = fn : card -> int
```

The notion of datatype is very basic and could enable us to build ML's elementary types from scratch. For example, the booleans could be defined simply by:

```
datatype bool = true | false;                                       76
> datatype  bool
> con false : bool
> con true : bool
```

and the positive integers by:

```
datatype int = zero | suc of int;                                   77
> datatype  int
> con suc : int -> int
> con zero : int
```

In a similar way, LISP S-expressions could be defined by:

```
datatype sexp = litatom of string                                    78
              | numatom of int
              | cons    of sexp * sexp;
> datatype  sexp
> con cons : sexp * sexp -> sexp
> con litatom : string -> sexp
> con numatom : int -> sexp

fun car (cons(x,y)) = x and cdr (cons(x,y)) = y;
> Warning: match nonexhaustive
> val car = fn : sexp -> sexp
> Warning: match nonexhaustive
> val cdr = fn : sexp -> sexp

val a1 = litatom "Foo" and a2 = numatom 1;
> val a1 = litatom "Foo" : sexp
> val a2 = numatom 1 : sexp

car(cons(a1,a2));
> val it = litatom "Foo" : sexp

cdr(cons(a1,a2));
> val it = numatom 1 : sexp
```

Notice the warning from the compiler that the patterns in the definitions of `car` and `cdr` are not exhaustive; these funtions are only partially specified — namely only on lists built with `cons` (i.e. non-atoms).

```
car (litatom "foo");                                                 79
> uncaught exception Match
```

4.20 Abstract types

New types can also be defined by abstraction. For example, a type `time` could be defined as follows:

```
exception BadTime;                                                   80
> exception BadTime

abstype time = time of int * int
 with
  fun maketime(hrs,mins)   = if  hrs<0  orelse 23<hrs orelse
                                 mins<0 orelse 59<mins
                              then raise BadTime
                              else time(hrs,mins)
  and hours(time(t1,t2))   = t1
  and minutes(time(t1,t2)) = t2
 end;
> type time
> val maketime = fn : int * int -> time
> val hours = fn : time -> int
> val minutes = fn : time -> int
```

This defines an abstract type `time` and three primitive functions: `maketime`, `hours` and `minutes`.

In general, an abstract type declaration has the form `abstype` *d* `with` *b* `end` where *d* is a datatype specification and *b* is a binding, i.e. the kind of phrase that can follow `val`. Such a declaration introduces a new type, *ty* say, as specified by the datatype declaration *d*. However, the constructors declared on *ty* by *d* are only available within *b*. The only bindings that result from executing the `abstype` declaration are those specified in *b*.

Thus an abstract type declaration simultaneously declares a new type together with primitive functions for the type; the representation datatype is not accessible outside the `with`-part of the declaration.

```
val t = maketime(8,30);                                                  81
> val t = - : time

(hours t , minutes t);
> val it = (8,30) : int * int
```

Notice that values of an abstract type are printed as `-`, since their representation is hidden from the user.

4.21 Type constructors

Both `list` and `*` are examples of type constructors; `list` has one argument (hence `'a list`) whereas `*` has two (hence `'a * 'b`). Type constructors may have various predefined operations associated with them, for example `list` has `null`, `hd`, `tl`, ... etc. Because of pattern matching, it is not necessary to have any predefined operations for `*`. One can define, for example, `fst` and `snd` by.

```
fun fst(x,y) = x and snd(x,y) = y;                                       82
> val fst = fn : 'a * 'b -> 'a
> val snd = fn : 'a * 'b -> 'b

val p = (8,30);
> val p = (8,30) : int * int

fst p;
> val it = 8 : int

snd p;
> val it = 30 : int
```

A type constructor `set`, that represents sets by lists without repetitions, can be defined in the following way:

```
abstype 'a set = set of 'a list                                          83
 with
  val emptyset = set[]
  fun isempty(set s) = null s
  fun member(_, set[])     = false
   |  member(x, set(y::z)) = (x=y) orelse member(x, set z)
  fun add(x, set[])     = set[x]
   |  add(x, set(y::z)) = if x=y
                            then set(y::z)
                            else let val set l = add(x, set z) in
                                  set(y::l)
                                 end
 end
> val emptyset = [] : 'a list
> val isempty = fn : 'a set -> bool
> val member = fn : ''a * ''a set -> bool
> val add = fn : ''a * ''a set -> ''a set
```

Note that the operation `add` ensures that no repetitions of elements occur in the list representing the set. Here is an example using these sets:

```
val s = add(1,(add(2,(add(3,emptyset)))));                                84
> val s = - : int set

member(3,s);
> val it = true : bool

member(5,s);
> val it = false : bool
```

4.22 References and assignment

References are 'boxes' that can contain values. The contents of such boxes can be changed using the assignment operator `:=`. The type *ty* `ref` is possessed by references containing values of type *ty*.

References are created using the `ref` operator. This takes a value of type *ty* to a value of type *ty* `ref`. [4] The expression x`:=`e changes the contents of the reference that is the value of x to be the value of e. The value of this assignment expression is the dummy value `()`; this is the unique value of the one-element type `unit`. Assignments are executed for a 'side effect', not for their value.

The contents of a reference can be extracted using the `!` operator (error message below from SML/NJ).

```
x:=1;                                                                     85
> std_in:7.1-7.4 Error: operator and operand don't agree (tycon mismatch)
>   operator domain: 'Z ref * 'Z
>   operand:         int * int
>   in expression:
>     := (x,1)

val x = ref 1 and y = ref 2;
> val x = ref 1 : int ref
> val y = ref 2 : int ref

x;
> val it = ref 1 : int ref

x:=6;
> val it = () : unit

x;
> val it = ref 6 : int ref

!x;
> val it = 6 : int
```

References should only be resorted to in exceptional circumstances as experience shows that their use increases the probability of errors.

4.23 Iteration

Here is an iterative definition of `fact` using two local references: `count` and `result`.

[4] There are some horrible subtleties associated with the types of references, which are ignored here. The treatment of references in ML is currently in a state of flux.

```
 fun fact n =                                                    86
  let val count = ref n and result = ref 1
  in while !count > 0
     do (result := !count * !result;
         count := !count-1);
     !result
  end;
> val fact = fn : int -> int

fact 6;
> val it = 720 : int
```

The semicolon denotes sequencing. When an expression $e_1;\ldots;e_n$ is evaluated, each e_i is evaluated in turn and the value of the entire expression is the value of e_n.

Evaluating `while` e `do` c consists in evaluating e and if the result is true c is evaluated for its side-effect and then the whole process repeats. If e evaluates to false, then the evaluation of `while` e `do` c terminates with value `()`.

4.24 Programming in the large

Sophisticated features for structuring collections of declarations ('programming in the large') are provided in Standard ML (but not in earlier versions of ML). These are designed to support the use of ML for large scale system building. They account for much of the complexity of the language.

Standard ML of New Jersey is implemented in itself and makes extensive use of these features. Edinburgh ML does not implement them.

The concepts of structures, signatures and functors, which provide the structuring constructs for programming in the large, are not covered in this course (hence their absence from Edinburgh ML on Thor will not be a problem).

Chapter 5

Case study 1: parsing

The lexical analysis and parsing programs described here are intended to illustrate functional programming methods and ML, rather than parsing theory. The style of parsing presented is quite reasonable for small lightweight ad hoc parsers, but would be inappropriate for large applications, which should be handled using heavyweight parser generators like YACC.[1]

5.1 Lexical analysis

Lexical analysis converts sequences of characters into sequences of tokens (also called "words" or "lexemes").

For us, a token will be either a number (sequence of digits), an identifier (a sequence of letters or digits starting with a letter) or a 'special symbol' such as `+`, `*`, `<`, `==>` or `++`. Special symbols are specified by a table (see below).

A number is a sequence of digits, The ML infix operator `<=` is overloaded and can be applied to strings. If x and y are single-character strings, then x`<=`y just tests whether the ASCII code of x is less then or equal to that of y. Thus a single-character string representing a digit can be characterised by the predicate `IsDigit`:

```
fun IsDigit x = "0" <= x andalso x <= "9";                            87
> val IsDigit = fn : string -> bool
```

A letter can similarly be characterised by making use of the fact that the ASCII codes of all lower case letters are adjacent and also the codes of all upper case latters are adjacent.

```
fun IsLetter x =                                                      88
 ("a" <= x andalso x <= "z") orelse ("A" <= x andalso x <= "Z");
> val IsLetter = fn : string -> bool
```

Token are separated by 'separators', which will be taken to be spaces, newlines and tabs, hence:

```
fun IsSeparator x = (x = " " orelse x = "\n" orelse x = "\t");        89
> val IsSeparator = fn : string -> bool
```

Single characters that are not digits, letters or separators will be assumed to be special symbols. Multi-character special symbols (e.g. `==>`) are considered later.

The input is assumed to be supplied as a list of single-charater strings. Lexical analysis consists on converting such a list to a list of tokens.

Suppose the input just consists of numbers separated by separators. A function `Tokenise` that did lexical analysis for just this case would need to repeatedly remove

[1]There is an ML based version of YACC. The parser for SML/NJ uses this.

digits until a non-digit (e.g. a separator) was reached, and then implode the removed characters into a string representing a token and add that to the list of tokens.

The function `GetNumber` takes a list, l say, of single-character strings and returns a pair consisting of (i) a string representing a number consisting of all the digits in l up to the first non-digit and (ii) the remainder of l after these digits have been removed. It is convenient to define `GetNum` using an auxiliary function `GetNumAux` that has an extra argument `buf` for accumulating a (reversed) list of characters making up the number.

90

```
fun GetNumAux buf [] = (implode(rev buf), [])
 |  GetNumAux buf (l as (x::l')) =
     if IsDigit x then GetNumAux (x::buf) l'
                  else (implode(rev buf),l);
> val GetNumAux = fn : string list -> string list -> string * string list

GetNumAux ["a","b","c"] ["1","2","3"," ","4","5"];
> val it = ("cba123",[" ","4","5"]) : string * string list
```

Then `GetNum` is simply defined by:

91

```
val GetNum = GetNumAux [];
> val GetNum = fn : string list -> string * string list

GetNum ["1","2","3"," ","4","5"];
> val it = ("123",[" ","4","5"]) : string * string list
```

The definition of `GetNumAux` could have been localised to `GetNum` using `local···in···end`.

Notice that if the list argument of `GetNum` doesn't start with a number, then the empty token (`implode[]`) will be returned.

92

```
GetNum ["a","0","1"];
> val it = ("",["a","0","1"]) : string * string list
```

This problem will go away when we improve the code later on.

Identifiers can be lexically analysed by similar programs:

93

```
fun GetIdentAux buf [] = (implode(rev buf), [])
 |  GetIdentAux buf (l as (x::l')) =
     if IsLetter x orelse IsDigit x
      then GetIdentAux (x::buf) l'
      else (implode(rev buf),l);
> val GetIdentAux = fn : string list -> string list -> string * string list

GetIdentAux ["a","b","c"] ["e","f","g","4","5"," ","6","7"];
> val it = ("cbaefg45",[" ","6","7"]) : string * string list
```

An identifier must start with a letter, so `GetIdent` is defined by:

94

```
exception GetIdentErr;
> exception GetIdentErr

fun GetIdent (x::l) =
 if IsLetter x then GetIdentAux [x] l else raise GetIdentErr;
> val GetIdent = fn : string list -> string * string list
```

The lexical analysis of numbers and identifiers can be streamlined and unified by defining a single general function `GetTail` that takes a predicate as an argument and then uses this to test whether to keep accumulating characters in `buf` or to terminate. Then `GetNumAux` corresponds to `GetTail IsDigit` and `GetIdentAux` to `GetTail (fn x => IsLetter x orelse IsDigit x)`.

The definition of `GetTail` is similar to that of `GetNumAux` and `GetIdentAux`.

```
fun GetTail p buf [] = (implode(rev buf),[])                                95
 |  GetTail p buf (l as x::l') =
     if p x then GetTail p (x::buf) l' else (implode(rev buf),l);
> val GetTail = fn
>   : (string->bool) -> string list -> string list -> string * string list
```

Using `GetTail`, a function to get the next token is easy to define:

```
fun GetNextToken [x] = (x,[])                                               96
 |  GetNextToken (x::l) =
     if IsLetter x
      then GetTail (fn x => IsLetter x orelse IsDigit x) [x] l
      else if IsDigit x
            then GetTail IsDigit [x] l
            else (x,l);
> val GetNextToken = fn : string list -> string * string list
```

To lexically analyse a list of characters, `GetNextToken` is repeatedly called and separators are discarded.

```
fun Tokenise [] = []                                                        97
 |  Tokenise (l as x::l')  =
     if IsSeparator x
      then Tokenise l'
      else let val (t,l'') = GetNextToken l
           in t::(Tokenise l'') end;
> val Tokenise = fn : string list -> string list

Tokenise (explode "123abcde1][  ] 56a");
> val it = ["123","abcde1","]","[","]","56","a"] : string list
```

`Tokenise` does not handle multi-character special symbols. These will be specified by a table, represented as a list of pairs, that shows which characters can follow each initial segment of each special symbol (such a table represents a state-transition function for an automaton). For example, suppose the special symbols are `<=`, `<<`, `=>`, `=`, `==>` and `->`, then the table would be:

```
[("<",  ["=","<"]),                                                         98
 ("=",  [">","="]),
 ("-",  [">"]),
 ("==", [">"])]
```

This is not fully general because if `==>` is a special symbol, then the representation above forces `==` to be also. A fully general treatment of special symbols is left as an exercise.

Some utility functions are needed. `Mem` tests whether an element occurs in a list.

```
fun Mem x [] = false                                                        99
 |  Mem x (x'::l) = (x=x') orelse Mem x l;
> val Mem = fn : ''a -> ''a list -> bool

Mem 3 [1,2,3,4,5,6,7];
> val it = true : bool

Mem 9 [1,2,3,4,5,6,7];
> val it = false : bool
```

Notice the equality types.

`Get` looks up the list of possible successors of a given string in a special-symbol table.

100
```
fun Get x [] = []
 |  Get x ((x',l)::rest) = if x=x' then l else Get x rest;
> val Get = fn : ''a -> (''a * 'b list) list -> 'b list

Get "=" [("<",["=","<"]), ("=",[">","="]), ("-",[">"]), ("==",[">"])];
> val it = [">","="] : string list

Get "?" [("<",["=","<"]), ("=",[">","="]), ("-",[">"]), ("==",[">"])];
> val it = [] : string list
```

The function `GetSymbol` takes a special-symbol table and a token and then extends the token by removing characters from the input until the table says that no further extension is possible.

101
```
fun GetSymbol spectab tok [] = (tok,[])
 |  GetSymbol spectab tok (l as x::l') =
     if Mem x (Get tok spectab) then GetSymbol spectab (tok^x) l'
                                 else (tok,l);
> val GetSymbol = fn
>   : (string * string list) list
>     -> string -> string list -> string * string list
```

The function `GetNextToken` can be enhanced to handle special symbols. It needs to take a special-symbol table as an argument.

102
```
fun GetNextToken spectab [x] = (x,[])
 |  GetNextToken spectab (x::(l as x'::l')) =
     if IsLetter x
      then GetTail (fn x => IsLetter x orelse IsDigit x) [x] l
      else if IsDigit x
            then GetTail IsDigit [x] l
            else if Mem x' (Get x spectab)
                  then GetSymbol spectab (implode[x,x']) l'
                  else (x,l);
> val GetNextToken = fn
>   : (string * string list) list -> string list -> string * string list
```

Now `Tokenise` can be enhanced to use the new `GetNextToken`.

103
```
fun Tokenise spectab [] = []
 |  Tokenise spectab (l as x::l')  =
     if IsSeparator x
      then Tokenise spectab l'
      else let val (t,l'') = GetNextToken spectab l
           in t::(Tokenise spectab l'') end;
> val GetNextToken = fn
>   : (string * string list) list -> string list -> string * string list
```

Here is a particular table:

104
```
val SpecTab = [("=",  ["<",">","="]),
               ("<",  ["<",">"]),
               (">",  ["<",">"]),
               ("==", [">"])];
> val SpecTab =
>   [("=",["<",">","="]),("<",["<",">"]),(">",["<",">"]),("==",[">"])]
>   : (string * string list) list

Tokenise SpecTab (explode "a==>b c5 d5==ff+gg7");
> val it = ["a","==>","b","c5","d5","==","ff","+","gg7"] : string list
```

In the next section the lexical analyset `Lex` will be used.

```
val Lex = Tokenise SpecTab o explode;                                   105
> val Lex = fn : string -> string list

Lex "a==>b c5 d5==ff+gg7";
> val it = ["a","==>","b","c5","d5","==","ff","+","gg7"] : string list
```

5.2 Simple special cases of parsing

Before giving a complete parser, some special cases are considered.

5.2.1 Applicative expressions

Examples of applicative expressions are `x`, `f x`, `(f x) y`, `f(f x)`, `f(g x)(h x)` etc. Parse trees for such expression can be represented by the recursive datatype `tree`.

```
datatype tree = Atom of string | Comb of tree * tree;                   106
> datatype  tree
>   con Atom : string -> tree
>   con Comb : tree * tree -> tree
```

Right associative without brackets

Suppose for the moment: (i) the input is supplied as a list of atoms, (ii) brackets are ignored and (ii) application is taken to be right associative. Then a simple parser is:

```
fun Parse [next]       = Atom next                                      107
 |  Parse (next::rest) = Comb(Atom next, Parse rest);
> Warning: match nonexhaustive
> val Parse = fn : string list -> tree

Parse["f", "x", "y", "z"];
> val it = Comb (Atom "f",Comb (Atom "x",Comb (Atom "y",Atom "z"))) : tree
```

Left associative without brackets

The usual convention is for application to be left associative. A parser for this is only slightly more complex.
To parse `f x y z` the following intermediate parsings need to be done:

1. `f` is parsed to `Atom "f"`
2. `f x` is parsed to `Comb(Atom "f", Atom "x")`
3. `f x y` is parsed to `Comb(Comb(Atom "f", Atom "x"), Atom "y")`

Intermediate parse trees will be 'passed forward' via a variable `t` of an auxiliary function `Parser`.

108

```
fun Parser t []           = t
 |  Parser t (next::rest) = Parser (Comb(t, Atom next)) rest;
> val Parser = fn : tree -> string list -> tree

fun Parse [next]       = Atom next
 |  Parse (next::rest) = Parser (Atom next) rest;
> Warning: match nonexhaustive
> val Parse = fn : string list -> tree

Parse["f", "x", "y", "z"];
> val it = Comb (Comb (Comb (Atom "f",Atom "x"),Atom "y"),Atom "z") : tree
```

Right associative with brackets

Brackets will now be considered. To parse $\cdots(e)\cdots$, the parser must be called recursively inside the brackets to parse e, and then the presence of the closing bracket must be checked. Such a recursive call needs to return the parse tree for e and the rest of the input list. Thus the type of `Parse` changes to `string list -> tree * string list`.

If the parser encounters an unexpected closing bracket then it returns the parse tree so far and the rest of the input. For example, `["x","y",")","z"]` should parse to `(Comb(Atom "x", Atom "y"), [")","z"])`.

However, there may be no "parse tree so far" and to handle this case it is convenient to add an empty tree `Nil` to the type `tree`.

109

```
datatype tree = Nil | Atom of string | Comb of tree * tree;
> datatype  tree
>   con Atom : string -> tree
>   con Comb : tree * tree -> tree
>   con Nil : tree
```

Right associative function application is considered first. A first attempt is:

110

```
exception MissingClosingBracket;
> exception MissingClosingBracket

fun Parse []                = (Nil,[])
 |  Parse (rest as ")"::_) = (Nil,rest)
 |  Parse ("("::rest)       =
     (case Parse rest
      of   (t, ")"::rest') => let val (t',rest'') = Parse rest'
                              in (Comb(t,t'), rest'') end
       |    _               => raise MissingClosingBracket)
 |  Parse (next::rest) = let val (t,rest') = Parse rest
                         in (Comb(Atom next,t),rest') end;
> val Parse = fn : string list -> tree * string list
```

This doesn't quite work:

111

```
Parse ["x"];
> val it = (Comb (Atom "x",Nil),[]) : tree * string list

Parse ["x","y","z"];
> val it = (Comb (Atom "x",Comb (Atom "y",Comb (Atom "z",Nil))),[])

Parse ["x","y",")","z"];
> val it =
   (Comb (Atom "x",Comb (Atom "y",Nil)),[")","z"]) : tree * string list
```

The empty parse tree `Nil` returned when `Parse` exits needs to be removed. This is easily done by replacing `Comb` by `MkComb`, where:

112
```
fun MkComb(t,Nil) = t
 |  MkComb p      = Comb p;
```

Then `Parse` is redefined:

113
```
fun Parse []               = (Nil,[])
 |  Parse (rest as ")"::_) = (Nil,rest)
 |  Parse ("("::rest)      =
     (case Parse rest
      of   (t, ")"::rest') => let val (t',rest'') = Parse rest'
                               in (MkComb(t,t'), rest'') end
       |    _              => raise MissingClosingBracket)
 |  Parse (next::rest) = let val (t,rest') = Parse rest
                         in (MkComb(Atom next,t),rest') end;
> val Parse = fn : string list -> tree * string list
```

Parse now works on well-formed expressions.

114
```
Parse ["x"];
> val it = (Atom "x",[]) : tree * string list

Parse ["x","y","z"];
> val it =
>  (Comb (Atom "x",Comb (Atom "y",Atom "z")),[]) : tree * string list

Parse ["x","y",")","z"];
val it = (Comb (Atom "x",Atom "y"),[")","z"]) : tree * string list
```

However, the empty parse tree `Nil` can still be generated, but only in pathological situations.

115
```
Parse ["(",")"];
> val it = (Nil,[]) : tree * string list

Parse ["(",")","a"];
> val it = (Comb (Nil,Atom "a"),[]) : tree * string list

Parse ["(",")","(",")"];
> val it = (Nil,[]) : tree * string list

Parse [")","x"];
> val it = (Nil,[")","x"]) : tree * string list
```

This might be acceptable, but probably it is better to distinguish the first three examples from the last. In the modified version of `Parse` that follows, `()` parses to the 'empty atom' `Atom ""`, where `""` is the empty string.

Here is the revised definition of `Parse`:

116
```
fun Parse []               = (Nil,[])
 |  Parse ("("::")"::rest) = let val (t,rest') = Parse rest
                              in (MkComb(Atom "",t),rest') end
 |  Parse (rest as ")"::_) = (Nil,rest)
 |  Parse ("("::rest)      =
     (case Parse rest
      of   (t, ")"::rest') => let val (t',rest'') = Parse rest'
                               in (MkComb(t,t'), rest'') end
       |    _              => raise MissingClosingBracket)
 |  Parse (next::rest)     = let val (t,rest') = Parse rest
                              in (MkComb(Atom next,t),rest') end;
> val Parse = fn : string list -> tree * string list
```

The pathological examples now become:

```
Parse ["(",")"];                                                    117
> val it = (Atom "",[]) : tree * string list

Parse ["(",")","a"];
> val it = (Comb (Atom "",Atom "a"),[]) : tree * string list

Parse ["(",")","(",")"];
> val it = (Comb (Atom "",Atom ""),[]) : tree * string list
```

The second, fourth and fifth clauses of this latest definition of `Parse` contain some repetition. This can be mitigated by defining an auxiliary function for building a combination. `BuildComb parse t inp` builds a combination whose operator is `t` and whose operand is got by calling the supplied parser function `parse` on the supplied input `inp`. The combination and the remainder of the input are returned.

```
fun BuildComb parse t inp =                                         118
 let val (t', rest) = parse inp
 in (MkComb(t,t'), rest) end;
> val BuildComb = fn : ('a -> tree * 'b) -> tree -> 'a -> tree * 'b

fun Parse []               = (Nil,[])
 |  Parse ("("::")"::rest) = BuildComb Parse (Atom "") rest
 |  Parse (rest as ")"::_) = (Nil,rest)
 |  Parse ("("::rest)      =
      (case Parse rest
       of   (t, ")"::rest') => BuildComb Parse t rest'
       |    _               => raise MissingClosingBracket)
 |  Parse (next::rest)     = BuildComb Parse (Atom next) rest;
```

Notice how a mutual recursion is set up between `Parse` and `BuildComb` via the function argument `parse`. This technique will be exploited again.

The messy fourth clause of `Parse` could be simplified with another auxiliary function that called a supplied parser and then checked for a given input symbol.

```
fun CheckSym parse sym exn inp =                                    119
 case Parse inp
 of   (t, next::rest) => if next=sym then BuildComb parse t rest
                                     else raise exn
 |    _               => raise exn;
> val CheckSym = fn
>   : (string list -> tree * 'a)
>      -> string -> exn -> string list -> tree * 'a

fun Parse []               = (Nil,[])
 |  Parse ("("::")"::rest) = BuildComb Parse (Atom "") rest
 |  Parse (rest as ")"::_) = (Nil,rest)
 |  Parse ("("::rest)      = CheckSym Parse ")" MissingClosingBracket rest
 |  Parse (next::rest)     = BuildComb Parse (Atom next) rest;
```

Whether or not this is an improvement is a matter of taste.

Left associative with brackets

The technique used above – passing intermediate parse trees as arguments to an auxiliary function – will be used to parse left-associating applicative expressions with brackets. Here is a first attempt.

```
fun Parser t [] = (t, [])                                                120
 |  Parser t ("("::")"::rest) = Parser (Comb(Atom "",t)) rest
 |  Parser t (inp as ")"::_) = (t, inp)
 |  Parser t [next] = (Comb(t,Atom next), [])
 |  Parser t ("("::rest) =
     (case  Parser Nil rest
      of    (t', ")"::rest') => Parser (Comb(t,t')) rest'
      |     _                => raise MissingClosingBracket)
 |  Parser t  (next::rest) = Parser (Comb(t,Atom next)) rest;

val Parse = Parser Nil;
```

This has a familiar problem with `Nil`

```
Parse ["x"];                                                             121
> val it = (Comb (Nil,Atom "x"),[]) : tree * string list

Parse ["x","(","y","z",")","w"];
> val it =
>   (Comb
>      (Comb
>         (Comb (Nil,Atom "x"),
>          Comb (Comb (Nil,Atom "y"),Atom "z")),Atom "w"),
>    []) : tree * string list
```

The solution is to use `MkComb` instead of `Comb`, where:

```
fun MkComb(Nil,t2) = t2                                                  122
 |  MkComb p       = Comb p;

fun Parser t [] = (t, [])
 |  Parser t ("("::")"::rest) = Parser (Comb(Atom "",t)) rest
 |  Parser t (inp as ")"::_) = (t, inp)
 |  Parser t [next] = (MkComb(t,Atom next), [])
 |  Parser t ("("::rest) =
     (case  Parser Nil rest
      of    (t', ")"::rest') => Parser (MkComb(t,t')) rest'
      |     _                => raise MissingClosingBracket)
 |  Parser t  (next::rest) = Parser (MkComb(t,Atom next)) rest;

val Parse = Parser Nil;
```

Then:

```
Parse ["x"];                                                             123
> val it = (Atom "x",[]) : tree * string list

Parse ["x","(","y","z",")","w"];
> val it = (Comb (Comb (Atom "x",Comb (Atom "y",Atom "z")),Atom "w"),[])
>   : tree * string list
```

5.2.2 Precedence parsing of infixes

The parsing of expressions like $x + y \times z$ will now be considered. Parse trees are represented by:

```
datatype tree = Nil                                                      124
              | Atom of string
              | BinOp of string * tree * tree;
```

Binary operators are assumed to have a precedence given by a table represented as a list of pairs.

```
val BinopTable =                                                        125
 [("*", 7),
  ("+", 6)];
```

The function Lookup gets the precedence of an operator from such a table.

```
fun Lookup ((s,n)::tab) x = if x=s then n else Lookup tab x;           126
> Warning: match nonexhaustive
> val Lookup = fn : (''a * 'b) list -> ''a -> 'b
```

Note that the use of = forces an equality type. A string is an operator if it is assigned a precedence by the precedence table:

```
fun InTable [] x = false                                                127
 |  InTable ((s,n)::tab) x = (x=s orelse InTable tab x);
> val InTable = fn : (''a * 'b) list -> ''a -> bool
```

The parser function Parser that follows is quite tricky, but uses many of the principles that have already been illustrated. The main new ingredient are precedences. Assume * (i.e. the ASCII version of $\times$) has higher precedence than =. Consider the parsing of x*y+z versus the parsing of x+y*z.

For x*y+z the parser must proceed by:

A1: first building BinOp("*", Atom "x", Atom "y")

A2: then building Atom "z"

A3: finally building Binop("+", BinOp("*", Atom "x", Atom "y"), Atom "z")

For x+y*z the parser must proceed by:

B1: first building Atom "x"

B2: then building BinOp("*", Atom "y", Atom "z")

A3: finally building Binop("+", Atom "x", BinOp("*", Atom "y", Atom "z"))

In both cases, the left hand argument to the operator must be held whilst the right hand argument is parsed. This will be done by giving Parser an extra parameter: the parse-tree of the already-parsed argument (or Nil). This is the same idea already used to parse left associative applicative expressions (in that case, the extra parameter holding the already-parsed rator). The name of the extra parameter will be t.

Precedences are used to decide between A1–A3 and B1–B3. During the parsing there is a *current precedence* of the parse. For example, if the data in BinopTable above is used, then when parsing the second argument of + the precedence will be 6 and when parsing the second argument of * the precedence will be 7.

If the current precedence is m and a binary operator, *op* say, is encountered whose precedence is n, then:

A: if m>n the expression just parsed is the second argument of the operator that proceeds it (case A1–A3)

B: if not m>n then the expression just parsed is the first argument of the already-encountered operator *op* (case B1–B3).

The expression just parsed will be the parse tree bound to the parse tree parameter (`t`) of `Parser`. So in case A above this parameter should be returned immediately. In case B, the parser should be called recursively to get the parse tree, `t'` say, of the second argument of *op* and then `BinOp(`*op*`,t,t')` returned.

With this explanation, I hope the ML code to achieve this is comprehensible.

```
fun Parser tab m t [] = (t, [])                                                128
 |  Parser tab m t (inp as next::rest) =
     if InTable tab next
      then let val n = Lookup tab next
           in if (m:int) > n
               then (t, inp)
               else let val (t',rest') = Parser tab n Nil rest
                    in  Parser tab m (BinOp(next,t, t')) rest' end
           end
      else Parser tab m (Atom next) rest;
> val Parser = fn
>  : (string * int) list
>     -> int -> tree -> string list -> tree * string list

val Parse = Parser BinopTable 0 Nil;
> val Parse = fn : string list -> tree * string list
```

Here are some examples,

```
Parse ["x","*","y","+","z"];                                                   129
> val it = (BinOp ("+",BinOp ("*",Atom "x",Atom "y"),Atom "z"),[])

Parse ["x","+","y","*","z"];
> val it = (BinOp ("+",Atom "x",BinOp ("*",Atom "y",Atom "z")),[])

Parse ["x","+","y","+","z"];
> val it = (BinOp ("+",Atom "x",BinOp ("+",Atom "y",Atom "z")),[])
>   : tree * string list
```

The last of these examples shows that binary operators are parsed as right associative. This is because `>` is used to compare the current precedence with that of an encountered operator. Since m>m is always false, the effect is as though the the operator on the left is not of higher precedence than the one on the right, hence right associativity. If `>` is changed to `>=` then operators will parse as left associative. The general case where some operators are left associative and some right associative can be handled by giving operators both left and right precedences. This is considered in Section 5.3.

A property of the above parser is that if `next` is not a known operator (i.e. is not in `tab`), then the last parse tree parsed (viz `t`) is thrown away.

```
Parse ["x","y","z"];                                                           130
> val it = (Atom "z",[]) : tree * string list
```

Instead of doing this, juxtaposed expressions without any intervening binary operators can be interpreted as function applications. First, parse trees have to be updated to permit this.

```
datatype tree = Nil                                                            131
              | Atom of string
              | Comb of tree * tree
              | BinOp of string * tree * tree;
```

Then the last line of `Parser` is modified:

132

```
fun Parser tab m t [] = (t, [])
 |  Parser tab m t (inp as next::rest) =
     if InTable tab next
      then let val n = Lookup tab next
           in if (m:int) > n
               then (t, inp)
               else let val (t',rest') = Parser tab n Nil rest
                    in  Parser tab m (BinOp(next,t, t')) rest' end
           end
      else Parser tab m (Comb(t, Atom next)) rest;
> val Parser = fn
>  : (string * int) list
>     -> int -> tree -> string list -> tree * string list

val Parse = Parser BinopTable 0 Nil;
> val Parse = fn : string list -> tree * string list
```

This almost works:

133

```
Parse ["x","y","z"];
> val it = (Comb (Comb (Comb (Nil,Atom "x"),Atom "y"),Atom "z"),[])
```

The usual trick of using `MkComb` instead of `Comb` is needed.

134

```
fun MkComb(Nil,t2) = t2
 |  MkComb p       = Comb p;

fun Parser tab m t [] = (t, [])
 |  Parser tab m t (inp as next::rest) =
     if InTable tab next
      then let val n = Lookup tab next
           in if (m:int) > n
               then (t, inp)
               else let val (t',rest') = Parser tab n Nil rest
                    in  Parser tab m (BinOp(next,t, t')) rest' end
           end
      else Parser tab m (MkComb(t, Atom next)) rest;

val Parse = Parser BinopTable 0 Nil;
```

This works.

135

```
Parse ["x","y","z"];
> val it = (Comb (Comb (Atom "x",Atom "y"),Atom "z"),[])

Parse ["x","y","+","z"];
> val it = (BinOp ("+",Comb (Atom "x",Atom "y"),Atom "z"),[])
>   : tree * string list
```

Notice that function application binds tighter than binary operators, which is normally what is wanted (though achieved here in a rather ad hoc and accidental manner). Two things that `Parse` doesn't handle are unary operators and brackets. The datatype of parse trees needs to be adjusted to handle unary operators.

136

```
datatype tree = Nil
              | Atom of string
              | Comb of tree * tree
              | Unop of string * tree
              | BinOp of string * tree * tree;

fun MkComb(Nil,t2) = t2
 |  MkComb p       = Comb p;
```

Unary operators need a precedence. If `~` has higher precedence than `+` then `~x+y` parses as `BinOp("+", Unop("~", Atom "x"), Atom "y")`. If `~` has lower precedence than `+` then `~x+y` parses as `Unop("~", BinOp("+", Atom "x", Atom "y"))`.
The precendences of unary operators will be held in a table.

```
val UnopTable =                                                    137
 [("~", 8),
  ("!", 5)];
```

Parsing is now straightforward: both the unary and binory operator tables need to be passed to `Parser` and an extra clauses is added to test for unary operators.

```
fun Parser (tab as (utab,btab)) m t [] = (t, [])                   138
 |  Parser (tab as (utab,btab)) m t (inp as next::rest) =
     if InTable utab next
      then let val n = Lookup utab next
           in let val (t',rest') = Parser tab n Nil rest
              in Parser tab m (MkComb(t, Unop(next,t'))) rest' end
           end
      else if InTable btab next
            then let val n = Lookup btab next
                 in if (m:int) > n
                     then (t, inp)
                     else let val (t',rest') = Parser tab n Nil rest
                          in  Parser tab m (BinOp(next,t, t')) rest' end
                 end
            else Parser tab m (MkComb(t, Atom next)) rest;

> val Parse = Parser (UnopTable,BinopTable) 0 Nil;
```

Here are some examples:

```
Parse ["x"];                                                       139
> val it = (Atom "x",[]) : tree * string list
Parse ["~","x"];
val it = (Unop ("~",Atom "x"),[]) : tree * string list

Parse ["~","x","+","y"];
> val it = (BinOp ("+",Unop ("~",Atom "x"),Atom "y"),[])

Parse ["!","x","+","y"];
> val it = (Unop ("!",BinOp ("+",Atom "x",Atom "y")),[])
```

Brackets can be handled in the same way as they were for (left associated) applicative expressions.

```
fun Parser (tab as (utab,btab)) m t [] = (t, [])                          140
  |  Parser tab m t ("("::")"::rest) = Parser tab m (Comb(Atom "",t)) rest
  |  Parser tab m t (inp as ")"::_) = (t, inp)
  |  Parser tab m t ("("::rest) =
      (case  Parser tab 0 Nil rest
       of    (t', ")"::rest') => Parser tab m (MkComb(t,t')) rest'
       |     _                => raise MissingClosingBracket)
  |  Parser (tab as (utab,btab)) m t (inp as next::rest) =
      if InTable utab next
       then let val n = Lookup utab next
            in let val (t',rest') = Parser tab n Nil rest
               in Parser tab m (MkComb(t, Unop(next,t'))) rest' end
            end
       else if InTable btab next
             then let val n = Lookup btab next
                  in if (m:int) > n
                      then (t, inp)
                      else let val (t',rest') = Parser tab n Nil rest
                           in  Parser tab m (BinOp(next,t, t')) rest' end
                  end
             else Parser tab m (MkComb(t, Atom next)) rest;

val Parse = Parser (UnopTable,BinopTable) 0 Nil;
```

Here are some more examples.

```
fun P s = Parse(explode s);                                               141
> val P = fn : string -> tree * string list

P "~(x+y)";
> val it = (Unop ("~",BinOp ("+",Atom "x",Atom "y")),[])

P "(~x)+y";
> val it = (BinOp ("+",Unop ("~",Atom "x"),Atom "y"),[])

P "(x+y)(zw)";
> val it =
>   (Comb (BinOp ("+",Atom "x",Atom "y"),Comb (Atom "z",Atom "w")),[])
```

5.3 A general top-down precedence parser

The parser just given works by looking at the next item being input and then invokes some action, which depends on the item, to parse the rest of the input. A more general scheme is to associate actions with items and then to have a simple parsing loop that consists in repeatedly reading an item and then executing the associated action.[2]

A rather general datatype of parse trees is the following.

```
datatype tree = Nil                                                       142
              | Atom of string
              | Comb of tree * tree
              | Node of string * tree list;
```

Unary operator expressions will parse to trees of the form `Node(`$name$`,[`arg`])` and binary operator expressions to trees of the form `Node(`$name$`,[`arg_1`,`arg_2`])`.

The familiar `MkComb` hack will be needed. The empty parse tree `Nil` should never arise as the right component of a combination (since left associative application will be adopted), so an exception will be raised if it does.

[2]The parser described here is loosely based on Vaughan Pratt's CGOL system (MIT, 1974).

```
exception NilRightArg;                                                143

fun MkComb(Nil,t2) = t2
 |  MkComb(t1,Nil) = raise NilRightArg
 |  MkComb(t1, t2) = Comb(t1,t2);
```

The action associated with an item may involve recursive calls to the parser. To handle this the techique described earlier of passing a parse function as an argument can be used (see `BuildComb` and `CheckSym` described above). The type of parse functions is given by the the following type abbreviation.

```
type parser = int -> tree -> string list -> tree * string list;       144
```

Selected input items will have precedences and actions associated with them. Precedences are integers. Intuitively, actions are represented by a functions of type `parser`. However, since an action might need to recursively invoke the whole parser, it should be passed a parsing function. In general, an action must be represented by a function of type `parser->parser`. A symbol table associates precedences and actions to strings.

```
type symtab = string -> int * (parser -> parser);                     145
```

The main parsing function is now very simple, since all the detail has been hived-off into the symbol table.

```
fun Parser symtab (m:int) t [] = (t,[])                               146
 |  Parser symtab m t (inp as next::rest) =
     let val (n,parsefn) = symtab next
     in if m>=n then (t,inp)
               else parsefn (Parser symtab) m t inp
     end;
```

The parse stops on the empty string. If the input isn't empty, then the next item is looked up in the symbol table. Left association will be taken as the default, so if the current precedence equals or exceeds the precedence of the next item, then the parse stops and the last item parsed (`t`) is returned, with the rest of the input. If the current precedence is less than the precedence of the next item, then the parse action associated with the next item in the symbol table is executed. The parse function `Parser symtab` is passed to the parse action, so that it can (if necessary) invoke the whole parse recursively.

The definition of `Parser` intuitively has type `symtab->parser`. However, the actual type assigned by ML is more general:

```
('a
   -> int
      * ((int -> 'b -> 'a list -> 'b * 'a list)
         -> int -> 'b -> 'a list -> 'b * 'a list))
  -> int -> 'b -> 'a list -> 'b * 'a list
```

To constrain the types so that typechecking yields the intuitive type requires some contortions. The following does it.

```
fun Parser (tab:symtab) : parser =                                    147
     fn m
      => fn t
          => fn [] => (t,[])
             |  (inp as next::rest) =>
                  let val (n,parsefn) = tab next
                  in if m>=n then (t,inp)
                             else parsefn (Parser tab) m t inp
                  end;
>  val Parser = fn : symtab -> parser
```

Standard ML seems rather worse than its predecessors in the flexibility it allows for writing type constraints.

Notice that every input item is supposed to have an entry in the symbol table. The kind of items that might be encountered include atoms, unary operators, binary operators, brackets (both opening and closing) and keywords associated with other kinds of constructs (e.g. `if`, `then`, `else`, `while`).

Generic functions to construct appropriate symbol table entries for these will now be described.

The parser is initially invoked with a specific symbol table, precedence `0` and `t` set to `Nil`.

The action associated with an atom, a say, is just to return `Atom` a. Since the atom may be the argument of some preceding function, whose parse tree will be bound to `t`, the parse tree that is actually returned by the parse action of an atom is `MkComb(t,Atom` a`)`.

```
fun ParseAtom parse p t (next::rest) =                                148
 parse p (MkComb(t,Atom next)) rest;
```

The action associated with an opening bracket is to recursively call the parser, check that there is a matching closing bracket, remove it, and then continue the parse. The function `ParseBracket` below is the action invoked by an opening bracket. It takes as a parameter the closing bracket it should check for. The parse tree `t` is combined, using `MkComb`, with the parse tree `t'` of the stuff parsed inside of the brackets. If `t` is `Nil` then the definition of `MkComb` ensures that `t'` becomes the new 'last-thing-parsed' bound to `t` in the rest of the parse. However if `t` is not `Nil`, then what is being parsed must have the form e_1 `(`e_2`)`, where `t` is the parse tree of e_1, so a combination is generated (namely, the parse tree of e_1 applied to the parse tree of e_2).

```
exception MissingClosingBracket;                                      149

fun ParseBracket close parse p t (_::rest) =
  let val (t', next'::rest') = parse 0 Nil rest
  in if close=next' then parse p (MkComb(t,t')) rest'
                    else raise MissingClosingBracket
  end;
```

One can ensure that the parsing initiated by an opening bracket will terminate at a closing bracket by giving the closing bracket a sufficiently low precedence in the symbol table (e.g. `0`). Closing brackets should always terminate the current parse, so it is an error to try to execute the parse acton associated with them in the symbol table (the type of the symbol table is such that all items have some action – in the case of closing brackets this should never actually be invoked).

```
exception TerminatorParseErr;                                         150

fun Terminator parse _ = raise TerminatorParseErr;
```

The next function provides a rather general way of specifying parser actions. The idea is that to parse a given kind of construct the parser is called recursively to get each constituent and then a node containing the resulting constituent parse trees is returned. Each recursive invokation of the parser might require some local checking for keywords etc. For example, to parse `if` e `then` e_1 `else` e_2 the parser is called to get the parse tree of the e, then the presence of `then` is checked (`then` must be a terminator) and it is removed, then the parser is invoked to get the parse tree for e_1, then the presence of `else` is checked (`else` must be a terminator) and it it removed, then the parser is invoked again to get the parse tree of e_2 and finally a node like `Node("COND", [t,t1,t2])` is returned.

The function `ParseSeq` below takes a constructor function `mktree` (for building a node), invokes the parser a number of times and then builds a parse tree by applying `mktree` to the resulting constituent parse trees.

Each invokation of the parser can be 'wrapped around' with some extra checking activity. This is specified by providing a list of functions of type `parser->parser`: applying such a function to a parser produces a new parser with the checking added on. The simplest case of this is no checking, which is specified by the identity function.

```
fun Id x = x;                                                          151
```

`ChkAft` is used to modify a parser to check that a given keyword occurs after the parser is invoked. If p:`parser` is a parser function, then `ChkAft` s p is a parser function that first invokes p, then checks for s and deletes it if found and raises an exception otherwise.

```
exception ChkAftErr;                                                   152

fun ChkAft s parse m t inp =
 case parse m t inp
 of   (t', s'::rest) => if s=s' then (t',rest) else raise ChkAftErr;
```

The function `ParseSeq` below takes a parse tree constructor function `mktree` of type `tree * tree list -> tree` for building a node. The first parse tree is the one passed as a parameter (`t`) to the parser and the list of parse trees are the constituents that have just been parsed.

Constructors for building parse trees of unary operator expressions and binary operator expressions are `MkUnop` and `MkBinop`, respectively.

Suppose u is a unary operator and consider e_1 u e_2: this should parse to `Comb(`$\hat{e}_1$`, Unop(`u`, `$\hat{e}_2$`))`, where $\hat{e}_1$ and $\hat{e}_2$ are the parse trees of e_1 and e_2, respectively. The parse tree constructor for unary operators is thus:

```
fun MkUnop unop (t,tl) = MkComb(t,Node(unop,tl));                      153
```

Suppose b is a binary operator and consider e_1 b e_2: this should parse to `Binop(`b`, `$\hat{e}_1$`, `$\hat{e}_2$`)`, where $\hat{e}_1$ and $\hat{e}_2$ are the parse trees of e_1 and e_2, respectively. The parse tree constructor for binary operators is thus:

```
fun MkBinop bnop (t,tl) = Node(bnop,t::tl);                            154
```

The function `ParseSeq` also takes as a parameter a list of parser transformations (e.g. `Id` or `ChkAft` f) and returns a parser that recursively invokes the parser once for each parser transformation and then builds a parse tree using `mktree` applied to the resulting constituent parse trees. For example, the parsing of conditionals is specified by:

```
[ChkAft "then", ChkAft "else", Id]
```

`ParseSeq` uses an auxiliary function `ParseSeqAux` that iterates down the list of supplied parse tree transformers invoking them in turn. The definitions are short, but admittedly cryptic! To try to improve their readability type constraints have been added to constrain excess polymophism. Without the constraints, `ParseSeq` gets the incomprehensibly general type:

```
(tree * 'a list -> 'b)
  -> int
     -> ((int -> 'b -> 'c list -> 'd)
         -> int -> tree -> 'c list -> 'a * 'c list) list
        -> (int -> 'b -> 'c list -> 'd) -> int -> tree -> 'c list -> 'd
```

with the constraints the type is:

```
(tree * tree list -> tree)
  -> int -> (parser -> parser) list -> parser -> parser
```

Unfortunately, as with `Parser`, it is necesary to go to some contortions to achieve this type constraint. Instead of writing:

```
ParseSeq mktree m fl parse n t (_::rest) = ...
```

It is necessary to write

```
fun ParseSeq mktree m fl parse =
 fn n => fn t => fn (_::rest) => ...
```

and then add the type constraints shown below.

```
fun ParseSeqAux m [f:parser->parser] (parse:parser) n inp =          155
     let val (t, rest1) = f parse m Nil inp
     in  ([t], rest1) end
 |  ParseSeqAux m (f::fl : (parser->parser)list) parse n inp =
     let val (t, rest1) = f parse 0 Nil inp
     in let val (l,rest2) = ParseSeqAux m fl parse 0 rest1
        in  (t::l, rest2) end
     end;

fun ParseSeq mktree m (fl:(parser->parser)list) (parse:parser) : parser =
 fn n => fn t => fn (_::rest) =>
  let val (l,rest1) = ParseSeqAux m fl parse n rest
  in parse n (mktree(t,l)) rest1 end;
```

A symbol table is a function of type `string -> int * (parser -> parser)`. Here is an example:

```
fun SymTab "*"    = (7,  ParseSeq (MkBinop "MULT") 8 [Id])            156
 |  SymTab "+"    = (6,  ParseSeq (MkBinop "ADD") 5 [Id])
 |  SymTab "~"    = (10, ParseSeq (MkUnop "MINUS") 9 [Id])
 |  SymTab "if"   = (10, ParseSeq (MkUnop "COND") 0 [ChkAft "then",
                                                     ChkAft "else",
                                                     Id])
 |  SymTab "("    = (10, ParseBracket ")")
 |  SymTab ")"    = (0,  Terminator)
 |  SymTab "then" = (0,  Terminator)
 |  SymTab "else" = (0,  Terminator)
 |  SymTab x      = (10, ParseAtom);
```

Notice that the left precedence of `*` is `7` which is the same as its right precedence. However, the left precedence of `+` is 6 which is greater than its right precedence `5`. The effect of this is to make `*` left associative and `+` right associative. In general, if the left precedence is less than or equal to the right precedence, then left associativity results, otherwise right associativity results.

The complete parser `P` defined below uses the lexical analyser `Lex` and the symbol table above. Assume the code for `Lex`, as described in Section 5.1, is in the file `Lex.ml`.

157

```
use "Lex.ml";

val P = Parser SymTab 0 Nil o Lex;
> val P = fn : string -> tree * string list

P "f if x then y + z else y * z";
> val it =
>   (Comb
>      (Atom "f",
>       Node
>         ("COND",
>          [Atom "x",Node ("ADD",[Atom "y",Atom "z"]),
>           Node ("MULT",[Atom "y",Atom "z"])])),[])
```

Chapter 6

Case study 2: the λ-calculus

It is assumed that integers and (unary and binary) operations over integers are primitive. The type `atom` packages these up into a single datatype. Both unary operator atoms (`Op1`) and binary operator atoms (`Op2`) have a name and a semantics.

158
```
datatype atom = Num of int
              | Op1 of string * (int->int)
              | Op2 of string * (int*int->int);
```

The application of an atomic operation to a value is defined by the function `ConApply` (see below). The application of a binary operator `b` to `m` results in a unary operator named `mb` expecting the other argument.

To convert the argument `m` to a string that can be concatenated with the name of the operator, a function to convert a number to a string giving its decimal representation is defined.

159
```
fun StringOfNum 0 = "0"
 |  StringOfNum 1 = "1"
 |  StringOfNum 2 = "2"
 |  StringOfNum 3 = "3"
 |  StringOfNum 4 = "4"
 |  StringOfNum 5 = "5"
 |  StringOfNum 6 = "6"
 |  StringOfNum 7 = "7"
 |  StringOfNum 8 = "8"
 |  StringOfNum 9 = "9"
 |  StringOfNum n =
     (StringOfNum(n div 10)) ^ (StringOfNum(n mod 10));

StringOfNum 1574;
> val it = "1574" : string
```

Now `ConApply` can be defined.

160
```
fun ConApply(Op1(_,f1), Num m) = Num(f1 m)
 |  ConApply(Op2(x,f2), Num m) = Op1((StringOfNum m^x), fn n => f2(m,n));
> val ConApply = fn : atom * atom -> atom

ConApply(Op2("+",op +), Num 2);
> val it = Op1 ("2+",fn) : atom

ConApply(it, Num 3);
> val it = Num 5 : atom
```

λ-expressions are represented by the datatype `lam`.

161
```
datatype lam = Var of string
             | Con of atom
             | App of (lam * lam)
             | Abs of (string * lam);
```

6.1 A λ-calculus parser

It is convenient to have a λ-calculus parser. Assume the code of the parser described in Section 5.3 is in the file `Parser.ml`.

```
use "Parser.ml";                                                    162
```

A sutable symbol table for the λ-calculus is:

```
fun LamSymTab "*"    = (7,  ParseSeq (MkBinop "MULT") 7 [Id])       163
 |  LamSymTab "+"    = (6,  ParseSeq (MkBinop "ADD") 6 [Id])
 |  LamSymTab "."    = (0,  Terminator)
 |  LamSymTab "\\"   = (10, ParseSeq(MkUnop "Abs") 0 [ChkAft ".", Id])
 |  LamSymTab "("    = (10, ParseBracket ")")
 |  LamSymTab ")"    = (0,  Terminator)
 |  LamSymTab x      = (10, ParseAtom);
```

Note that `"\\"` is our ASCII representation of λ. This is actually just a single backslash; the first one is the escape character needed to include the second one in the string!

The following function lexically analyses and then parses a string (recall that `Parser` returns a parse tree and the remaining input).

```
fun ParseLam s =                                                    164
 let val (t,[]) = Parser LamSymTab 0 Nil (Lex s)
 in t end;
> Warning: binding not exhaustive
> val ParseLam = fn : string -> tree

ParseLam "(\\x.x+1) 200";
> val it =
>   Comb (Node ("Abs",[Atom "x",Node ("ADD",[Atom "x",Atom "1"])]),
>         Atom "200")
>  : tree
```

The output of `ParseLam` is an element of the general purse parse tree type `tree` defined on page 82. This is easily converted to type `lam`. A function for testing whether a string represents a number (i.e. is a string of digits) is needed.

```
fun IsNumber s =                                                    165
 let fun TestDigList [] = true
      |  TestDigList (x::l) = IsDigit x andalso TestDigList l
 in TestDigList(explode s)
 end;
> val IsNumber = fn : string -> bool
```

If a string represents a number then the following provides a way of converting it to a number (i.e. value of type `int`).

166

```
fun DigitVal "0" = 0
 |  DigitVal "1" = 1
 |  DigitVal "2" = 2
 |  DigitVal "3" = 3
 |  DigitVal "4" = 4
 |  DigitVal "5" = 5
 |  DigitVal "6" = 6
 |  DigitVal "7" = 7
 |  DigitVal "8" = 8
 |  DigitVal "9" = 9;
> Warning: match nonexhaustive
> val DigitVal = fn : string -> int

fun NumOfString s =
 let fun ListVal [] = 0
      |  ListVal (x::l) = DigitVal x +  10 * (ListVal l)
 in ListVal(rev(explode s))
 end;
> val NumOfString = fn : string -> int

NumOfString "2001";
> val it = 2001 : int
```

Armed with this string-to-number converter, it is routine to convert values of type `tree` to values of type `lam`. The fourth clause of the definition of `Convert` below is a little hack to make: `\x1 x2 ... xn. e` parse as: `\x1.\x2. ... \xn. e`. This hack makes use of the fact that sequences of variables parse as left-associated applications.

167

```
ParseLam "\\x y z. w";
> val it = Node ("Abs",[Comb (Comb (Atom "x",Atom "y"),Atom "z"),Atom "w"])
```

168

```
fun Convert (Atom x) =
     if IsNumber x then Con(Num(NumOfString x)) else Var x
 |  Convert (Comb(a,b)) =
     App(Convert a, Convert b)
 |  Convert (Node("Abs",[Atom x, a])) =
     Abs(x,Convert a)
 |  Convert (Node("Abs",[Comb(a1, Atom x), a2])) =
     Convert(Node("Abs",[a1, Node("Abs",[Atom x,a2])]))
 |  Convert (Node("ADD",[a,b])) =
     App(App(Con(Op2("+",(op+))), Convert a), Convert b);
> Warning: match nonexhaustive
> val Convert = fn : tree -> lam
```

The function `PL` (for "Parse Lambda expression") parses a string and then converts it to a value of type `lam`.

169

```
val PL = Convert o ParseLam;
> val PL = fn : string -> lam

PL "x+y";
> val it = App (App (Con (Op2 fn),Var "x"),Var "y") : lam

PL "(\\x.x+y) y";
> val it =
>  App (Abs ("x",App (App (Con (Op2 fn),Var "x"),Var "y")),
>       Var "y")
>  : lam
```

Here is the fixed-point operator **Y** (see Section 2.4):

```
PL "\\f. (\\x f.(\\z.x x f)) (\\x f.(\\z.x x f))";                    170
> val it =
>   Abs
>    ("f",
>     App
>      (Abs ("x",Abs ("f",Abs ("z",App (App (Var "x",Var "x"),Var "f")))),
>       Abs ("x",Abs ("f",Abs ("z",App (App (Var "x",Var "x"),Var "f"))))))
>   : lam
```

An 'unparser' (or 'pretty-printer') will be useful for viewing elements of type `lam`. The one that follows (`UPL`) is rather crude – for example, it does not attempt to format expressions across lines, though it does at leat avoid putting brackets around variables.

The name `UPL` stands for "UnParse Lambda expression" and `BUPL` for "Bracket and UnParse Lambda expression".

```
fun UPL (Var x)                             = x                           171
 |  UPL (Con(Num n))                        = StringOfNum n
 |  UPL (Con(Op1(x,_)))                     = x
 |  UPL (Con(Op2(x,_)))                     = x
 |  UPL (App(Con(Op1(x,_)),e))              = x ^ " " ^ BUPL e
 |  UPL (App(App(Con(Op2(x,_)),e1),e2))     = BUPL e1 ^ x ^ BUPL e2
 |  UPL (App(e1,e2))                        = UPL e1 ^ " " ^ BUPL e2
 |  UPL (Abs(x,e))                          = "(\\" ^ x ^ ". " ^ UPL e ^ ")"
and BUPL(Var x)      = x
 |  BUPL(Con(Num n)) = StringOfNum n
 |  BUPL e           = "(" ^ UPL e ^ ")";
```

6.2 Implementing substitution

Recall the definition of substitution on page 10.

E	$E\,[E'/V]$
V	E'
V' (where $V \neq V'$)	V'
$E_1\ E_2$	$E_1\,[E'/V]\ E_2\,[E'/V]$
$\lambda V.\ E_1$	$\lambda V.\ E_1$
$\lambda V'.\ E_1$ (where $V \neq V'$ and V' is not free in E')	$\lambda V'.\ E_1\,[E'/V]$
$\lambda V'.\ E_1$ (where $V \neq V'$ and V' is free in E')	$\lambda V''.\ E_1\,[V''/V']\,[E'/V]$ where V'' is a variable not free in E' or E_1

This is easily implemented in ML. Some auxiliary set-theoretic functions on lists are needed (some of which have been met before). First a test for membership.

```
fun Mem x [] = false                                                      172
 |  Mem x (x'::s) = (x=x') orelse Mem x s;
> val Mem = fn : ''a -> ''a list -> bool
```

Note that the union of two lists defined below does not introduce duplicates.

```
fun Union [] l = l                                                    173
 |  Union (x::l1) l2 =
     if Mem x l2 then Union l1 l2 else x::(Union l1 l2);
> val Union = fn : ''a list -> ''a list -> ''a list

Union [1,2,3,4,5] [2,3,4,5,6,7];
> val it = [1,2,3,4,5,6,7] : int list
```

`Subtract l1 l2` removes all members of `l2` from `l1` (i.e. is '`l1` minus `l2`').

```
fun Subtract [] l = []                                                174
 |  Subtract (x::l1) l2 =
     if Mem x l2 then Subtract l1 l2 else x::(Subtract l1 l2);
> val Subtract = fn : ''a list -> ''a list -> ''a list

Subtract [1,2,3,4,5] [3,4,5,6];
> val it = [1,2] : int list
```

Using `Mem`, `Union` and `Subtract` the function `Frees` to compute a list of the free variables in a λ-expression is easily defined.

```
fun Frees (Var x) = [x]                                               175
 |  Frees (Con c) = []
 |  Frees (App(e1,e2)) = Union (Frees e1) (Frees e2)
 |  Frees (Abs(x,e)) = Subtract (Frees e) [x];
> val Frees = fn : lam -> string list

PL "\\x.x+y";
> val it = Abs ("x",App (App (Con (Op2 fn),Var "x"),Var "y")) : lam

Frees it;
> val it = ["y"] : string list
```

Substitution needs to rename variables to avoid 'capture'. This will be done by priming them.

```
fun Prime x = x^"'";                                                  176

Prime "x";
> val it = "x'" : string
```

`Variant xl x` primes `x` sufficient number of times so that the result does not occur in the list `xl`.

```
fun Variant xl x =                                                    177
 if Mem x xl then Variant xl (Prime x) else x;
> val Variant = fn : string list -> string -> string

Variant ["x","y","z","y'","w"] "y";
> val it = "y''" : string
```

Now, at last, substitution can be defined: `Subst E E' V` computes `E[E'/V]` according to the table above.

```
fun Subst (e as Var x') e' x = if x=x' then e' else e                178
 |  Subst (e as Con c) e' x =  e
 |  Subst (App(e1, e2)) e' x =  App(Subst e1 e' x, Subst e2 e' x)
 |  Subst (e as Abs(x',e1)) e' x =
     if x=x' then e
             else if Mem x' (Frees e')
                   then let val x'' = Variant (Frees e' @ Frees e1) x'
                        in Abs(x'', Subst(Subst e1 (Var x'') x') e' x)
                        end
                   else Abs(x', Subst e1 e' x);
> val Subst = fn : lam -> lam -> string -> lam
```

Here are some examples:

```
Subst (PL"(\\x.x+y) x") (PL"1") "x";                                 179
> val it =
>  App (Abs ("x",App (App (Con (Op2 fn),Var "x"),Var "y")),
>       Con (Num 1))
>   : lam

UPL it;
> val it = "(\\x. x+y)(1)" : string

UPL(Subst (PL"\\x.x+y") (PL"x+1") "y");
> val it = "(\\x'. x'+(x+1))" : string
```

A function `EvalN` can now be defined to do normal order reduction (sometimes called 'call-by-name' [30]). Note that the evaluation does not 'go inside' λ-bodies, so does not compute normal forms.

```
fun EvalN (e as Var _ ) = e                                          180
 |  EvalN (e as Con _) = e
 |  EvalN (Abs(x,e)) = Abs(x, e)
 |  EvalN (App(Con a1, Con a2)) = Con(ConApply(a1,a2))
 |  EvalN (App(e1,e2)) =
     case EvalN e1
     of   (Abs(x,e3))     => EvalN(Subst e3 e2 x)
     |    (e1' as Con a1) => (case EvalN e2
                              of   (Con a2) => Con(ConApply(a1,a2))
                              |    e2'      => App(e1',e2'))
     |    e1'         => App(e1', EvalN e2);
> val EvalN = fn : lam -> lam
```

Here is a typical example that only terminates with normal order evaluation:

```
EvalN (PL"(\\x.1) ((\\x. x x) (\\x. x x))");                         181
> val it = Con (Num 1) : lam
```

val true = ”
x y z.w”

6.3 The SECD machine

Call-by-value evaluation can be programmed with the function `EvalV`.

```
fun EvalV (e as Var _)    = e                                    182
 |  EvalV (e as Con _)    = e
 |  EvalV (e as Abs(_,_)) = e
 |  EvalV (App(e1,e2))    =
     let val e2' = EvalV e2
     in
     (case EvalV e1
      of   (Abs(x,e3)) => EvalV(Subst e3 e2' x)
      |    (e1' as Con a) => (case e2'
                               of   (Con a2) => Con(ConApply(a1,a2))
                               |    _        => App(e1',e2'))
      |    e1'         => App(e1',e2'))
     end;
```

The SECD machine is a classical virtual machine for reducing λ-expressions using call-by-value. It was developed in the 1960's by Peter Landin and has been analysed by Gordon Plotkin [30]. Various more recent practical vitual machines for ML are descendents of the SECD machine.

The name SECD comes from Stack, Environment, Control and Dump which are the four components of the machine state.

The stack of an SECD machine holds a sequence (represented by a list) of atoms and closures. The environment provides values of variables. A closure is an abstraction paired with an environment. The mutually recursive datatypes of `item` and `env` represent items and environments, respectively. Environments are represented by association lists of variables (represented by strings) and and items.

```
datatype item = Atomic of atom                                   183
              | Closure of (lam * env)
and      env  = EmptyEnv
              | Env of string * item * env;
```

The function `Lookup` looks up the value of a variable in an environment (and raises an exception is the variable doesn't have a variable).

```
exception LookupErr;                                             184

fun Lookup(s,EmptyEnv)       = raise LookupErr
 |  Lookup(s,Env(s',i,env))  = if s=s' then i else Lookup(s,env);
```

The control of an SECD is a sequence (represented by a list) of instructions which are either the special operation `Ap` or a λ-expression.

```
datatype instruction = Ap | Exp of lam;                          185
```

A datatype `state` that represents SECD machine states can now be defined.

```
type stack   = item list                                         186
and  control = instruction list;

datatype state = NullState
               | State of (stack * env * control * state);
```

The transitions of the SECD machine are given by the function `Step`.

```
fun Step(State(v::S, E, [], State(S',E',C',D'))) =                    187
      State(v::S', E', C', D')
 |  Step(State(S, E, Exp(Var x)::C, D)) =
      State(Lookup(x,E)::S, E, C, D)
 |  Step(State(S, E, Exp(Con v)::C, D)) =
      State(Atomic v::S, E, C, D)
 |  Step(State(S, E, Exp(Abs(x,e))::C, D )) =
      State(Closure(Abs(x,e),E)::S, E, C, D)
 |  Step(State(Closure(Abs(x,e),E')::(v::S), E, Ap::C, D)) =
      State([], Env(x,v,E'), [Exp e], State(S,E,C,D))
 |  Step(State(Atomic v1::(Atomic v2::S), E, Ap::C, D )) =
      State(Atomic(ConApply(v1,v2))::S, E, C, D)
 |  Step(State(S, E, Exp(App(e1,e2))::C, D)) =
      State(S, E, Exp e2::(Exp e1::(Ap::C)), D);
```

The function `Run` iterates `Step` until a final state is reached and then returns a list of all the intermediate states.

```
fun Run(state as State([_],EmptyEnv,[],NullState)) = [state]          188
 |  Run state = state::Run(Step state);
> val Run = fn : state -> state list
```

The function `Eval` takes a λ-expression and evaluates it using the SECD machine.

```
fun Eval e =                                                          189
 let fun EvalAux(State([v],EmptyEnv,[],NullState)) = v
      |  EvalAux state = EvalAux(Step state)
 in EvalAux(State([],EmptyEnv,[Exp e],NullState)) end;
> val Eval = fn : lam -> item
```

`Load` loads a lambda-expression into an SECD state ready for running.

```
fun Load e = State([],EmptyEnv,[Exp e],NullState);                    190
> val Load = fn : lam -> state
```

`SECDRun` parses a string, loads the resulting λ-expression into an SECD state and then runs the result. `SECDEval` is similar, but it just `Eval`s the result.

```
fun SECDRun s = Run(Load s);                                          191
> val SECDRun = fn : lam -> state list

fun SECDEval s = Eval(PL s);
> val SECDEval = fn : string -> item

SECDEval "(\\x.\\y. x+y) 1 2";
> val it = Atomic (Num 3) : item
```

Bibliography

[1] Augustsson, L., 'A compiler for lazy ML', in *Proceedings of the ACM Symposium on LISP and Functional Programming*, Austin, pp. 218-227, 1984.

[2] Barendregt, H.P., *The Lambda Calculus* (revised edition), Studies in Logic 103, North-Holland, Amsterdam, 1984.

[3] Barron, D.W. and Strachey, C. 'Programming', in Fox, L. (ed.), *Advances in Programming and Non-numerical Computation* (Chapter 3), Pergamon Press, 1966.

[4] Bird, R. and Wadler, P., *An Introduction to Functional Programming*, Prentice Hall, 1988.

[5] Boyer, R.S. and Moore, J S., *A Computational Logic*, Academic Press, 1979.

[6] De Bruijn, N.G., 'Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation', *Indag. Math.*, **34**, pp. 381-392, 1972.

[7] Burge, W., *Recursive Programming Techniques*, Addison-Wesley, 1975.

[8] Clarke, T.J.W., et al., 'SKIM – the S, K, I Reduction Machine', in *Proceedings of the 1980 ACM LISP Conference*, pp. 128-135, 1980.

[9] Curry, H.B. and Feys, R., *Combinatory Logic, Vol. I*, North Holland, Amsterdam, 1958.

[10] Curry, H.B., Hindley, J.R. and Seldin, J.P. *Combinatory Logic, Vol. II*, Studies in Logic 65, North Holland, Amsterdam, 1972.

[11] Fairbairn, J. and Wray, S.C., 'Code generation techniques for functional languages', in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, Cambridge, Mass., pp. 94-104, 1986.

[12] Gordon, M.J.C., 'On the power of list iteration', *The Computer Journal*, **22**, No. 4, 1979.

[13] Gordon, M.J.C.,*The Denotational Description of Programming Languages*, Springer-Verlag, 1979.

[14] Gordon, M.J.C., *Programming Language Theory and its Implementation*, Prentice Hall International Series in Computer Science, 1988 (out of print).

[15] Gordon, M.J.C., Milner, A.J.R.G. and Wadsworth, C.P., *Edinburgh LCF: a mechanized logic of computation*, Springer Lecture Notes in Computer Science, Springer-Verlag, 1979.

[16] Henderson, P., *Functional Programming, Application and Implementation*, Prentice Hall, 1980.

[17] Henderson, P. and Morris, J.M., 'A lazy evaluator', in *Proceedings of The Third Symposium on the Principles of Programming Languages, Atlanta, Georgia*, pp. 95-103, 1976.

[18] Hindley, J.R., 'Combinatory reductions and lambda-reductions compared', *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, **23**, pp. 169-180, 1977.

[19] Hindley, J.R. and Seldin, J.P., *Introduction to Combinators and λ-Calculus*, London Mathematical Society Student Texts, 1, Cambridge University Press, 1986.

[20] Hughes, R.J.M., 'Super combinators: a new implementation method for applicative languages', in *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh*, 1982.

[21] Kleene, S.C., 'λ-definability and recursiveness', *Duke Math. J.*, pp. 340-353, 1936.

[22] Krishnamurthy, E.V. and Vickers, B.P., 'Compact numeral representation with combinators', *The Journal of Symbolic Logic*, **52**, No. 2, pp. 519-525, June 1987.

[23] Lamport, L., *LaTeX: A Document Preparation System*, Addison-Wesley, 1986.

[24] Landin, P.J., 'The next 700 programming languages', *Comm. Assoc. Comput. Mach.*, **9**, pp. 157-164, 1966.

[25] Levy, J.-J., 'Optimal reductions in the lambda calculus', in Hindley, J.R. and Seldin, J.P. (eds), *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, Academic Press, New York and London, 1980.

[26] Mauny, M. and Suárez, A., 'Implementing functional languages in the categorical abstract machine', in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pp. 266-278, Cambridge, Mass., 1986.

[27] Milner, A.J.R.G., 'A proposal for Standard ML', in *Proceedings of the ACM Symposium on LISP and Functional Programming, Austin*, 1984.

[28] Morris, J.H., *Lambda Calculus Models of Programming Languages*, Ph.D. Dissertation, M.I.T., 1968.

[29] Peyton Jones, S.L., *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

[30] Plotkin, G.D., 'Call-by-name, call-by-value and the λ-calculus', *Theoretical Computer Science*, **1**, pp 125–159, 1975.

[31] Schönfinkel, M., 'Über die Bausteine der mathematischen Logik', *Math. Annalen* **92**, pp. 305-316, 1924. Translation printed as 'On the building blocks of mathematical logic', in van Heijenoort, J. (ed.), *From Frege to Gödel*, Harvard University Press, 1967.

[32] Scott, D.S., 'Models for various type free calculi', in Suppes, P. et al. (eds), *Logic, Methodology and Philosophy of ScienceIV*, Studies in Logic 74, North-Holland, Amsterdam, 1973.

[33] Stoy, J.E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, M.I.T. Press, 1977.

[34] Turner, D.A., 'A new implementation technique for applicative languages', *Software Practice and Experience*, **9**, pp. 31-49, 1979.

[35] Turner, D.A., 'Another algorithm for bracket abstraction', *The Journal of Symbolic Logic*, **44**, No. 2, pp. 267-270, June 1979.

[36] Turner, D.A., 'Functional programs as executable specifications', in Hoare, C.A.R. and Shepherdson, J.C. (eds), *Mathematical Logic and Programming Languages*, Prentice Hall, 1985.

[37] Wadsworth, C.P., 'The relation between computational and denotational properties for Scott's D_∞-models of the lambda-calculus', *S.I.A.M. Journal of Computing*, **5**, pp. 488-521, 1976.

[38] Wadsworth, C.P., 'Some unusual λ-calculus numeral systems', in Hindley, J.R. and Seldin, J.P. (eds), *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, Academic Press, New York and London, 1980.