Programming in Standard ML

(WORKING DRAFT OF AUGUST 29, 2002.)

Robert Harper
Carnegie Mellon University

Spring Semester, 2001

Preface

This book is an introduction to programming with the Standard ML programming language. It began life as a set of lecture notes for Computer Science 15–212: Principles of Programming, the second semester of the introductory sequence in the undergraduate computer science curriculum at Carnegie Mellon University. It has subsequently been used in many other courses at Carnegie Mellon, and at a number of universities around the world. It is intended to supersede my *Introduction to Standard ML*, which has been widely circulated over the last ten years.

Standard ML is a formally defined programming language. *The Definition of Standard ML (Revised)* is the formal definition of the language. It is supplemented by the *Standard ML Basis Library*, which defines a common basis of types that are shared by all implementations of the language. *Commentary on Standard ML* discusses some of the decisions that went into the design of the first version of the language.

There are several implementations of Standard ML available for a wide variety of hardware and software platforms. The best-known compilers are *Standard ML of New Jersey*, *Moscow ML*, *MLKit*, and *PolyML*. These are all freely available on the worldwide web. Please refer to *The Standard ML Home Page* for up-to-date information on Standard ML and its implementations.

Numerous people have contributed directly and indirectly to this text. I am especially grateful to the following people for their helpful comments and suggestions: Marc Bezem, Franck van Breugel, Karl Crary, Mike Erdmann, Matthias Felleisen, Joel Jones, John Lafferty, Frank Pfenning, Chris Stone, Dave Swasey, Johan Wallen, Scott Williams, and Jeannette Wing. I am also grateful to the many students of 15-212 who used these notes and sent in their suggestions over the years.

These notes are a work in progress. Corrections, comments and suggestions are most welcome.

Contents

Part I

Overview

Standard ML is a type-safe programming language that embodies many innovative ideas in programming language design. It is a statically typed language, with an extensible type system. It supports polymorphic type inference, which all but eliminates the burden of specifying types of variables and greatly facilitates code re-use. It provides efficient automatic storage management for data structures and functions. It encourages functional (effect-free) programming where appropriate, but allows imperative (effect-ful) programming where necessary. It facilitates programming with recursive and symbolic data structures by supporting the definition of functions by pattern matching. It features an extensible exception mechanism for handling error conditions and effecting non-local transfers of control. It provides a richly expressive and flexible module system for structuring large programs, including mechanisms for enforcing abstraction, imposing hierarchical structure, and building generic modules. It is portable across platforms and implementations because it has a precise definition. It provides a portable standard basis library that defines a rich collection of commonly-used types and routines.

Many implementations go beyond the standard to provide experimental language features, extensive libraries of commonly-used routines, and useful program development tools. Details can be found with the documentation for your compiler, but here's some of what you may expect. Most implementations provide an interactive system supporting on-line program development, including tools for compiling, linking, and analyzing the behavior of programs. A few implementations are "batch compilers" that rely on the ambient operating system to manage the construction of large programs from compiled parts. Nearly every compiler generates native machine code, even when used interactively, but some also generate code for a portable abstract machine. Most implementations support separate compilation and provide tools for managing large systems and shared libraries. Some implementations provide tools for tracing and stepping programs; many provide tools for time and space profiling. Most implementations supplement the standard basis library with a rich collection of handy components such as dictionaries, hash tables, or interfaces to the ambient operating system. Some implementations support language extensions such as support for concurrent programming (using message-passing or locking), richer forms of modularity constructs, and support for "lazy" data structures.

Chapter 1

Programming in Standard ML

1.1 A Regular Expression Package

To develop a feel for the language and how it is used, let us consider the implementation of a package for matching strings against regular expressions. We'll structure the implementation into two modules, an implementation of regular expressions themselves and an implementation of a matching algorithm for them.

These two modules are concisely described by the following *signatures*.

```
signature REGEXP = sig
  datatype regexp =
    Zero | One | Char of char |
    Plus of regexp * regexp |
    Times of regexp * regexp |
    Star of regexp
  exception SyntaxError of string
  val parse : string -> regexp
  val format : regexp -> string
end
signature MATCHER = sig
  structure RegExp : REGEXP
  val match : RegExp.regexp -> string -> bool
end
```

The signature `REGEXP` describes a module that implements regular expressions. It consists of a description of the abstract syntax of regular expressions, together with operations for parsing and unparsing them. The signature MATCHER describes a module that implements a matcher for a given notion of regular expression. It contains a function match that, when given a regular expression, returns a function that determines whether or not a given string matches that expression. Obviously the matcher is dependent on the implementation of regular expressions. This is expressed by a *structure specification* that specifies a hierarchical dependence of an implementation of a matcher on an implementation of regular expressions — any implementation of the MATCHER signature must include an implementation of regular expressions as a constituent module. This ensures that the matcher is self-contained, and does not rely on implicit conventions for determining *which* implementation of regular expressions it employs.

The definition of the abstract syntax of regular expressions in the signature REGEXP takes the form of a *datatype declaration* that is reminiscent of a context-free grammar, but which abstracts from matters of lexical presentation (such as precedences of operators, parenthesization, conventions for naming the operators, *etc.*.) The abstract syntax consists of six clauses, corresponding to the regular expressions $\mathbf{0}$, $\mathbf{1}$, $\mathbf{a}$, $r_1 + r_2$, $r_1\, r_2$, and r^*.[1] The functions parse and format specify the parser and unparser for regular expressions. The parser takes a string as argument and yields a regular expression; if the string is ill-formed, the parser raises the exception SyntaxError with an associated string describing the source of the error. The unparser takes a regular expression and yields a string that parses to that regular expression. In general there are many strings that parse to the same regular expressions; the unparser generally tries to choose one that is easiest to read.

The implementation of the matcher consists of two modules: an implementation of regular expressions and an implementation of the matcher itself. An implementation of a signature is called a *structure*. The implementation of the matching package consists of two structures, one implementing the signature REGEXP, the other implementing MATCHER. Thus the overall package is implemented by the following two *structure declarations*:

```
structure RegExp :> REGEXP = ...
structure Matcher :> MATCHER = ...
```

The structure identifier RegExp is bound to an implementation of the REGEXP signature. Conformance with the signature is ensured by the *ascription* of the signature REGEXP to the binding of RegExp using the ":>" notation. Not only does this check that the implementation (which has been elided

[1] Some authors use $\emptyset$ for $\mathbf{0}$ and ε for $\mathbf{1}$.

here) conforms with the requirements of the signature `REGEXP`, but it also ensures that subsequent code cannot rely on any properties of the implementation other than those explicitly specified in the signature. This helps to ensure that modules are kept separate, facilitating subsequent changes to the code.

Similarly, the structure identifier `Matcher` is bound to a structure that implements the matching algorithm in terms of the preceding implementation `RegExp` of `REGEXP`. The ascribed signature specifies that the structure `Matcher` must conform to the requirements of the signature `MATCHER`. Notice that the structure `Matcher` refers to the structure `RegExp` in its implementation.

Once these structure declarations have been processed, we may use the package by referring to its components using *paths*, or *long identifiers*. The function `Matcher.match` has type

```
Matcher.RegExp.regexp -> string -> bool,
```

reflecting the fact that it takes a regular expression *as implemented within the package itself* and yields a matching function on strings. We may build a regular expression by applying the parser, `Matcher.RegExp.parse`, to a string representing a regular expression, then passing the resulting value of type `Matcher.RegExp.regexp` to `Matcher.match`.[2]

Here's an example of the matcher in action:

```
val regexp =
    Matcher.RegExp.parse "(a+b)*"
val matches =
    Matcher.match regexp

val ex1 = matches "aabba"     (* yields true *)
val ex2 = matches "abac"      (* yields false *)
```

The use of long identifiers can get tedious at times. There are two typical methods for alleviating the burden. One is to introduce a synonym for a long package name. Here's an example:

[2]It might seem that one can apply `Matcher.match` to the output of `RegExp.parse`, since `Matcher.RegExp.parse` is just `RegExp.parse`. However, this relationship is not stated in the interface, so there is a *pro forma* distinction between the two. See Chapter 22 for more information on the subtle issue of *sharing*.

```
structure M = Matcher
structure R = M.RegExp
val regexp = R.parse "((a + %).(b + %))*"
val matches = M.match regexp
val ex1 = matches "aabba"
val ex2 = matches "abac"
```

Another is to "open" the structure, incorporating its bindings into the current environment:

```
open Matcher Matcher.RegExp
val regexp = parse "(a+b)*"
val matches = match regexp
val ex1 = matches "aabba"
val ex2 = matches "abac"
```

It is advisable to be sparing in the use of `open` because it is often hard to anticipate exactly which bindings are incorporated into the environment by its use.

Now let's look at the internals of the structures `RegExp` and `Matcher`. Here's a bird's eye view of `RegExp`:

```
structure RegExp :> REGEXP = struct
  datatype regexp =
    Zero | One | Char of char |
    Plus of regexp * regexp |
    Times of regexp * regexp |
    Star of regexp
  ⋮
  fun tokenize s = ...
  ⋮
  fun parse s =
      let
          val (r, s') =
            parse_rexp (tokenize (String.explode s))
      in
          case s' of
            nil => r
          | _ => raise SyntaxError "Bad input."
      end
```

```
      handle LexicalError =>
        raise SyntaxError "Bad input."
  ⋮
  fun format r =
      String.implode (format_exp r)
end
```

The elision indicates that portions of the code have been omitted so that we can get a high-level view of the structure of the implementation.

The structure `RegExp` is bracketed by the keywords `struct` and `end`. The type `regexp` is implemented precisely as specified by the `datatype` declaration in the signature `REGEXP`. The parser is implemented by a function that, when given a string, "explodes" it into a list of characters, transforms the character list into a list of "tokens" (abstract symbols representing lexical atoms), and finally parses the resulting list of tokens to obtain its abstract syntax. If there is remaining input after the parse, or if the tokenizer encountered an illegal token, an appropriate syntax error is signalled. The formatter is implemented by a function that, when given a piece of abstract syntax, formats it into a list of characters that are then "imploded" to form a string. The parser and formatter work with character lists, rather than strings, because it is easier to process lists incrementally than it is to process strings.

It is interesting to consider in more detail the structure of the parser since it exemplifies well the use of pattern matching to define functions. Let's start with the tokenizer, which we present here *in toto*:

```
datatype token =
  AtSign | Percent | Literal of char |
  PlusSign | TimesSign |
  Asterisk | LParen | RParen

exception LexicalError

fun tokenize nil = nil
  | tokenize (#"+" :: cs) = PlusSign :: tokenize cs
  | tokenize (#"." :: cs) = TimesSign :: tokenize cs
  | tokenize (#"*" :: cs) = Asterisk :: tokenize cs
  | tokenize (#"(" :: cs) = LParen :: tokenize cs
  | tokenize (#")" :: cs) = RParen :: tokenize cs
  | tokenize (#"@" :: cs) = AtSign :: tokenize cs
  | tokenize (#"%" :: cs) = Percent :: tokenize cs
  | tokenize (#"\\" :: c :: cs) =
```

```
    Literal c :: tokenize cs
  | tokenize (#"\\" :: nil) = raise LexicalError
  | tokenize (#" " :: cs) = tokenize cs
  | tokenize (c :: cs) = Literal c :: tokenize cs
```

The symbol "`@`" stands for the empty regular expression and the symbol "`%`" stands for the regular expression accepting only the null string. Concatentation is indicated by "`.`", alternation by "`+`", and iteration by "`*`".

We use a datatype declaration to introduce the type of tokens corresponding to the symbols of the input language. The function `tokenize` has type `char list -> token list`; it transforms a list of characters into a list of tokens. It is defined by a series of clauses that dispatch on the first character of the list of characters given as input, yielding a list of tokens. The correspondence between characters and tokens is relatively straightforward, the only non-trivial case being to admit the use of a backslash to "quote" a reserved symbol as a character of input. (More sophisticated languages have more sophisticated token structures; for example, words (consecutive sequences of letters) are often regarded as a single token of input.) Notice that it is quite natural to "look ahead" in the input stream in the case of the backslash character, using a pattern that dispatches on the first two characters (if there are such) of the input, and proceeding accordingly. (It is a lexical error to have a backslash at the end of the input.)

Let's turn to the parser. It is a simple recursive-descent parser implementing the precedence conventions for regular expressions given earlier. These conventions may be formally specified by the following grammar, which not only enforces precedence conventions, but also allows for the use of parenthesization to override them.

$$
\begin{array}{rcl}
rexp & ::= & rfac \mid rtrm\texttt{+}rexp \\
rtrm & ::= & rfac \mid rfac\,\texttt{.}\,rtrm \\
rfac & ::= & ratm \mid ratm\,\texttt{*} \\
ratm & ::= & \texttt{@} \mid \texttt{\%} \mid a \mid \texttt{(}rexp\texttt{)}
\end{array}
$$

The implementation of the parser follows this grammar quite closely. It consists of four mutually recursive functions, `parse_rexp`, `parse_rtrm`, `parse_rfac`, and `parse_ratm`. These implement what is known as a *recursive descent* parser that dispatches on the head of the token list to determine how to proceed.

```
fun parse_rexp ts =
    let
```

```
        val (r, ts') = parse_rtrm ts
    in
        case ts'
          of (PlusSign :: ts'') =>
             let
                 val (r', ts''') = parse_rexp ts''
             in
                (Plus (r, r'), ts''')
             end
           | _ => (r, ts')
    end
and parse_rtrm ts = ...
and parse_rfac ts =
    let
        val (r, ts') = parse_ratm ts
    in
        case ts'
        of (Asterisk :: ts'') => (Star r, ts'')
         | _ => (r, ts')
    end
and parse_ratm nil =
      raise SyntaxError ("No atom")
  | parse_ratm (AtSign :: ts) = (Zero, ts)
  | parse_ratm (Percent :: ts) = (One, ts)
  | parse_ratm ((Literal c) :: ts) = (Char c, ts)
  | parse_ratm (LParen :: ts) =
    let
        val (r, ts') = parse_rexp ts
    in
        case ts'
          of (RParen :: ts'') => (r, ts'')
           | _ =>
             raise SyntaxError "No close paren"
    end
```

Notice that it is quite simple to implement "lookahead" using patterns that inspect the token list for specified tokens. This parser makes no attempt to recover from syntax errors, but one could imagine doing so, using standard techniques.

This completes the implementation of regular expressions. Now for the matcher. The matcher proceeds by a recursive analysis of the regular expression. The main difficulty is to account for concatenation — to match a string against the regular expression $r_1\, r_2$ we must match some initial segment against r_1, then match the corresponding final segment against r_2. This suggests that we generalize the matcher to one that checks whether some initial segment of a string matches a given regular expression, then passes the remaining final segment to a *continuation*, a function that determines what to do after the initial segment has been successfully matched. This facilitates implementation of concatentation, but how do we ensure that at the outermost call the entire string has been matched? We achieve this by using an *initial continuation* that checks whether the final segment is empty.

Here's the code, written as a structure implementing the signature `MATCHER`:

```
structure Matcher :> MATCHER =
struct
  structure RegExp = RegExp
  open RegExp
  fun match_is Zero _ k = false
    | match_is One cs k = k cs
    | match_is (Char c) nil _ = false
    | match_is (Char c) (d::cs) k = (c=d) andalso (k cs)
    | match_is (Plus (r1, r2)) cs k =
      match_is r1 cs k  orelse  match_is r2 cs k
    | match_is (Times (r1, r2)) cs k =
      match_is r1 cs (fn cs' => match_is r2 cs' k)
    | match_is (Star r) cs k =
      k cs  orelse
      match_is r cs (fn cs' => match_is (Star r) cs' k)
  fun match regexp string =
      match_is
       regexp
       (String.explode string)
       (fn nil => true | _ => false)
end
```

Note that we incorporate the structure `RegExp` into the structure `Matcher`, in accordance with the requirements of the signature. The function `match` explodes the string into a list of characters (to facilitiate sequential processing of the input), then calls `match_is` with an initial continuation that ensures that the remaining input is empty to determine the result. The type of `match_is` is

```
RegExp.regexp -> char list ->
  (char list -> bool) -> bool.
```

That is, `match_is` takes in succession a regular expression, a list of characters, and a continuation of type `char list -> bool`; it yields as result a value of type `bool`. This is a fairly complicated type, but notice that nowhere did we have to write this type in the code! The type inference mechanism of ML took care of determining what that type must be based on an analysis of the code itself.

Since `match_is` takes a function as argument, it is said to be a *higher-order function*. The simplicity of the matcher is due in large measure to the ease with which we can manipulate functions in ML. Notice that we create a new, unnamed function to pass as a continuation in the case of concatenation — it is the function that matches the second part of the regular expression to the characters remaining after matching an initial segment against the first part. We use a similar technique to implement matching against an iterated regular expression — we attempt to match the null string, but if this fails, we match against the regular expression being iterated followed by the iteration once again. This neatly captures the "zero or more times" interpretation of iteration of a regular expression.

> *Important*: the code given above contains a subtle error. Can you find it? If not, see chapter 27 for further discussion!

This completes our brief overview of Standard ML. The remainder of these notes are structured into three parts. The first part is a detailed introduction to the core language, the language in which we write programs in ML. The second part is concerned with the module language, the means by which we structure large programs in ML. The third is about programming techniques, methods for building reliable and robust programs. I hope you enjoy it!

1.2 Sample Code

Here is the complete code for this chapter:

```
signature REGEXP = sig
```

```
  datatype regexp =
    Zero | One | Char of char |
    Plus of regexp * regexp | Times of regexp * regexp |
    Star of regexp

  exception SyntaxError of string
  val parse : string -> regexp

  val format : regexp -> string

end

signature MATCHER = sig

  structure RegExp : REGEXP

  val match : RegExp.regexp -> string -> bool

end

structure RegExp :> REGEXP = struct

  datatype token =
    AtSign | Percent | Literal of char | PlusSign | TimesSign |
    Asterisk | LParen | RParen

  exception LexicalError

  fun tokenize nil = nil
    | tokenize (#"+" :: cs) = (PlusSign :: tokenize cs)
    | tokenize (#"." :: cs) = (TimesSign :: tokenize cs)
    | tokenize (#"*" :: cs) = (Asterisk :: tokenize cs)
    | tokenize (#"(" :: cs) = (LParen :: tokenize cs)
    | tokenize (#")" :: cs) = (RParen :: tokenize cs)
    | tokenize (#"@" :: cs) = (AtSign :: tokenize cs)
    | tokenize (#"%" :: cs) = (Percent :: tokenize cs)
    | tokenize (#"\\" :: c :: cs) = Literal c :: tokenize cs
    | tokenize (#"\\" :: nil) = raise LexicalError
    | tokenize (#" " :: cs) = tokenize cs
    | tokenize (c :: cs) = Literal c :: tokenize cs

  datatype regexp =
    Zero | One | Char of char |
    Plus of regexp * regexp | Times of regexp * regexp |
    Star of regexp
```

```
exception SyntaxError of string

fun parse_exp ts =
    let
        val (r, ts') = parse_term ts
    in
        case ts'
          of (PlusSign::ts'') =>
             let
                 val (r', ts''') = parse_exp ts''
             in
                 (Plus (r, r'), ts''')
             end
           | _ => (r, ts')
    end

and parse_term ts =
    let
        val (r, ts') = parse_factor ts
    in
        case ts'
          of (TimesSign::ts'') =>
             let
                 val (r', ts''') = parse_term ts''
             in
                 (Times (r, r'), ts''')
             end
           | _ => (r, ts')
    end

and parse_factor ts =
    let
        val (r, ts') = parse_atom ts
    in
        case ts'
          of (Asterisk :: ts'') => (Star r, ts'')
           | _ => (r, ts')
    end

and parse_atom nil = raise SyntaxError ("Factor expected\n")
  | parse_atom (AtSign :: ts) = (Zero, ts)
  | parse_atom (Percent :: ts) = (One, ts)
  | parse_atom ((Literal c) :: ts) = (Char c, ts)
  | parse_atom (LParen :: ts) =
```

```
    let
        val (r, ts') = parse_exp ts
    in
        case ts'
          of nil => raise SyntaxError ("Right-parenthesis expected\n")
           | (RParen :: ts'') => (r, ts'')
           | _ => raise SyntaxError ("Right-parenthesis expected\n")
    end

fun parse s =
    let
        val (r, ts) = parse_exp (tokenize (String.explode s))
    in
        case ts
          of nil => r
           | _ => raise SyntaxError "Unexpected input.\n"
    end
    handle LexicalError => raise SyntaxError "Illegal input.\n"

fun format_exp Zero = [#"@"]
  | format_exp One = [#"%"]
  | format_exp (Char c) = [c]
  | format_exp (Plus (r1, r2)) =
    let
        val s1 = format_exp r1
        val s2 = format_exp r2
    in
        [#"("] @ s1 @ [#"+"] @ s2 @ [#")"]
    end
  | format_exp (Times (r1, r2)) =
    let
        val s1 = format_exp r1
        val s2 = format_exp r2
    in
        s1 @ [#"*"] @ s2
    end
  | format_exp (Star r) =
    let
        val s = format_exp r
    in
        [#"("] @ s @ [#")"] @ [#"*"]
    end

fun format r = String.implode (format_exp r)
```

```
end

structure Matcher :> MATCHER =
struct

    structure RegExp = RegExp

    open RegExp

    fun match_is Zero _ k = false
      | match_is One cs k = k cs
      | match_is (Char c) nil _ = false
      | match_is (Char c) (c'::cs) k = (c=c') andalso (k cs)
      | match_is (Plus (r1, r2)) cs k =
        (match_is r1 cs k) orelse (match_is r2 cs k)
      | match_is (Times (r1, r2)) cs k =
        match_is r1 cs (fn cs' => match_is r2 cs' k)
      | match_is (r as Star r1) cs k =
        (k cs) orelse match_is r1 cs (fn cs' => match_is r cs' k)

    fun match regexp string =
        match_is regexp (String.explode string)
        (fn nil => true | _ => false)

end
```

Part II

The Core Language

All Standard ML is divided into two parts. The first part, the *core language*, comprises the fundamental programming constructs of the language — the primitive types and operations, the means of defining and using functions, mechanisms for definining new types, and so on. The second part, the *module language*, comprises the mechanisms for structuring programs into separate units and is described in Part III. Here we introduce the core language.

Chapter 2

Types, Values, and Effects

2.1 Evaluation and Execution

Most familiar programming languages, such as C or Java, are based on an *imperative* model of computation. Programs are thought of as specifying a sequence of *commands* that modify the memory of the computer. Each step of execution examines the current contents of memory, performs a simple computation, modifies the memory, and continues with the next instruction. The individual commands are executed for their *effect* on the memory (which we may take to include both the internal memory and registers and the external input/output devices). The progress of the computation is controlled by evaluation of expressions, such as boolean tests or arithmetic operations, that are executed for their value. Conditional commands branch according to the value of some expression. Many languages maintain a distinction between expressions and commands, but often (in C, for example) expressions may also modify the memory, so that even expression evaluation has an effect.

Computation in ML is of a somewhat different nature. The emphasis in ML is on computation by *evaluation of expressions*, rather than *execution of commands*. The idea of computation is as a generalization of your experience from high school algebra in which you are given a polynomial in a variable x and are asked to calculate its value at a given value of x. We proceed by "plugging in" the given value for x, and then, using the rules of arithmetic, determine the value of the polynomial. The evaluation model of computation used in ML is based on the same idea, but rather than restrict ourselves to arithmetic operations on the reals, we admit a richer variety of values and a richer variety of primitive operations on them.

The evaluation model of computation enjoys several advantages over the more familiar imperative model. Because of its close relationship to mathematics, it is much easier to develop mathematical techniques for reasoning about the behavior of programs. These techniques are important tools for helping us to ensure that programs work properly without having to resort to tedious testing and debugging that can only show the presence of errors, never their absence. Moreover, they provide important tools for documenting the reasoning that went into the formulation of a program, making the code easier to understand and maintain.

What is more, the evaluation model subsumes the imperative model as a special case. Execution of commands for the effect on memory can be seen as a special case of evaluation of expressions by introducing primitive operations for allocating, accessing, and modifying memory. Rather than forcing all aspects of computation into the framework of memory modification, we instead take expression evaluation as the primary notion. Doing so allows us to support imperative programming without destroying the mathematical elegance of the evaluation model for programs that don't use memory. As we will see, it is quite remarkable how seldom memory modification is required. Nevertheless, the language provides for storage-based computation for those few times that it is actually necessary.

2.2 The ML Computation Model

Computation in ML consists of evaluation of expressions. Each expression has three important characteristics:

- It may or may not have a *type*.
- It may or may not have a *value*.
- It may or may not engender an *effect*.

These characteristics are all that you need to know to compute with an expression.

The type of an expression is a description of the value it yields, should it yield a value at all. For example, for an expression to have type `int` is to say that its value (should it have one) is a number, and for an expression to have type `real` is to say that its value (if any) is a floating point number. In general we can think of the type of an expression as a "prediction" of the form of the value that it has, should it have one. Every expression is required to have at least one type; those that do are said to be *well-typed*.

Those without a type are said to be *ill-typed*; they are considered ineligible for evaluation. The *type checker* determines whether or not an expression is well-typed, rejecting with an error those that are not.

A well-typed expression is evaluated to determine its value, if indeed it has one. An expression can fail to have a value because its evaluation never terminates or because it raises an exception, either because of a run-time fault such as division by zero or because some programmer-defined condition is signalled during its evaluation. If an expression has a value, the form of that value is predicted by its type. For example, if an expression evaluates to a value v and its type is `bool`, then v must be either `true` or `false`; it cannot be, say, `17` or `3.14`. The *soundness* of the type system ensures the accuracy of the predictions made by the type checker.

Evaluation of an expression might also engender an *effect*. Effects include such phenomena as raising an exception, modifying memory, performing input or output, or sending a message on the network. It is important to note that the type of an expression says nothing about its possible effects! An expression of type `int` might well display a message on the screen before returning an integer value. This possibility is not accounted for in the type of the expression, which classifies only its value. For this reason effects are sometimes called *side effects*, to stress that they happen "off to the side" during evaluation, and are not part of the value of the expression. We will ignore effects until chapter 13. For the time being we will assume that all expressions are *effect-free*, or *pure*.

2.2.1 Type Checking

What is a type? What types are there? Generally speaking, a type is defined by specifying three things:

- a *name* for the type,
- the *values* of the type, and
- the *operations* that may be performed on values of the type.

Often the division of labor into values and operations is not completely clear-cut, but it nevertheless serves as a very useful guideline for describing types.

Let's consider first the type of integers. Its name is `int`. The values of type `int` are the numerals `0`, `1`, `~1`, `2`, `~2`, and so on. (Note that negative numbers are written with a prefix tilde, rather than a minus sign!) Operations on integers include addition, `+`, subtraction, `-`, multiplication, `*`, quotient, `div`, and remainder, `mod`. Arithmetic expressions are formed in the familiar manner, for example, `3*2+6`, governed by the usual rules of precedence. Parentheses may be used to override the precedence conventions, just as in ordinary mathematical practice. Thus the preceding expression may be equivalently written as `(3*2)+6`, but we may also write `3*(2+6)` to override the default precedences.

The formation of expressions is governed by a set of *typing rules* that define the types of expressions in terms of the types of their constituent expressions (if any). The typing rules are generally quite intuitive since they are consistent with our experience in mathematics and in other programming languages. In their full generality the rules are somewhat involved, but we will sneak up on them by first considering only a small fragment of the language, building up additional machinery as we go along.

Here are some simple arithmetic expressions, written using *infix* notation for the operations (meaning that the operator comes between the arguments, as is customary in mathematics):

```
3
3 + 4
4 div 3
4 mod 3
```

Each of these expressions is well-formed; in fact, they each have type `int`. This is indicated by a *typing assertion* of the form *exp* : *typ*, which states that the expression *exp* has the type *typ*. A typing assertion is said to be *valid* iff the expression *exp* does indeed have the type *typ*. The following are all valid typing assertions:

```
3 : int
3 + 4 : int
4 div 3 : int
4 mod 3 : int
```

Why are these typing assertions valid? In the case of the value `3`, it is an axiom that integer numerals have integer type. What about the expression `3+4`? The addition operation takes two arguments, each of which must have type `int`. Since both arguments in fact have type `int`, it follows that the entire expression is of type `int`. For more complex cases we reason analogously, for example, deducing that `(3+4) div (2+3): int` by observing that `(3+4): int` and `(2+3): int`.

The reasoning involved in demonstrating the validity of a typing assertion may be summarized by a *typing derivation* consisting of a nested sequence of typing assertions, each justified either by an axiom, or a typing rule for an operation. For example, the validity of the typing assertion `(3+7) div 5` : `int` is justified by the following derivation:

1. `(3+7)`: `int`, because
   (a) `3` : `int` because it is an axiom
   (b) `7` : `int` because it is an axiom
   (c) the arguments of + must be integers, and the result of + is an integer
2. `5` : `int` because it is an axiom
3. the arguments of `div` must be integers, and the result is an integer

The outermost steps justify the assertion `(3+4) div 5` : `int` by demonstrating that the arguments each have type `int`. Recursively, the inner steps justify that `(3+4)`: `int`.

2.2.2 Evaluation

Evaluation of expressions is defined by a set of *evaluation rules* that determine how the value of a compound expression is determined as a function of the values of its constituent expressions (if any). Since the value of an operator is determined by the values of its arguments, ML is sometimes said to be a *call-by-value* language. While this may seem like the only sensible way to define evaluation, we will see in chapter 15 that this need not be the case — some operations may yield a value *without* evaluating their arguments. Such operations are sometimes said to be *lazy*, to distinguish them from *eager* operations that require their arguments to be evaluated before the operation is performed.

An *evaluation assertion* has the form $exp \Downarrow val$. This assertion states that the expression exp has value val. It should be intuitively clear that the following evaluation assertions are valid.

```
5 ⇓ 5
2+3 ⇓ 5
(2+3) div (1+4) ⇓ 1
```

An evaluation assertion may be justified by an *evaluation derivation*, which is similar to a typing derivation. For example, we may justify the assertion `(3+7) div 5` $\Downarrow$ `2` by the derivation

1. `(3+7)` $\Downarrow$ `10` because

   (a) `3` $\Downarrow$ `3` because it is an axiom

   (b) `2` $\Downarrow$ `2` because it is an axiom

   (c) Adding `3` to `7` yields `10`.

2. `5` $\Downarrow$ `5` because it is an axiom

3. Dividing `10` by `5` yields `2`.

Note that is an axiom that a numeral evaluates to itself; numerals are fully-evaluated expressions, or *values*. Second, the rules of arithmetic are used to determine that adding `3` and `7` yields `10`.

Not every expression has a value. A simple example is the expression `5 div 0`, which is undefined. If you attempt to evaluate this expression it will incur a run-time error, reflecting the erroneous attempt to find the number n that, when multiplied by 0, yields 5. The error is expressed in ML by raising an *exception*; we will have more to say about exceptions in chapter 12. Another reason that a well-typed expression might not have a value is that the attempt to evaluate it leads to an infinite loop. We don't yet have the machinery in place to define such expressions, but we will soon see that it is possible for an expression to *diverge*, or run forever, when evaluated.

2.3 Types, Types, Types

What types are there besides the integers? Here are a few useful *base* types of ML:

- *Type name*: `real`
  - *Values*: `3.14, 2.17,0.1E6,...`
  - *Operations*: `+`, `-`, `*`, `/`, `=`, `<`, ...
- *Type name*: `char`
  - *Values*: `#"a", #"b",...`
  - *Operations*: `ord`,`chr`,`=`, `<`, ...

- *Type name*: `string`
  - *Values*: `"abc", "1234",...`
  - *Operations*: `^`, `size`, `=`, `<`, ...
- *Type name*: `bool`
  - *Values*: `true`, `false`
  - *Operations*: `if` *exp* `then` exp_1 `else` exp_2

There are many, many (in fact, infinitely many!) others, but these are enough to get us started. (See V for a complete description of the primitive types of ML, including the ones given above.)

Notice that some of the arithmetic operations for real numbers are written the same way as for the corresponding operation on integers. For example, we may write `3.1+2.7` to perform a floating point addition of two floating point numbers. This is called *overloading*; the addition operation is said to be *overloaded* at the types `int` and `real`. In an expression involving addition the type checker tries to resolve which form of addition (fixed point or floating point) you mean. If the arguments are `int`'s, then fixed point addition is used; if the arguments are `real`'s, then floating addition is used; otherwise an error is reported.[1] Note that ML does *not* perform any implicit conversions between types! For example, the expression `3+3.14` is rejected as ill-formed! If you intend floating point addition, you must write instead `real(3)+3.14`, which converts the integer `3` to its floating point representation before performing the addition. If, on the other hand, you intend integer addition, you must write `3+round(3.14)`, which converts `3.14` to an integer by rounding before performing the addition.

Finally, note that floating point division is a *different* operation from integer quotient! Thus we write `3.1/2.7` for the result of dividing `3.1` by `2.7`, which results in a floating point number. We reserve the operator `div` for integers, and use `/` for floating point division.

The *conditional expression*

`if` *exp* `then` exp_1 `else` exp_2

is used to discriminate on a Boolean value. It has type *typ* if *exp* has type `bool` and both exp_1 and exp_2 have type *typ*. Notice that both "arms" of the conditional must have the same type! It is evaluated by first evaluating *exp*, then proceeding to evaluate either exp_1 or exp_2, according to whether the value of *exp* is `true` or `false`. For example,

[1]If the type of the arguments cannot be determined, the type defaults to `int`.

```
if 1<2 then "less" else "greater"
```

evaluates to `"less"` since the value of the expression `1<2` is true.

Note that the expression

```
if 1<2 then 0 else (1 div 0)
```

evaluates to `0`, even though `1 div 0` incurs a run-time error. This is because evaluation of the conditional proceeds *either* to the `then` clause *or* to the `else` clause, depending on the outcome of the boolean test. Whichever clause is evaluated, the other is simply discarded without further consideration.

Although we may, in fact, test equality of two boolean expressions, it is rarely useful to do so. Beginners often writen conditionals of the form

if *exp* = true then exp_1 else exp_2.

But this is equivalent to the simpler expression

if *exp* then exp_1 else exp_2.

Similarly, rather than write

if *exp* = false then exp_1 else exp_2,

it is better to write

if not *exp* then exp_1 else exp_2

or, better yet, just

if *exp* then exp_2 else exp_1.

2.4 Type Errors

Now that we have more than one type, we have enough rope to hang ourselves by forming *ill-typed* expressions. For example, the following expressions are not well-typed:

```
size 45
#"1" + 1
#"2" ^ "1"
3.14 + 2
```

In each case we are “misusing” an operator with arguments of the wrong type.

This raises a natural question: is the following expression well-typed or not?

```
if 1<2 then 0 else ("abc"+4)
```

Since the boolean test will come out `true`, the `else` clause will never be executed, and hence need not be constrained to be well-typed. While this reasoning is sensible for such a simple example, in general it is impossible for the type checker to determine the outcome of the boolean test during type checking. To be safe the type checker “assumes the worst” and insists that both clauses of the conditional be well-typed, and in fact have the *same* type, to ensure that the conditional expression can be given a type, namely that of both of its clauses.

Chapter 3

Declarations

3.1 Variables

Just as in any other programming language, values may be assigned to variables, which may then be used in expressions to stand for that value. However, in sharp contrast to most familiar languages, *variables in ML do not vary*! A value may be *bound* to a variable using a construct called a *value binding*. Once a variable is bound to a value, it is bound to it for life; there is no possibility of changing the binding of a variable once it has been bound. In this respect variables in ML are more akin to variables in mathematics than to variables in languages such as C.

A type may also be bound to a *type constructor* using a *type binding*. A bound type constructor stands for the type bound to it, and can never stand for any other type. For this reason a type binding is sometimes called a *type abbreviation* — the type constructor stands for the type to which it is bound.[1]

A value or type binding introduces a "new" variable or type constructor, distinct from all others of that class, for use within its range of significance, or *scope*. Scoping in ML is *static*, or *lexical*, meaning that the range of significance of a variable or type constructor is determined by the program text, not by the order of evaluation of its constituent expressions. (Languages with *dynamic* scope adopt the opposite convention.) For the time being variables and type constructors have *global scope*, meaning that the range of significance of the variable or type constructor is the "rest" of the program — the part that lexically follows the binding. However, we will soon introduce mechanisms for limiting the scopes of variables or type con-

[1] By the same token a value binding might also be called a *value abbreviation*, but for some reason it never is.

structors to a given expression.

3.2 Basic Bindings

3.2.1 Type Bindings

Any type may be given a name using a *type binding*. At this stage we have so few types that it is hard to justify binding type names to identifiers, but we'll do it anyway because we'll need it later. Here are some examples of type bindings:

```
type float = real
type count = int and average = real
```

The first type binding introduces the type constructor `float`, which subsequently is synonymous with `real`. The second introduces *two* type constructors, `count` and `average`, which stand for `int` and `real`, respectively.

In general a type binding introduces one or more new type constructors *simultaneously* in the sense that the definitions of the type constructors may not involve any of the type constructors being defined. Thus a binding such as

```
type float = real and average = float
```

is nonsensical (in isolation) since the type constructors `float` and `average` are introduced simultaneously, and hence cannot refer to one another.

The syntax for type bindings is

type $tycon_1$ = typ_1
and ...
and $tycon_n$ = typ_n

where each $tycon_i$ is a type constructor and each typ_i is a type expression.

3.2.2 Value Bindings

A value may be given a name using a *value binding*. Here are some examples:

```
val m : int = 3+2
val pi : real = 3.14 and e : real = 2.17
```

The first binding introduces the variable `m`, specifying its type to be `int` and its value to be `5`. The second introduces two variables, `pi` and `e`, simultaneously, both having type `real`, and with `pi` having value `3.14` and `e` having value `2.17`. Notice that a value binding specifies both the type and the value of a variable.

The syntax of value bindings is

```
val var_1 : typ_1 = exp_1
and ...
and var_n : typ_n = exp_n,
```

where each var_i is a variable, each typ_i is a type expression, and each exp_i is an expression.

A value binding of the form

```
val var : typ = exp
```

is type-checked by ensuring that the expression exp has type typ. If not, the binding is rejected as ill-formed. If so, the binding is evaluated using the *bind-by-value* rule: first exp is evaluated to obtain its value val, then val is bound to var. If exp does not have a value, then the declaration does not bind anything to the variable var.

The purpose of a binding is to make a variable available for use within its scope. In the case of a type binding we may use the type variable introduced by that binding in type expressions occurring within its scope. For example, in the presence of the type bindings above, we may write

```
val pi : float = 3.14
```

since the type constructor `float` is bound to the type `real`, the type of the expression `3.14`. Similarly, we may make use of the variable introduced by a value binding in value expressions occurring within its scope.

Continuing from the preceding binding, we may use the expression

```
sin pi
```

to stand for `0.0` (approximately), and we may bind this value to a variable by writing

```
val x : float = sin pi
```

As these examples illustrate, type checking and evaluation are *context dependent* in the presence of type and value bindings since we must refer to these bindings to determine the types and values of expressions. For example, to determine that the above binding for `x` is well-formed, we must consult the binding for `pi` to determine that it has type `float`, consult the binding for `float` to determine that it is synonymous with `real`, which is necessary for the binding of `x` to have type `float`.

The rough-and-ready rule for both type-checking and evaluation is that a bound variable or type constructor is implicitly *replaced* by its binding prior to type checking and evaluation. This is sometimes called the *substitution principle* for bindings. For example, to evaluate the expression `cos x` in the scope of the above declarations, we first replace the occurrence of `x` by its value (approximately `0.0`), then compute as before, yielding (approximately) `1.0`. Later on we will have to refine this simple principle to take account of more sophisticated language features, but it is useful nonetheless to keep this simple idea in mind.

3.3 Compound Declarations

Bindings may be combined to form *declarations*. A binding is an atomic declaration, even though it may introduce many variables simultaneously. Two declarations may be combined by *sequential composition* by simply writing them one after the other, optionally separated by a semicolon. Thus we may write the declaration

```
val m : int = 3+2
val n : int = m*m
```

which binds `m` to `5` and `n` to `25`. Subsequently, we may evaluate `m+n` to obtain the value `30`. In general a sequential composition of declarations has the form $dec_1 \ldots dec_n$, where n is at least 2. The scopes of these declarations are *nested* within one another: the scope of dec_1 includes $dec_2, \ldots, dec_n$, the scope of dec_2 includes $dec_3, \ldots, dec_n$, and so on.

One thing to keep in mind is that *binding is not assignment*. The binding of a variable never changes; once bound to a value, it is always bound to that value (within the scope of the binding). However, we may *shadow* a binding by introducing a second binding for a variable within the scope of the first binding. Continuing the above example, we may write

```
val n : real = 2.17
```

to introduce a new variable `n` with both a different type and a different value than the earlier binding. The new binding eclipses the old one, which may then be discarded since it is no longer accessible. (Later on, we will see that in the presence of higher-order functions shadowed bindings are not always discarded, but are preserved as private data in a closure. One might say that old bindings never die, they just fade away.)

3.4 Limiting Scope

The scope of a variable or type constructor may be delimited by using `let` expressions and `local` declarations. A `let` expression has the form

let *dec* in *exp* end

where *dec* is any declaration and *exp* is any expression. The scope of the declaration *dec* is limited to the expression *exp*. The bindings introduced by *dec* are discarded upon completion of evaluation of *exp*.

Similarly, we may limit the scope of one declaration to another declaration by writing

local *dec* in *dec′* end.

The scope of the bindings in *dec* is limited to the declaration *dec′*. After processing *dec′*, the bindings in *dec* may be discarded.

The value of a `let` expression is determined by evaluating the declaration part, then evaluating the expression relative to the bindings introduced by the declaration, yielding this value as the overall value of the `let` expression. An example will help clarify the idea:

```
let
    val m : int = 3
    val n : int = m*m
in
    m*n
end
```

This expression has type `int` and value `27`, as you can readily verify by first calculating the bindings for `m` and `n`, then computing the value of `m*n` relative to these bindings. The bindings for `m` and `n` are local to the expression `m*n`, and are not accessible from outside the expression.

If the declaration part of a `let` expression eclipses earlier bindings, the ambient bindings are restored upon completion of evaluation of the `let` expression. Thus the following expression evaluates to `54`:

```
val m : int = 2
val r : int =
    let
        val m : int = 3
        val n : int = m*m
    in
        m*n
    end * m
```

The binding of `m` is temporarily overridden during the evaluation of the `let` expression, then restored upon completion of this evaluation.

3.5 Typing and Evaluation

To complete this chapter, let's consider in more detail the context-sensitivity of type checking and evaluation in the presence of bindings. The key ideas are:

- Type checking must take account of the declared type of a variable.
- Evaluation must take account of the declared value of a variable.

This is achieved by maintaining *environments* for type checking and evaluation. The *type environment* records the types of variables; the *value environment* records their values. For example, after processing the compound declaration

```
val m : int = 0
val x : real = Math.sqrt(2.0)
val c : char = #"a"
```

the type environment contains the information

```
val m : int
val x : real
val c : char
```

and the value environment contains the information

```
val m = 0
val x = 1.414
val c = #"a"
```

In a sense the value declarations have been divided in "half", separating the type from the value information.

Thus we see that value bindings have significance for both type checking and evaluation. In contrast type bindings have significance only for type checking, and hence contribute only to the type environment. A type binding such as

```
type float = real
```

is recorded in its entirety in the type environment, and no change is made to the value environment. Subsequently, whenever we encounter the type constructor `float` in a type expression, it is replaced by `real` in accordance with the type binding above.

In chapter 2 we said that a typing assertion has the form $exp : typ$, and that an evaluation assertion has the form $exp \Downarrow val$. While two-place typing and evaluation assertions are sufficient for *closed* expressions (those without variables), we must extend these relations to account for *open* expressions (those with variables). Each must be equipped with an *environment* recording information about type constructors and variables introduced by declarations.

Typing assertions are generalized to have the form

$$typenv \ \vdash \ exp \ : \ typ$$

where $typenv$ is a *type environment* that records the bindings of type constructors and the types of variables that may occur in exp.[2] We may think of $typenv$ as a sequence of specifications of one of the following two forms:

1. `type` $typvar$ `=` typ
2. `val` var `:` typ

Note that the second form does *not* include the binding for var, only its type!

Evaluation assertions are generalized to have the form

$$valenv \ \vdash \ exp \ \Downarrow \ val$$

where $valenv$ is a *value environment* that records the bindings of the variables that may occur in exp. We may think of $valenv$ as a sequence of specifications of the form

[2] The *turnstile* symbol, "$\vdash$", is simply a punctuation mark separating the type environment from the expression and its type.

$$\texttt{val}\ \ \mathit{var}\ \ \texttt{=}\ \ \mathit{val}$$

that bind the value val to the variable var.

Finally, we also need a new assertion, called *type equivalence*, that determines when two types are equivalent, relative to a type environment. This is written

$$\mathit{typenv}\ \vdash\ \mathit{typ}_1\ \equiv\ \mathit{typ}_2$$

Two types are equivalent iff they are the same when the type constructors defined in typenv are replaced by their bindings.

The primary use of a type environment is to record the types of the value variables that are available for use in a given expression. This is expressed by the following axiom:

$$\ldots\texttt{val}\ \ \mathit{var}\ \ \texttt{:}\ \ \mathit{typ}\ \ldots \vdash \mathit{var} : \mathit{typ}$$

In words, if the specification `val` var `:` typ occurs in the type environment, then we may conclude that the variable var has type typ. This rule glosses over an important point. In order to account for shadowing we require that the *rightmost* specification govern the type of a variable. That way re-binding of variables at different types behaves as expected.

Similarly, the evaluation relation must take account of the value environment. Evaluation of variables is governed by the following axiom:

$$\ldots\texttt{val}\ \ \mathit{var}\ \ \texttt{=}\ \ \mathit{val}\ \ldots \vdash \mathit{var} \Downarrow \mathit{val}$$

Here again we assume that the `val` specification is the rightmost one governing the variable var to ensure that the scoping rules are respected.

The role of the type equivalence assertion is to ensure that type constructors always stand for their bindings. This is expressed by the following axiom:

$$\ldots\texttt{type}\ \ \mathit{typvar}\ \ \texttt{=}\ \ \mathit{typ}\ \ldots \vdash \mathit{typvar} \equiv \mathit{typ}$$

Once again, the rightmost specification for typvar governs the assertion.

Chapter 4

Functions

4.1 Functions as Templates

So far we just have the means to calculate the values of expressions, and to bind these values to variables for future reference. In this chapter we will introduce the ability to *abstract* the data from a calculation, leaving behind the bare *pattern* of the calculation. This pattern may then be *instantiated* as often as you like so that the calculation may be repeated with specified data values plugged in.

For example, consider the expression `2*(3+4)`. The data might be taken to be the values `2`, `3`, and `4`, leaving behind the pattern `□ * (□ + □)`, with "holes" where the data used to be. We might equally well take the data to just be `2` and `3`, and leave behind the pattern `□ * (□ + 4)`. Or we might even regard `*` and `+` as the data, leaving `2 □ (3 □ 4)` as the pattern! What is important is that a complete expression can be recovered by filling in the holes with chosen data.

Since a pattern can contain many different holes that can be independently instantiated, it is necessary to give *names* to the holes so that instantiation consists of plugging in a given value for all occurrences of a name in an expression. These names are, of course, just variables, and instantiation is just the process of substituting a value for all occurrences of a variable in a given expression. A pattern may therefore be viewed as a *function* of the variables that occur within it; the pattern is instantiated by *applying* the function to argument values.

This view of functions is similar to our experience from high school algebra. In algebra we manipulate polynomials such as x^2+2x+1 as a form of expression denoting a real number, with the variable x representing a fixed, but unknown, quantity. (Indeed, variables in algebra are sometimes called *unknowns*, or *indeterminates*.) It is also possible to think of a polynomial as a function on the real line: given a real number x, a polynomial determines a real number y computed by the given combination of arithmetic operations. Indeed, we sometimes write equations such as $f(x) = x^2 + 2x + 1$, to stand for the function f determined by the polynomial. In the univariate case we can get away with just writing the polynomial for the function, but in the multivariate case we must be more careful: we may regard the polynomial $x^2 + 2xy + y^2$ to be a function of x, a function of y, or a function of both x and y. In these cases we write $f(x) = x^2 + 2xy + y^2$ when x varies and y is held fixed, and $g(y) = x^2 + 2xy + y^2$ when y varies for fixed x, and $h(x, y) = x^2 + 2xy + y^2$, when both vary jointly.

In algebra it is usually left implicit that the variables x and y range over the real numbers, and that f, g, and h are functions on the real line. However, to be fully explicit, we sometimes write something like

$$f : \mathbb{R} \to \mathbb{R} \; : \; x \in R \mapsto x^2 + 2x + 1$$

to indicate that f is a function on the reals sending $x \in \mathbb{R}$ to $x^2 + 2x + 1 \in \mathbb{R}$. This notation has the virtue of separating the *name* of the function, f, from the function itself, the mapping that sends $x \in \mathbb{R}$ to $x^2 + 2x + 1$. It also emphasizes that functions are a kind of "value" in mathematics (namely, a certain set of ordered pairs), and that the variable f is bound to that value (*i.e.*, that set) by the declaration. This viewpoint is especially important once we consider operators, such as the differential operator, that map functions to functions. For example, if f is a differentiable function on the real line, the function $D\,f$ is its first derivative, another function on the real line. In the case of the function f defined above the function $D\,f$ sends $x \in \mathbb{R}$ to $2x + 2$.

4.2 Functions and Application

The treatment of functions in ML is very similar, except that we stress both the *algorithmic* aspects of functions (*how* they determine values from arguments), as well as the *extensional* aspects (*what* they compute). As in mathematics, a function in ML is a kind of value, namely a value of *function type* of the form *typ* `->` *typ*′. The type *typ* is the *domain type* (the type of arguments) of the function, and *typ*′ is its *range type* (the type of its results). We compute with a function by *applying* it to an *argument* value of its domain type and calculating the *result*, a value of its range type. Function application is indicated by juxtaposition: we simply write the argument next to the function.

The values of function type consist of *primitive functions*, such as addition and square root, and *function expressions*, which are also called *lambda expressions*,[1] of the form

```
fn var : typ => exp
```

The variable *var* is called the *parameter*, and the expression *exp* is called the *body*. It has type *typ*->*typ*′ provided that *exp* has type *typ*′ under the assumption that the parameter *var* has the type *typ*.

To apply such a function expression to an argument value *val*, we add the binding

```
val var = val
```

to the value environment, and evaluate *exp*, obtaining a value *val*′. Then the value binding for the parameter is removed, and the result value, *val*′, is returned as the value of the application.

For example, `Math.sqrt` is a primitive function of type `real->real` that may be applied to a real number to obtain its square root. For example, the expression `Math.sqrt 2.0` evaluates to `1.414` (approximately). We can, if we wish, parenthesize the argument, writing `Math.sqrt (2.0)` for the sake of clarity; this is especially useful for expressions such as `Math.sqrt (Math.sqrt 2.0)`. The square root function is built in. We may write the fourth root function as the following function expression:

```
fn x : real => Math.sqrt (Math.sqrt x)
```

It may be applied to an argument by writing an expression such as

```
(fn x : real => Math.sqrt (Math.sqrt x)) (16.0),
```

which calculates the fourth root of `16.0`. The calculation proceeds by binding the variable `x` to the argument `16.0`, then evaluating the expression `Math.sqrt (Math.sqrt x)` in the presence of this binding. When evaluation completes, we drop the binding of `x` from the environment, since it is no longer needed.

Notice that we did not give the fourth root function a name; it is an "anonymous" function. We may give it a name using the declaration forms introduced in chapter 3. For example, we may bind the fourth root function to the variable `fourthroot` using the following declaration:

[1] For purely historical reasons.

```
val fourthroot : real -> real =
    fn x : real => Math.sqrt (Math.sqrt x)
```

We may then write `fourthroot 16.0` to compute the fourth root of `16.0`.

This notation for defining functions quickly becomes tiresome, so ML provides a special syntax for function bindings that is more concise and natural. Instead of using the `val` binding above to define `fourthroot`, we may instead write

```
fun fourthroot (x:real):real = Math.sqrt (Math.sqrt x)
```

This declaration has the same meaning as the earlier `val` binding, namely it binds `fn x:real => Math.sqrt(Math.sqrt x)` to the variable `fourthroot`.

It is important to note that function applications in ML are evaluated according to the *call-by-value* rule: the arguments to a function are evaluated before the function is called. Put in other terms, functions are defined to act on values, rather than on unevaluated expressions. Thus, to evaluate an expression such as `fourthroot (2.0+2.0)`, we proceed as follows:

1. Evaluate `fourthroot` to the function value `fn x : real => Math.sqrt (Math.sqrt x)`.
2. Evaluate the argument `2.0+2.0` to its value `4.0`
3. Bind `x` to the value `4.0`.
4. Evaluate `Math.sqrt (Math.sqrt x)` to `1.414` (approximately).
   (a) Evaluate `Math.sqrt` to a function value (the primitive square root function).
   (b) Evaluate the argument expression `Math.sqrt x` to its value, approximately `1.414`.
      i. Evaluate `Math.sqrt` to a function value (the primitive square root function).
      ii. Evaluate `x` to its value, `2.0`.
      iii. Compute the square root of `2.0`, yielding `1.414`.
   (c) Compute the square root of `1.414`, yielding `1.189`.
5. Drop the binding for the variable `x`.

Notice that we evaluate *both* the function and argument positions of an application expression — both the function and argument are expressions yielding values of the appropriate type. The value of the function position must be a value of function type, either a primitive function or a lambda expression, and the value of the argument position must be a value of the domain type of the function. In this case the result value (if any) will be of the range type of the function. Functions in ML are *first-class*, meaning that they may be computed as the value of an expression. We are not limited to applying only named functions, but rather may compute "new" functions on the fly and apply these to arguments. This is a source of considerable expressive power, as we shall see in the sequel.

Using similar techniques we may define functions with arbitrary domain and range. For example, the following are all valid function declarations:

```
fun srev (s:string):string = implode (rev (explode s))
fun pal (s:string):string = s ^ (srev s)
fun double (n:int):int = n + n
fun square (n:int):int = n * n
fun halve (n:int):int = n div 2
fun is_even (n:int):bool = (n mod 2 = 0)
```

Thus `pal "ot"` evaluates to the string `"otto"`, and `is_even 4` evaluates to `true`.

4.3 Binding and Scope, Revisited

A function expression of the form

`fn` *var*`:`*typ* `=>` *exp*

binds the variable *var* within the body *exp* of the function. Unlike `val` bindings, function expressions bind a variable without giving it a specific value. The value of the parameter is only determined when the function is applied, and then only temporarily, for the duration of the evaluation of its body.

It is worth reviewing the rules for binding and scope of variables that we introduced in chapter 3 in the presence of function expressions. As before we adhere to the principle of *static scope*, according to which variables are taken to refer to the *nearest enclosing binding* of that variable, whether by a `val` binding or by a `fn` expression.

Thus, in the following example, the occurrences of `x` in the body of the function `f` refer to the parameter of `f`, whereas the occurrences of `x` in the body of `g` refer to the preceding `val` binding.

```
val x:real = 2.0
fun f(x:real):real = x+x
fun g(y:real):real = x+y
```

Local `val` bindings may shadow parameters, as well as other `val` bindings. For example, consider the following function declaration:

```
fun h(x:real):real =
    let val x:real = 2.0 in x+x end * x
```

The inner binding of `x` by the `val` declaration shadows the parameter `x` of `h`, but *only* within the body of the `let` expression. Thus the last occurrence of `x` refers to the parameter of `h`, whereas the preceding two occurrences refer to the inner binding of `x` to `2.0`.

The phrases "inner" and "outer" binding refer to the *logical structure*, or *abstract syntax* of an expression. In the preceding example, the body of `h` lies "within" the scope of the parameter `x`, and the expression `x+x` lies within the scope of the `val` binding for `x`. Since the occurrences of `x` within the body of the `let` lie within the scope of the inner `val` binding, they are taken to refer to that binding, rather than to the parameter. On the other hand the last occurrence of `x` does not lie within the scope of the `val` binding, and hence refers to the paramter of `h`.

In general the names of parameters do not matter; we can rename them at will without affecting the meaning of the program, provided that we simultaneously (and consistently) rename the binding occurrence and all uses of that variable. Thus the functions `f` and `g` below are completely equivalent to each other:

```
fun f(x:int):int = x*x
fun g(y:int):int = y*y
```

A parameter is just a placeholder; its name is not important.

Our ability to rename parameters is constrained by the static scoping rule. We may rename a parameter to whatever we'd like, provided that we don't change the way in which uses of a variable are resolved. For example, consider the following situation:

```
val x:real = 2.0
fun h(y:real):real = x+y
```

The parameter `y` to `h` may be renamed to `z` without affecting its meaning. However, we may *not* rename it to `x`, for doing so changes its meaning! That is, the function

```
fun h'(x:real):real = x+x
```

does *not* have the same meaning as `h`, because now both occurrences of `x` in the body of `h'` refer to the parameter, whereas in `h` the variable `x` refers to the outer `val` binding, whereas the variable `y` refers to the parameter.

While this may seem like a minor technical issue, it is essential that you master these concepts now, for they play a central, and rather subtle, role later on.

Chapter 5

Products and Records

5.1 Product Types

A distinguishing feature of ML is that aggregate data structures, such as tuples, lists, arrays, or trees, may be created and manipulated with ease. In contrast to most familiar languages it is not necessary in ML to be concerned with allocation and deallocation of data structures, nor with any particular representation strategy involving, say, pointers or address arithmetic. Instead we may think of data structures as first-class values, on a par with every other value in the language. Just as it is unnecessary to think about "allocating" integers to evaluate an arithmetic expression, it is unnecessary to think about allocating more complex data structures such as tuples or lists.

5.1.1 Tuples

This chapter is concerned with the simplest form of aggregate data structure, the *n-tuple*. An n-tuple is a finite ordered sequence of values of the form

$$(\mathit{val}_1, \ldots, \mathit{val}_n),$$

where each val_i is a value. A 2-tuple is usually called a *pair*, a 3-tuple a *triple*, and so on.

An n-tuple is a value of a *product type* of the form

$$\mathit{typ}_1 * \ldots * \mathit{typ}_n.$$

Values of this type are n-tuples of the form

(val_1, ..., val_n),

where val_i is a value of type typ_i (for each $1 \le i \le n$).

Thus the following are well-formed bindings:

```
val pair : int * int = (2, 3)
val triple : int * real * string = (2, 2.0, "2")
val quadruple
    : int * int * real * real
    = (2,3,2.0,3.0)
val pair_of_pairs
    : (int * int) * (real * real)
    = ((2,3),(2.0,3.0))
```

The nesting of parentheses matters! A pair of pairs is not the same as a quadruple, so the last two bindings are of distinct values with distinct types.

There are two limiting cases, $n = 0$ and $n = 1$, that deserve special attention. A 0-tuple, which is also known as a *null tuple*, is the empty sequence of values, `()`. It is a value of type `unit`, which may be thought of as the 0-tuple type.[1] The null type is surprisingly useful, especially when programming with effects. On the other hand there seems to be no particular use for 1-tuples, and so they are absent from the language.

As a convenience, ML also provides a general *tuple expression* of the form

(exp_1, ..., exp_n),

where each exp_i is an arbitrary expression, not necessarily a value. Tuple expressions are evaluated from left to right, so that the above tuple expression evaluates to the tuple value yielding the tuple value

(val_1, ..., val_n),

provided that exp_1 evaluates to val_1, exp_2 evaluates to val_2, and so on. For example, the binding

```
val pair : int * int = (1+1, 5-2)
```

[1] In Java (and other languages) the type `unit` is misleadingly written `void`. The name "void" suggests that the type has *no* members, but in fact it has exactly one member, the null tuple!

binds the value (2, 3) to the variable pair.

Strictly speaking, it is not essential to have tuple expressions as a primitive notion in the language. Rather than write

$$(exp_1, \ldots, exp_n),$$

with the (implicit) understanding that the exp_i's are evaluated from left to right, we may instead write

```
let val x1 = exp1
    val x2 = exp2
    ⋮
    val xn = expn
in  (x1,...,xn) end
```

which makes the evaluation order explicit.

5.1.2 Tuple Patterns

One of the most powerful, and distinctive, features of ML is the use of *pattern matching* to access components of aggregate data structures. For example, suppose that *val* is a value of type

```
(int * string) * (real * char)
```

and we wish to retrieve the first component of the second component of *val*, a value of type real. Rather than explicitly "navigate" to this position to retrieve it, we may simply use a generalized form of value binding in which we select that component using a pattern:

```
val ((_, _), (r:real, _)) = val
```

The left-hand side of the val binding is a tuple pattern that describes a pair of pairs, binding the first component of the second component to the variable r. The underscores indicate "don't care" positions in the pattern — their values are not bound to any variable. If we wish to give names to all of the components, we may use the following value binding:

```
val ((i:int, s:string), (r:real, c:char)) = val
```

If we'd like we can even give names to the first and second components of the pair, without decomposing them into constituent parts:

```
val (is:int*string,rc:real*char) = val
```

The general form of a value binding is

val pat = exp,

where pat is a *pattern* and exp is an expression. A pattern is one of three forms:

1. A *variable pattern* of the form var:typ.
2. A *tuple pattern* of the form (pat_1,...,pat_n), where each pat_i is a pattern. This includes as a special case the null-tuple pattern, ().
3. A *wildcard pattern* of the form _.

The type of a pattern is determined by an inductive analysis of the form of the pattern:

1. A variable pattern var:typ is of type typ.
2. A tuple pattern (pat_1,...,pat_n) has type typ_1*⋯ *typ_n, where each pat_i is a pattern of type typ_i. The null-tuple pattern () has type unit.
3. The wildcard pattern _ has any type whatsoever.

A value binding of the form

val pat = exp

is well-typed iff pat and exp have the same type; otherwise the binding is ill-typed and is rejected.

For example, the following bindings are well-typed:

```
val (m:int, n:int) = (7+1,4 div 2)
val (m:int, r:real, s:string) = (7, 7.0, "7")
val ((m:int,n:int), (r:real, s:real)) = ((4,5),(3.1,2.7))
val (m:int, n:int, r:real, s:real) = (4,5,3.1,2.7)
```

In contrast, the following are ill-typed:

```
val (m:int,n:int,r:real,s:real) = ((4,5),(3.1,2.7))
val (m:int, r:real) = (7+1,4 div 2)
val (m:int, r:real) = (7, 7.0, "7")
```

Value bindings are evaluated using the *bind-by-value* principle discussed earlier, except that the binding process is now more complex than before. First, we evaluate the right-hand side of the binding to a value (if indeed it has one). This happens regardless of the form of the pattern — the right-hand side is *always* evaluated. Second, we perform *pattern matching* to determine the bindings for the variables in the pattern.

The process of matching a value against a pattern is defined by a set of rules for reducing bindings with complex patterns to a set of bindings with simpler patterns, stopping once we reach a binding with a variable pattern. The rules are as follows:

1. The variable binding `val` *var* `=` *val* is irreducible.

2. The wildcard binding `val _ =` *val* is discarded.

3. The tuple binding

   `val (`pat_1`, ... ,`pat_n`) =`
   `(`val_1`, ... ,`val_n`)`

   is reduced to the set of n bindings

   `val` pat_1 `=` val_1

   ⋮

   `val` pat_n `=` val_n

   In the case that $n = 0$ the tuple binding is simply discarded.

These simplifications are repeated until all bindings are irreducible, which leaves us with a set of variable bindings that constitute the result of pattern matching.

For example, evaluation of the binding

```
val ((m:int,n:int), (r:real, s:real)) = ((2,3),(2.0,3.0))
```

proceeds as follows. First, we compose this binding into the following two bindings:

```
val (m:int, n:int) = (2,3)
val (r:real, s:real) = (2.0,3.0).
```

Then we decompose each of these bindings in turn, resulting in the following set of four atomic bindings:

```
val m:int = 2
val n:int = 3
val r:real = 2.0
val s:real = 3.0
```

At this point the pattern-matching process is complete.

5.2 Record Types

Tuples are most useful when the number of positions is small. When the number of components grows beyond a small number, it becomes difficult to remember which position plays which role. In that case it is more natural to attach a *label* to each component of the tuple that mediates access to it. This is the notion of a *record type*.

A record type has the form

$\{lab_1 \texttt{:} typ_1 \texttt{,} \ldots \texttt{,} lab_n \texttt{:} typ_n\}$,

where $n \geq 0$, and all of the labels lab_i are distinct. A *record value* has the form

$\{lab_1 \texttt{=} val_1 \texttt{,} \ldots \texttt{,} lab_n \texttt{=} val_n\}$,

where val_i has type typ_i. A *record pattern* has the form

$\{lab_1 \texttt{=} pat_1 \texttt{,} \ldots \texttt{,} lab_n \texttt{=} pat_n\}$

which has type

$\{lab_1 \texttt{:} typ_1 \texttt{,} \ldots \texttt{,} lab_n \texttt{:} typ_n\}$

provided that each pat_i has type typ_i.

A record value binding of the form

$\texttt{val}$
$\quad \{lab_1 \texttt{=} pat_1 \texttt{,} \ldots \texttt{,} lab_n \texttt{=} pat_n\} \texttt{ = }$
$\quad \{lab_1 \texttt{=} val_1 \texttt{,} \ldots \texttt{,} lab_n \texttt{=} val_n\}$

is decomposed into the following set of bindings

$\texttt{val } pat_1 \texttt{ = } val_1$

$\vdots$

$\texttt{val } pat_n \texttt{ = } val_n \texttt{.}$

Since the components of a record are identified by name, not position, the order in which they occur in a record value or record pattern is not important. However, in a record *expression* (in which the components may not be fully evaluated), the fields are evaluated from left to right in the order written, just as for tuple expressions.

Here are some examples to help clarify the use of record types. First, let us define the record type `hyperlink` as follows:

```
type hyperlink =
     { protocol : string,
        address : string,
        display : string }
```

The record binding

```
val mailto_rwh : hyperlink =
    { protocol="mailto",
       address="rwh@cs.cmu.edu",
       display="Robert Harper" }
```

defines a variable of type `hyperlink`. The record binding

```
val { protocol=prot, display=disp, address=addr } = mailto_rwh
```

decomposes into the three variable bindings

```
val prot = "mailto"
val addr = "rwh@cs.cmu.edu"
val disp = "Robert Harper"
```

which extract the values of the fields of `mailto_rwh`.

Using wild cards we can extract selected fields from a record. For example, we may write

```
val {protocol=prot, address=_, display=_} = mailto_rwh
```

to bind the variable `prot` to the `protocol` field of the record value `mailto_rwh`.

It is quite common to encounter record types with tens of fields. In such cases even the wild card notation doesn't help much when it comes to selecting one or two fields from such a record. For this we often use *ellipsis patterns* in records, as illustrated by the following example.

```
val {protocol=prot,...} = intro_home
```

The pattern `{protocol=prot,...}` stands for the expanded pattern

```
{protocol=prot, address=_, display=_}
```

in which the ellided fields are implicitly bound to wildcard patterns.

In general the ellipsis is replaced by as many wildcard bindings as are necessary to fill out the pattern to be consistent with its type. In order for this to occur *the compiler must be able to determine unambiguously the type of the record pattern*. Here the right-hand side of the value binding determines the type of the pattern, which then determines which additional fields to fill in. In some situations the context does not disambiguate, in which case you must supply additional type information, or avoid the use of ellipsis notation.

Finally, ML provides a convenient abbreviated form of record pattern

$\{lab_1, \ldots, lab_n\}$

which stands for the pattern

$\{lab_1=var_1, \ldots, lab_n=var_n\}$

where the variables var_i are variables with the same name as the corresponding label lab_i. For example, the binding

```
val { protocol, address, display } = mailto_rwh
```

decomposes into the sequence of atomic bindings

```
val protocol = "mailto"
val address = "rwh@cs.cmu.edu"
val display = "Robert Harper"
```

This avoids the need to think up a variable name for each field; we can just make the label do "double duty" as a variable.

5.3 Multiple Arguments and Multiple Results

A function may bind more than one argument by using a pattern, rather than a variable, in the argument position. Function expressions are generalized to have the form

fn *pat* => *exp*

where *pat* is a pattern and *exp* is an expression. Application of such a function proceeds much as before, except that the argument value is matched against the parameter pattern to determine the bindings of zero or more variables, which are then used during the evaluation of the body of the function.

For example, we may make the following definition of the Euclidean distance function:

```
val dist
    : real * real -> real
    = fn (x:real, y:real) => sqrt (x*x + y*y)
```

This function may then be applied to a pair (a two-tuple!) of arguments to yield the distance between them. For example, `dist (2.0,3.0)` evaluates to (approximately) `4.0`.

Using `fun` notation, the distance function may be defined more concisely as follows:

```
fun dist (x:real, y:real):real = sqrt (x*x + y*y)
```

The meaning is the same as the more verbose `val` binding given earlier.

Keyword parameter passing is supported through the use of record patterns. For example, we may define the distance function using keyword parameters as follows:

```
fun dist' {x=x:real, y=y:real} = sqrt (x*x + y*y)
```

The expression `dist' {x=2.0,y=3.0}` invokes this function with the indicated x and y values.

Functions with multiple results may be thought of as functions yielding tuples (or records). For example, we might compute two different notions of distance between two points at once as follows:

```
fun dist2 (x:real, y:real):real*real
    = (sqrt (x*x+y*y), abs(x-y))
```

Notice that the result type is a pair, which may be thought of as two results.

These examples illustrate a pleasing regularity in the design of ML. Rather than introduce *ad hoc* notions such as multiple arguments, multiple results, or keyword parameters, we make use of the general mechanisms of tuples, records, and pattern matching.

It is sometimes useful to have a function to select a particular component from a tuple or record (*e.g.*, the third component or the component with a given labeled). Such functions may be easily defined using pattern matching. But since they arise so frequently, they are pre-defined in ML using *sharp notation*. For any tuple type

$$typ_1 \texttt{*} \cdots \texttt{*} typ_n \texttt{,}$$

and each $1 \leq i \leq n$, there is a function `#`i of type

$$typ_1 \texttt{*} \cdots \texttt{*} typ_n \texttt{->} typ_i$$

defined as follows:

```
fun #i (_, ..., _, x, _, ..., _) = x
```

where `x` occurs in the ith position of the tuple (and there are underscores in the other $n - 1$ positions).

Thus we may refer to the second field of a three-tuple val by writing `#2(`val`)`. It is bad style, however, to over-use the sharp notation; code is generally clearer and easier to maintain if you use patterns wherever possible. Compare, for example, the following definition of the Euclidean distance function written using sharp notation with the original.

```
fun dist (p:real*real):real
    = sqrt((#1 p)*(#1 p)+(#2 p)*(#2 p))
```

You can easily see that this gets out of hand very quickly, leading to unreadable code. *Use of the sharp notation is strongly discouraged!*

A similar notation is provided for record field selection. The following function `#lab` selects the component of a record with label lab.

```
fun #lab {lab=x,...} = x
```

Notice the use of ellipsis! Bear in mind the disambiguation requirement: any use of `#lab` must be in a context sufficient to determine the full record type of its argument.

Chapter 6

Case Analysis

6.1 Homegeneous and Heterogeneous Types

Tuple types have the property that all values of that type have the same form (n-tuples, for some n determined by the type); they are said to be *homogeneous*. For example, all values of type `int*real` are pairs whose first component is an integer and whose second component is a real. Any type-correct pattern will match any value of that type; there is no possibility of failure of pattern matching. The pattern `(x:int,y:real)` is of type `int*real` and hence will match any value of that type. On the other hand the pattern `(x:int,y:real,z:string)` is of type `int*real*string` and cannot be used to match against values of type `int*real`; attempting to do so *fails at compile time*.

Other types have values of more than one form; they are said to be *heterogeneous* types. For example, a value of type `int` might be `0`, `1`, `~1`, ... or a value of type `char` might be `#"a"` or `#"z"`. (Other examples of heterogeneous types will arise later on.) Corresponding to each of the values of these types is a pattern that matches only that value. Attempting to match any other value against that pattern *fails at execution time* with an error condition called a *bind failure*.

Here are some examples of pattern-matching against values of a heterogeneous type:

```
val 0 = 1-1
val (0,x) = (1-1, 34)
val (0, #"0") = (2-1, #"0")
```

The first two bindings succeed, the third fails. In the case of the second, the variable `x` is bound to `34` after the match. No variables are bound in the first or third examples.

6.2 Clausal Function Expressions

The importance of constant patterns becomes clearer once we consider how to define functions over heterogeneous types. This is achieved in ML using a *clausal function expression* whose general form is

$$\begin{array}{ll} \texttt{fn} & \mathit{pat}_1 \ \texttt{=>} \ \mathit{exp}_1 \\ \ \ | & \vdots \\ \ \ | & \mathit{pat}_n \ \texttt{=>} \ \mathit{exp}_n \end{array}$$

Each pat_i is a pattern and each exp_i is an expression involving the variables of the pattern pat_i. Each component pat`=>`exp is called a *clause*, or a *rule*. The entire assembly of rules is called a *match*.

The typing rules for matches ensure consistency of the clauses. Specifically, there must exist types typ_1 and typ_2 such that

1. Each pattern pat_i has type typ_1.
2. Each expression exp_i has type typ_2, given the types of the variables in pattern pat_i.

If these requirements are satisfied, the function has the type typ_1`->`typ_2.

Application of a clausal function to a value val proceeds by considering the clauses *in the order written*. At stage i, where $1 \leq i \leq n$, the argument value val is matched against the pattern pat_i; if the pattern match succeeds, evaluation continues with the evaluation of expression exp_i, with the variables of pat_i replaced by their values as determined by pattern matching. Otherwise we proceed to stage $i + 1$. If no pattern matches (*i.e.*, we reach stage $n+1$), then the application fails with an execution error called a *match failure*.

Here's an example. Consider the following clausal function:

```
val recip : int -> int =
    fn 0 => 0 | n:int => 1 div n
```

This defines an integer-valued reciprocal function on the integers, where the reciprocal of `0` is arbitrarily defined to be `0`. The function has two clauses, one for the argument `0`, the other for non-zero arguments `n`. (Note that `n` is guaranteed to be non-zero because the patterns are considered in order: we reach the pattern `n:int` only if the argument fails to match the pattern `0`.)

The `fun` notation is also generalized so that we may define `recip` using the following more concise syntax:

```
fun recip 0 = 0
  | recip (n:int) = 1 div n
```

One annoying thing to watch out for is that the `fun` form uses an equal sign to separate the pattern from the expression in a clause, whereas the `fn` form uses a double arrow.

Case analysis on the values of a heterogeneous type is performed by application of a clausally-defined function. The notation

```
case exp
  of pat1 => exp1
   | ...
   | patn => expn
```

is short for the application

```
(fn pat1 => exp1
  | ...
  | patn => expn)
exp.
```

Evaluation proceeds by first evaluating *exp*, then matching its value successively against the patterns in the match until one succeeds, and continuing with evaluation of the corresponding expression. The `case` expression fails if no pattern succeeds to match the value.

6.3 Booleans and Conditionals, Revisited

The type `bool` of booleans is perhaps the most basic example of a heterogeneous type. Its values are `true` and `false`. Functions may be defined on booleans using clausal definitions that match against the patterns `true` and `false`.

For example, the negation function may be defined clausally as follows:

```
fun not true = false
  | not false = true
```

The conditional expression

```
if exp then exp1 else exp2
```

is short-hand for the case analysis

```
case exp
  of true => exp1
   | false => exp2
```

which is itself short-hand for the application

```
(fn true => exp1 | false => exp2) exp.
```

The "short-circuit" conjunction and disjunction operations are defined as follows. The expression *exp*$_1$ `andalso` *exp*$_2$ is short for

```
if exp1 then exp2 else false
```

and the expression *exp*$_1$ `orelse` *exp*$_2$ is short for

```
if exp1 then true else exp2.
```

You should expand these into case expressions and check that they behave as expected. Pay particular attention to the evaluation order, and observe that the call-by-value principle is not violated by these expressions.

6.4 Exhaustiveness and Redundancy

Matches are subject to two forms of "sanity check" as an aid to the ML programmer. The first, called *exhaustiveness checking*, ensures that a well-formed match *covers* its domain type in the sense that every value of the domain must match one of its clauses. The second, called *redundancy checking*, ensures that no clause of a match is subsumed by the clauses that precede it. This means that the set of values covered by a clause in a match must not be contained entirely within the set of values covered by the preceding clauses of that match.

Redundant clauses are *always* a mistake — such a clause can never be executed. Redundant rules often arise accidentally. For example, the second rule of the following clausal function definition is redundant:

```
fun not True = false
  | not False = true
```

By capitalizing `True` we have turned it into a variable, rather than a constant pattern. Consequently, *every* value matches the first clause, rendering the second redundant.

Since the clauses of a match are considered in the order they are written, redundancy checking is correspondingly order-sensitive. In particular, changing the order of clauses in a well-formed, irredundant match can make it redundant, as in the following example:

```
fun recip (n:int) = 1 div n
  | recip 0 = 0
```

The second clause is redundant because the first matches *any* integer value, including `0`.

Inexhaustive matches may or may not be in error, depending on whether the match might ever be applied to a value that is not covered by any clause. Here is an example of a function with an inexhaustive match that is plausibly in error:

```
fun is_numeric #"0" = true
  | is_numeric #"1" = true
  | is_numeric #"2" = true
  | is_numeric #"3" = true
  | is_numeric #"4" = true
  | is_numeric #"5" = true
  | is_numeric #"6" = true
  | is_numeric #"7" = true
  | is_numeric #"8" = true
  | is_numeric #"9" = true
```

When applied to, say, `#"a"`, this function fails. Indeed, the function never returns `false` for any argument!

Perhaps what was intended here is to include a *catch-all* clause at the end:

```
fun is_numeric #"0" = true
    | is_numeric #"1" = true
    | is_numeric #"2" = true
    | is_numeric #"3" = true
    | is_numeric #"4" = true
    | is_numeric #"5" = true
    | is_numeric #"6" = true
    | is_numeric #"7" = true
    | is_numeric #"8" = true
    | is_numeric #"9" = true
    | is_numeric _ = false
```

The addition of a final catch-all clause renders the match exhaustive, because any value not matched by the first ten clauses will surely be matched by the eleventh.

Having said that, it is a very bad idea to simply add a catch-all clause to the end of every match to suppress inexhaustiveness warnings from the compiler. The exhaustiveness checker is your friend! Each such warning is a suggestion to double-check that match to be sure that you've not made a silly error of omission, but rather have intentionally left out cases that are ruled out by the invariants of the program. In chapter 10 we will see that the exhaustiveness checker is an extremely valuable tool for managing code evolution.

Chapter 7

Recursive Functions

So far we've only considered very simple functions (such as the reciprocal function) whose value is computed by a simple composition of primitive functions. In this chapter we introduce *recursive* functions, the principal means of iterative computation in ML. Informally, a recursive function is one that computes the result of a call by possibly making further calls to itself. Obviously, to avoid infinite regress, some calls must return their results without making any recursive calls. Those that do must ensure that the arguments are, in some sense, "smaller" so that the process will eventually terminate.

This informal description obscures a central point, namely the means by which we may convince ourselves that a function computes the result that we intend. In general we must prove that for all inputs of the domain type, the body of the function computes the "correct" value of result type. Usually the argument imposes some additional assumptions on the inputs, called the *pre-conditions*. The correctness requirement for the result is called a *post-condition*. Our burden is to prove that for every input satisfying the pre-conditions, the body evaluates to a result satisfying the post-condition. In fact we may carry out such an analysis for many different pre- and post-condition pairs, according to our interest. For example, the ML type checker proves that the body of a function yields a value of the range type (if it terminates) whenever it is given an argument of the domain type. Here the domain type is the pre-condition, and the range type is the post-condition. In most cases we are interested in deeper properties, examples of which we shall consider below.

To prove the correctness of a recursive function (with respect to given pre- and post-conditions) it is typically necessary to use some form of inductive reasoning. The base cases of the induction correspond to those cases that make no recursive calls; the inductive step corresponds to those that do. The beauty of inductive reasoning is that we may *assume* that the recursive calls work correctly when showing that a case involving recursive calls is correct. We must separately show that the base cases satisfy the given pre- and post-conditions. Taken together, these two steps are sufficient to establish the correctness of the function itself, by appeal to an induction principle that justifies the particular pattern of recursion.

No doubt this all sounds fairly theoretical. The point of this chapter is to show that it is also profoundly practical.

7.1 Self-Reference and Recursion

In order for a function to "call itself", it must have a name by which it can refer to itself. This is achieved by using a *recursive value binding*, which are ordinary value bindings qualified by the keyword `rec`. The simplest form of a recursive value binding is as follows:

val rec *var*:*typ* = *val*.

As in the non-recursive case, the left-hand is a pattern, but here the right-hand side must be a value. In fact the right-hand side must be a function expression, since only functions may be defined recursively in ML. The function may refer to itself by using the variable *var*.

Here's an example of a recursive value binding:

```
val rec factorial : int->int =
    fn 0 => 1 | n:int => n * factorial (n-1)
```

Using `fun` notation we may write the definition of factorial much more clearly and concisely as follows:

```
fun factorial 0 = 1
  | factorial (n:int) = n * factorial (n-1)
```

There is obviously a close correspondence between this formulation of `factorial` and the usual textbook definition of the factorial function in terms of recursion equations:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \quad (n > 0) \end{aligned}$$

Recursive value bindings are type-checked in a manner that may, at first glance, seem paradoxical. To check that the binding

```
val rec var : typ = val
```

is well-formed, we ensure that the value *val* has type *typ*, *assuming* that *var* has type *typ*. Since *var* refers to the value *val* itself, we are in effect assuming what we intend to prove while proving it!

(Incidentally, since *val* is required to be a function expression, the type *typ* will always be a function type.)

Let's look at an example. To check that the binding for `factorial` given above is well-formed, we *assume* that the variable `factorial` has type `int->int`, then *check* that its definition, the function

```
fn 0 => 1 | n:int => n * factorial (n-1),
```

has type `int->int`. To do so we must check that each clause has type `int->int` by checking for each clause that its pattern has type `int` and that its expression has type `int`. This is clearly true for the first clause of the definition. For the second, we assume that `n` has type `int`, then check that `n * factorial (n-1)` has type `int`. This is so because of the rules for the primitive arithmetic operations and because of our assumption that `factorial` has type `int->int`.

How are applications of recursive functions evaluated? The rules are almost the same as before, with one modification. We must arrange that all occurrences of the variable standing for the function are replaced by the function itself before we evaluate the body. That way all references to the variable standing for the function itself are indeed references to the function itself!

Suppose that we have the following recursive function binding

```
val rec var : typ =
    fn pat1 => exp1
     | ...
     | patn => expn
```

and we wish to apply *var* to the value *val* of type *typ*. As before, we consider each clause in turn, until we find the first pattern pat_i matching *val*. We proceed, as before, by evaluating exp_i, replacing the variables in pat_i by the bindings determined by pattern matching, but, *in addition*, we replace all occurrences of the *var* by its binding in exp_i before continuing evaluation.

For example, to evaluate `factorial 3`, we proceed by retrieving the binding of `factorial` and evaluating

```
(fn 0=>1 | n:int => n*factorial(n-1))(3).
```

Considering each clause in turn, we find that the first doesn't match, but the second does. We therefore continue by evaluating its right-hand side, the expression `n * factorial(n-1)`, after replacing `n` by `3` and `factorial` by its definition. We are left with the sub-problem of evaluating the expression

```
3 * (fn 0 => 1 | n:int => n*factorial(n-1))(2)
```

Proceeding as before, we reduce this to the sub-problem of evaluating

```
3 * (2 * (fn 0=>1 | n:int => n*factorial(n-1))(1)),
```

which reduces to the sub-problem of evaluating

```
3 * (2 * (1 * (fn 0=>1 | n:int => n*factorial(n-1))(0))),
```

which reduces to

```
3 * (2 * (1 * 1)),
```

which then evaluates to `6`, as desired.

Observe that the repeated substitution of `factorial` by its definition ensures that the recursive calls really do refer to the factorial function itself. Also observe that the size of the sub-problems grows until there are no more recursive calls, at which point the computation can complete. In broad outline, the computation proceeds as follows:

1. `factorial 3`
2. `3 * factorial 2`
3. `3 * 2 * factorial 1`
4. `3 * 2 * 1 * factorial 0`
5. `3 * 2 * 1 * 1`
6. `3 * 2 * 1`
7. `3 * 2`
8. `6`

Notice that the size of the expression first grows (in direct proportion to the argument), then shrinks as the pending multiplications are completed. This growth in expression size corresponds directly to a growth in run-time storage required to record the state of the pending computation.

7.2 Iteration

The definition of `factorial` given above should be contrasted with the following two-part definition:

```
fun helper (0,r:int) = r
  | helper (n:int,r:int) = helper (n-1,n*r)
fun factorial (n:int) = helper (n, 1)
```

First we define a "helper" function that takes two parameters, an integer argument and an *accumulator* that records the running partial result of the computation. The idea is that the accumulator re-associates the pending multiplications in the evaluation trace given above so that they can be performed prior to the recursive call, rather than after it completes. This reduces the space required to keep track of those pending steps. Second, we define `factorial` by calling `helper` with argument `n` and initial accumulator value `1`, corresponding to the product of zero terms (empty prefix).

As a matter of programming style, it is usual to conceal the definitions of helper functions using a `local` declaration. In practice we would make the following definition of the iterative version of `factorial`:

```
local
      fun helper (0,r:int) = r
        | helper (n:int,r:int) = helper (n-1,n*r)
in
      fun factorial (n:int) = helper (n,1)
end
```

This way the helper function is not visible, only the function of interest is "exported" by the declaration.

The important thing to observe about `helper` is that it is *iterative*, or *tail recursive*, meaning that the recursive call is the last step of evaluation of an application of it to an argument. This means that the evaluation trace of a call to `helper` with arguments `(3,1)` has the following general form:

1. `helper (3, 1)`
2. `helper (2, 3)`
3. `helper (1, 6)`
4. `helper (0, 6)`

5. `6`

Notice that there is no growth in the size of the expression because there are no pending computations to be resumed upon completion of the recursive call. Consequently, there is no growth in the space required for an application, in contrast to the first definition given above. Tail recursive definitions are analogous to loops in imperative languages: they merely iterate a computation, without requiring auxiliary storage.

7.3 Inductive Reasoning

Time and space usage are important, but what is more important is that the function compute the intended result. The key to the correctness of a recursive function is an inductive argument establishing its correctness. The critical ingredients are these:

1. An *input-output specification* of the intended behavior stating *pre-conditions* on the arguments and a *post-condition* on the result.

2. A proof that the specification holds for each clause of the function, *assuming* that it holds for any recursive calls.

3. An *induction principle* that justifies the correctness of the function as a whole, given the correctness of its clauses.

We'll illustrate the use of inductive reasoning by a graduated series of examples. First consider the simple, non-tail recursive definition of `factorial` given in section 7.1. One reasonable specification for `factorial` is as follows:

1. *Pre-condition*: $n \geq 0$.

2. *Post-condition*: `factorial` n evaluates to $n!$.

We are to establish the following statement of correctness of `factorial`:

> if $n \geq 0$, then `factorial` n evaluates to $n!$.

That is, we show that the pre-conditions imply the post-condition holds of the result of any application. This is called a *total correctness* assertion because it states not only that the post-condition holds of any result of application, but, moreover, that every application in fact yields a result (subject to the pre-condition on the argument).

In contrast, a *partial correctness* assertion does not insist on termination, only that the post-condition holds whenever the application terminates. This may be stated as the assertion

if $n \geq 0$ and `factorial` n evaluates to p, then $p = n!$.

Notice that this statement is true of a function that diverges whenever it is applied! In this sense a partial correctness assertion is weaker than a total correctness assertion.

Let us establish the total correctness of `factorial` using the pre- and post-conditions stated above. To do so, we apply the principle of *mathematical induction* on the argument n. Recall that this means we are to establish the specification for the case $n = 0$, and, assuming it to hold for $n >= 0$, show that it holds for $n + 1$. The base case, $n = 0$, is trivial: by definition `factorial` n evaluates to 1, which is $0!$. Now suppose that $n = m + 1$ for some $m >= 0$. By the inductive hypothesis we have that `factorial` m evaluates to $m!$ (since $m \geq 0$), and so by definition `factorial` n evaluates to

$$\begin{aligned} n \times m! &= (m+1) \times m! \\ &= (m+1)! \\ &= n!, \end{aligned}$$

as required. This completes the proof.

That was easy. What about the iterative definition of `factorial`? We focus on the behavior of `helper`. A suitable specification is given as follows:

1. *Pre-condition*: $n \geq 0$.

2. *Post-condition*: `helper (`n`, `r`)` evaluates to $n! \times r$.

To show the total correctness of `helper` with respect to this specification, we once again proceed by mathematical induction on n. We leave it as an exercise to give the details of the proof.

With this in hand it is easy to prove the correctness of `factorial` — if $n \geq 0$ then `factorial` n evaluates to the result of `helper (n, 1)`, which evaluates to $n! \times 1 = n!$. This completes the proof.

Helper functions correspond to lemmas, main functions correspond to theorems. Just as we use lemmas to help us prove theorems, we use helper functions to help us define main functions. The foregoing argument shows that this is more than an analogy, but lies at the heart of good programming style.

Here's an example of a function defined by *complete* induction (or *strong* induction), the Fibonacci function, defined on integers $n >= 0$:

```
(* for n>=0, fib n yields the nth Fibonacci number *)
fun fib 0 = 1
  | fib 1 = 1
  | fib (n:int) = fib (n-1) + fib (n-2)
```

The recursive calls are made not only on `n-1`, but also `n-2`, which is why we must appeal to complete induction to justify the definition. This definition of `fib` is very inefficient because it performs many redundant computations: to compute `fib n` requires that we compute `fib (n-1)` and `fib (n-2)`. To compute `fib (n-1)` requires that we compute `fib (n-2)` a second time, and `fib (n-3)`. Computing `fib (n-2)` requires computing `fib (n-3)` again, and `fib (n-4)`. As you can see, there is considerable redundancy here. It can be shown that the running time `fib` of is exponential in its argument, which is quite awful.

Here's a better solution: for each $n >= 0$ compute not only the nth Fibonacci number, but also the $(n-1)$st as well. (For $n = 0$ we define the "-1st" Fibonacci number to be zero). That way we can avoid redundant recomputation, resulting in a linear-time algorithm. Here's the code:

```
(* for n>=0, fib' n evaluates to (a, b), where
   a is the nth Fibonacci number, and
   b is the (n-1)st *)
fun fib' 0 = (1, 0)
  | fib' 1 = (1, 1)
  | fib' (n:int) =
    let
        val (a:int, b:int) = fib' (n-1)
    in
        (a+b, a)
    end
```

You might feel satisfied with this solution since it runs in time linear in n. It turns out (see Graham, Knuth, and Patashnik, *Concrete Mathematics* (Addison-Wesley 1989) for a derivation) that the recurrence

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

has a closed-form solution over the real numbers. This means that the nth Fibonacci number can be calculated directly, without recursion, by using floating point arithmetic. However, this is an unusual case. In most instances recursively-defined functions have no known closed-form solution, so that some form of iteration is inevitable.

7.4 Mutual Recursion

It is often useful to define two functions simultaneously, each of which calls the other (and possibly itself) to compute its result. Such functions are said to be *mutually recursive*. Here's a simple example to illustrate the point, namely testing whether a natural number is odd or even. The most obvious approach is to test whether the number is congruent to 0 mod 2, and indeed this is what one would do in practice. But to illustrate the idea of mutual recursion we instead use the following inductive characterization: 0 is even, and not odd; $n > 0$ is even iff $n - 1$ is odd; $n > 0$ is odd iff $n - 1$ is even. This may be coded up using two mutually-recursive procedures as follows:

```
fun even 0 = true
  | even n = odd (n-1)
and odd 0 = false
  | odd n = even (n-1)
```

Notice that `even` calls `odd` and `odd` calls `even`, so they are not definable separately from one another. We join their definitions using the keyword `and` to indicate that they are defined simultaneously by mutual recursion.

Chapter 8

Type Inference and Polymorphism

8.1 Type Inference

So far we've mostly written our programs in what is known as the *explicitly typed* style. This means that whenever we've introduced a variable, we've assigned it a type at its point of introduction. In particular every variable in a pattern has a type associated with it. As you may have noticed, this gets a little tedious after a while, especially when you're using clausal function definitions. A particularly pleasant feature of ML is that it allows you to omit this type information whenever it can be determined from context. This process is known as *type inference* since the compiler is inferring the missing type information based on context.

For example, there is no need to give a type to the variable `s` in the function

```
fn s:string => s ^ "\n".
```

The reason is that no other type for `s` makes sense, since `s` is used as an argument to string concatenation. Consequently, you may write simply

```
fn s => s ^ "\n",
```

leaving ML to insert "`:string`" for you.

When is it allowable to omit this information? Almost always, with very few exceptions. It is a deep, and important, result about ML that missing type information can (almost) always be reconstructed completely and unambiguously where it is omitted. This is called the *principal typing property* of ML: whenever type information is omitted, there is always a *most general* (*i.e.*, *least restrictive*) way to recover the omitted type information. If there is no way to recover the omitted type information, then the expression is ill-typed. Otherwise there is a "best" way to fill in the blanks, which will (almost) always be found by the compiler. This is an amazingly useful, and widely under-appreciated, property of ML. It means, for example, that the programmer can enjoy the full benefits of a static type system without paying any notational penalty whatsoever!

The prototypical example is the identity function, `fn x=>x`. The body of the function places no constraints on the type of `x`, since it merely returns `x` as the result without performing any computation on it. Since the behavior of the identity function is the same for all possible choices of type for its argument, it is said to be *polymorphic*. Therefore the identity function has *infinitely many* types, one for each choice of the type of the parameter `x`. Choosing the type of `x` to be *typ*, the type of the identity function is *typ*->*typ*. In other words every type for the identity function has the form *typ*->*typ*, where *typ* is the type of the argument.

Clearly there is a pattern here, which is captured by the notion of a *type scheme*. A type scheme is a type expression involving one or more *type variables* standing for an unknown, but arbitrary type expression. Type variables are written `'a` (pronounced "α"), `'b` (pronounced "β"), `'c` (pronounced "γ"), *etc.*. An *instance* of a type scheme is obtained by replacing each of the type variables occurring in it with a type scheme, with the same type scheme replacing each occurrence of a given type variable. For example, the type scheme `'a->'a` has instances `int->int`, `string->string`, `(int*int)->(int*int)`, and `('b->'b)->('b->'b)`, among infinitely many others. However, it does *not* have the type `int->string` as instance, since we are constrained to replace all occurrences of a type variable by the same type scheme. However, the type scheme `'a->'b` has both `int->int` and `int->string` as instances since there are different type variables occurring in the domain and range positions.

Type schemes are used to express the polymorphic behavior of functions. For example, we may write `fn x:'a=>x` for the polymorphic identity function of type `'a->'a`, meaning that the behavior of the identity function is independent of the type of `x`. Similarly, the behavior of the function `fn (x,y)=>x+1` is independent of the type of `y`, but constrains the type of `x` to be `int`. This may be expressed using type schemes by writing this function in the explicitly-typed form `fn (x:int,y:'a)=>x+1` with type `int*'a->int`.

In these examples we needed only one type variable to express the polymorphic behavior of a function, but usually we need more than one. For example, the function `fn (x,y) = x` constrains neither the type of `x` nor the type of `y`. Consequently we may choose their types freely and independently of one another. This may be expressed by writing this function in the form `fn (x:'a,y:'b)=>x` with type scheme `'a*'b->'a`. Notice that while it is correct to assign the type `'a*'a->'a` to this function, doing so would be overly restrictive since the types of the two parameters need not be the same. However, we could *not* assign the type `'a*'b->'c` to this function because the type of the result must be the same as the type of the first parameter: it returns its first parameter when invoked! The type scheme `'a*'b->'a` precisely captures the constraints that must be satisfied for the function to be type correct. It is said to be the *most general* or *principal type scheme* for the function.

It is a remarkable fact about ML that *every expression* (with the exception of a few pesky examples that we'll discuss below) *has a principal type scheme*. That is, there is (almost) always a *best* or *most general* way to infer types for expressions that *maximizes generality*, and hence *maximizes flexibility* in the use of the expression. Every expression "seeks its own depth" in the sense that an occurrence of that expression is assigned a type that is an instance of its principal type scheme determined by the context of use.

For example, if we write

```
(fn x=>x)(0),
```

the context forces the type of the identity function to be `int->int`, and if we write

```
(fn x=>x)(fn x=>x)(0)
```

the context forces the instance `(int->int)->(int->int)` of the principal type scheme for the identity at the first occurrence, and the instance `int->int` for the second.

How is this achieved? Type inference is a process of *constraint satisfaction*. First, the expression determines a set of equations governing the relationships among the types of its subexpressions. For example, if a function is applied to an argument, then a constraint equating the domain type of the function with the type of the argument is generated. Second, the constraints are solved using a process similar to Gaussian elimination, called *unification*. The equations can be classified by their solution sets as follows:

1. *Overconstrained*: there is no solution. This corresponds to a type error.

2. *Underconstrained*: there are many solutions. There are two sub-cases: *ambiguous* (due to overloading, which we will discuss further in section 8.3), or *polymorphic* (there is a "best" solution).

3. *Uniquely determined:* there is precisely one solution. This corresponds to a completely unambiguous type inference problem.

The free type variables in the solution to the system of equations may be thought of as determining the "degrees of freedom" or "range of polymorphism" of the type of an expression — the constraints are solvable for any choice of types to substitute for these free type variables.

This description of type inference as a constraint satisfaction procedure accounts for the notorious obscurity of type checking errors in ML. If a program is not type correct, then the system of constraints associated with it will not have a solution. The type inference procedure attempts to find a solution to these constraints, and at some point discovers that it cannot succeed. It is fundamentally impossible to attribute this inconsistency to any particular constraint; all that can be said is that the constraint set as a whole has no solution. The checker usually reports the first unsatisfiable equation it encounters, but this may or may not correspond to the "reason" (in the mind of the programmer) for the type error. The usual method for finding the error is to insert sufficient type information to narrow down the source of the inconsistency until the source of the difficulty is uncovered.

8.2 Polymorphic Definitions

There is an important interaction between polymorphic expressions and value bindings that may be illustrated by the following example. Suppose that we wish to bind the identity function to a variable `I` so that we may refer to it by name. We've previously observed that the identity function is polymorphic, with principal type scheme `'a->'a`. This may be captured by ascribing this type scheme to the variable `I` at the `val` binding. That is, we may write

```
val I : 'a->'a = fn x=>x
```

to ascribe the type scheme `'a->'a` to the variable `I`. (We may also write

```
fun I(x:'a):'a = x
```

for an equivalent binding of `I`.) Having done this, *each use of* `I` *determines a distinct instance of the ascribed type scheme* `'a->'a`. That is, both `I 0` and `I`

`I 0` are well-formed expressions, the first assigning the type `int->int` to `I`, the second assigning the types

```
(int->int)->(int->int)
```

and

```
int->int
```

to the two occurrences of `I`. Thus the variable `I` behaves precisely the same as its definition, `fn x=>x`, in any expression where it is used.

As a convenience ML also provides a form of type inference on value bindings that eliminates the need to ascribe a type scheme to the variable when it is bound. If no type is ascribed to a variable introduced by a `val` binding, then it is implicitly ascribed the principal type scheme of the right-hand side. For example, we may write

```
val I = fn x=>x
```

or

```
fun I(x) = x
```

as a binding for the variable . The type checker implicitly assigns the principal type scheme, `'a->'a`, of the binding to the variable `I`. In practice we often allow the type checker to infer the principal type of a variable, but it is often good form to explicitly indicate the intended type as a consistency check and for documentation purposes.

The treatment of `val` bindings ensures that a bound variable has precisely the same types as its binding. In other words the type checker behaves as though all uses of the bound variable are implicitly replaced by its binding before type checking. Since this may involve replication of the binding, the meaning of a program is not necessarily preserved by this transformation. (Think, for example, of any expression that opens a window on your screen: if you replicate the expression and evaluate it twice, it will open two windows. This is not the same as evaluating it only once, which results in one window.)

To ensure semantic consistency, variables introduced by a `val` binding are allowed to be polymorphic *only if* the right-hand side is a value. This is called the *value restriction* on polymorphic declarations. For `fun` bindings this restriction is always met since the right-hand side is implicitly a lambda expression, which is a value. However, it might be thought that the following declaration introduces a polymorphic variable of type `'a -> 'a`, but in fact it is rejected by the compiler:

```
val J = I I
```

The reason is that the right-hand side is not a value; it requires computation to determine its value. It is therefore ruled out as inadmissible for polymorphism; the variable `J` may not be used polymorphically in the remainder of the program. In this case the difficulty may be avoided by writing instead

```
fun J x = I I x
```

because now the binding of `J` is a lambda, which is a value.

In some rare circumstances this is not possible, and some polymorphism is lost. For example, the following declaration of a value of list type,[1]

```
val l = nil @ nil,
```

does not introduce an identifier with a polymorphic type, even though the almost equivalent declaration

```
val l = nil
```

does do so. Since the right-hand side is a list, we cannot apply the "trick" of defining `l` to be a function; we are stuck with a loss of polymorphism in this case. This particular example is not very impressive, but occasionally similar examples do arise in practice.

Why is the value restriction necessary? Later on, when we study mutable storage, we'll see that *some* restriction on polymorphism is essential if the language is to be type safe. The value restriction is an easily-remembered sufficient condition for soundness, but as the examples above illustrate, it is by no means necessary. The designers of ML were faced with a choice of simplicity *vs* flexibility; in this case they opted for simplicity at the expense of some expressiveness in the language.

8.3 Overloading

Type information cannot always be omitted. There are a few corner cases that create problems for type inference, most of which arise because of concessions that are motivated by long-standing, if dubious, notational practices.

The main source of difficulty stems from *overloading* of arithmetic operators. As a concession to long-standing practice in informal mathematics

[1] To be introduced in chapter 9.

and in many programming languages, the same notation is used for both integer and floating point arithmetic operations. As long as we are programming in an explicitly-typed style, this convention creates no particular problems. For example, in the function

```
fn x:int => x+x
```

it is clear that integer addition is called for, whereas in the function

```
fn x:real => x+x
```

it is equally obvious that floating point addition is intended.

However, if we *omit* type information, then a problem arises. What are we to make of the function

```
fn x => x+x ?
```

Does "+" stand for integer or floating point addition? There are two distinct reconstructions of the missing type information in this example, corresponding to the preceding two explictly-typed programs. Which is the compiler to choose?

When presented with such a program, the compiler has two choices:

1. Declare the expression ambiguous, and force the programmer to provide enough explicit type information to resolve the ambiguity.

2. Arbitrarily choose a "default" interpretation, say the integer arithmetic, that forces one interpretation or another.

Each approach has its advantages and disadvantages. Many compilers choose the second approach, but issue a warning indicating that it has done so. To avoid ambiguity, explicit type information is required from the programmer.

The situation is actually a bit more subtle than the preceding discussion implies. The reason is that the type inference process makes use of the surrounding context of an expression to help resolve ambiguities. For example, if the expression `fn x=>x+x` occurs in the following, larger expression, there is in fact no ambiguity:

```
(fn x => x+x)(3).
```

Since the function is applied to an integer argument, there is no question that the only possible resolution of the missing type information is to treat `x` as having type `int`, and hence to treat + as integer addition.

The important question is how much context is considered before the situation is considered ambiguous? The rule of thumb is that context is considered up to the nearest enclosing function declaration. For example, consider the following example:

```
let
    val double = fn x => x+x
in
    (double 3, double 4)
end
```

The function expression `fn x=>x+x` will be flagged as ambiguous, even though its only uses are with integer arguments. The reason is that value bindings are considered to be "units" of type inference for which all ambiguity must be resolved before type checking continues. If your compiler adopts the integer interpretation as default, the above program will be accepted (with a warning), but the following one will be rejected:

```
let
    val double = fn x => x+x
in
    (double 3.0, double 4.0)
end
```

Finally, note that the following program *must* be rejected because no resolution of the overloading of addition can render it meaningful:

```
let
    val double = fn x => x+x
in
    (double 3, double 3.0)
end
```

The ambiguity must be resolved at the `val` binding, which means that the compiler must commit at that point to treating the addition operation as either integer or floating point. No single choice can be correct, since we subsequently use `double` at both types.

A closely-related source of ambiguity is arises from the "record elision" notation described in chapter 5. Consider the function `#name`, defined by

```
fun #name {name=n:string, ...} = n
```

which selects the name field of a record. This definition is *ambiguous* because the compiler cannot uniquely determine the domain type of the function! Any of the following types are legitimate domain types for #name, none of which is "best":

```
{name:string}
{name:string,salary:real}
{name:string,salary:int}
{name:string,address:string}
```

Of course there are infinitely many such examples, none of which is clearly preferable to the other. This function definition is therefore rejected as ambiguous by the compiler — there is no one interpretation of the function that suffices for all possible uses.

In chapter 5 we mentioned that functions such as #name are *pre-defined* by the ML compiler, yet we just now claimed that such a function definition is rejected as ambiguous. Isn't this a contradiction? Not really, because what happens is that each occurrence of #name is replaced by the function

```
fn {name=n,...} = n
```

and then context is used to resolve the "local" ambiguity. This works well, provided that the complete record type of the arguments to #name can be determined from context. If not, the uses are rejected as ambiguous. Thus, the following expression is well-typed

```
fn r : {name:string,address:string,salary:int} =>
   (#name r, #address r)
```

because the record type of r is explicitly given. If the type of r were omitted, the expression would be rejected as ambiguous (unless the context resolves the ambiguity.)

Chapter 9

Programming with Lists

9.1 List Primitives

In chapter 5 we noted that aggregate data structures are especially easy to handle in ML. In this chapter we consider another important aggregate type, the *list* type. In addition to being an important form of aggregate type it also illustrates two other general features of the ML type system:

1. *Type constructors*, or *parameterized types*. The type of a list reveals the type of its elements.

2. *Recursive types*. The set of values of a list type are given by an inductive definition.

Informally, the values of type *typ* `list` are the finite lists of values of type *typ*. More precisely, the values of type *typ* `list` are given by an *inductive definition*, as follows:

1. `nil` is a value of type *typ* `list`.

2. if h is a value of type *typ*, and t is a value of type *typ* `list`, then h`::`t is a value of type *typ* `list`.

3. Nothing else is a value of type *typ* `list`.

The type expression *typ* `list` is a postfix notation for the application of the *type constructor* `list` to the type *typ*. Thus `list` is a kind of "function" mapping types to types: given a type *typ*, we may apply `list` to it to get another type, written *typ* `list`. The forms `nil` and `::` are the *value constructors* of type *typ* `list`. The nullary (no argument) constructor `nil` may be thought of as the empty list. The binary (two argument) constructor `::` constructs a non-empty list from a value h of type *typ* and another value t of type *typ* `list`; the resulting value, h`::`t, of type *typ* `list` is pronounced "h cons t" (for historical reasons). We say that "h is *cons'd onto* t", that h is the *head* of the list, and that t is its *tail*.

The definition of the values of type *typ* `list` given above is an example of an *inductive definition*. The type is said to be *recursive* because this definition is "self-referential" in the sense that the values of type *typ* `list` are defined in terms of (other) values of the same type. This is especially clear if we examine the types of the value constructors for the type *typ* `list`:

```
val nil : typ list
val (op ::) : typ * typ list -> typ list
```

The notation `op ::` is used to *refer* to the `::` operator as a function, rather than to *use* it to form a list, which requires infix notation.

Two things are notable here:

1. The `::` operation takes as its second argument a value of type *typ* `list`, and yields a result of type *typ* `list`. This self-referential aspect is characteristic of an inductive definition.

2. Both `nil` and `op ::` are *polymorphic* in the type of the underlying elements of the list. Thus `nil` is the empty list of type *typ* `list` for any element type *typ*, and `op ::` constructs a non-empty list independently of the type of the elements of that list.

It is easy to see that a value *val* of type *typ* `list` has the form

$$val_1 \texttt{::(} val_2 \texttt{:: (} \cdots \texttt{ ::(} val_n \texttt{::nil)} \cdots \texttt{))}$$

for some $n \geq 0$, where val_i is a value of type typ_i for each $1 \leq i \leq n$. For according to the inductive definition of the values of type *typ* `list`, the value *val* must either be `nil`, which is of the above form, or val_1`::`val', where val' is a value of type *typ* `list`. By induction val' has the form

$$\texttt{(} val_2 \texttt{:: (} \cdots \texttt{ ::(} val_n \texttt{::nil)} \cdots \texttt{))}$$

and hence *val* again has the specified form.

By convention the operator `::` is *right-associative*, so we may omit the parentheses and just write

$$val_1 \texttt{::} val_2 \texttt{::} \cdots \texttt{::} val_n \texttt{::nil}$$

as the general form of *val* of type *typ* `list`. This may be further abbreviated using *list notation*, writing

[val_1, val_2, ..., val_n]

for the same list. This notation emphasizes the interpretation of lists as finite sequences of values, but it obscures the fundamental inductive character of lists as being built up from `nil` using the `::` operation.

9.2 Computing With Lists

How do we compute with values of list type? Since the values are defined inductively, it is natural that functions on lists be defined recursively, using a clausal definition that analyzes the structure of a list. Here's a definition of the function `length` that computes the number of elements of a list:

```
fun length nil = 0
    | length (_::t) = 1 + length t
```

The definition is given by induction on the structure of the list argument. The base case is the empty list, `nil`. The inductive step is the non-empty list `_::t` (notice that we do not need to give a name to the head). Its definition is given in terms of the tail of the list `t`, which is "smaller" than the list `_::t`. The type of `length` is `'a list -> int`; it is defined for lists of values of any type whatsoever.

We may define other functions following a similar pattern. Here's the function to append two lists:

```
fun append (nil, l) = l
  | append (h::t, l) = h :: append (t, l)
```

This function is built into ML; it is written using infix notation as exp_1 `@` exp_2. The running time of `append` is proportional to the length of the first list, as should be obvious from its definition.

Here's a function to reverse a list.

```
fun rev nil = nil
  | rev (h::t) = rev t @ [h]
```

Its running time is $O(n^2)$, where n is the length of the argument list. This can be demonstrated by writing down a recurrence that defines the running time $T(n)$ on a list of length n.

$$\begin{aligned} T(0) &= O(1) \\ T(n+1) &= T(n) + O(n) \end{aligned}$$

Solving the recurrence we obtain the result $T(n) = O(n^2)$.

Can we do better? Oddly, we can take advantage of the *non-associativity* of `::` to give a tail-recursive definition of `rev`.

```
local
    fun helper (nil, a) = a
      | helper (h::t, a) = helper (t, h::a)
in
    fun rev' l = helper (l, nil)
end
```

The general idea of introducing an accumulator is the same as before, except that by re-ordering the applications of `::` we reverse the list! The helper function reverses its first argument and prepends it to its second argument. That is, `helper (l, a)` evaluates to `(rev l) @ a`, where we assume here an independent definition of `rev` for the sake of the specification. Notice that `helper` runs in time proportional to the length of its first argument, and hence `rev'` runs in time proportional to the length of its argument.

The correctness of functions defined on lists may be established using the principle of *structural induction*. We illustrate this by establishing that the function `helper` satisfies the following specification:

> for every l and a of type *typ* `list`, `helper(`l`, `a`)` evaluates to the result of appending a to the reversal of l.

That is, there are no pre-conditions on l and a, and we establish the post-condition that `helper (`l`, `a`)` yields `(rev `l`) @ `a.

The proof is by structural induction on the list l. If l is `nil`, then `helper (`l`,`a`)` evaluates to a, which fulfills the post-condition. If l is the list h`::`t, then the application `helper (`l`, `a`)` reduces to the value of `helper (`t`, (`h`::`a`))`. By the inductive hypothesis this is just `(rev `t`) @ (`h` :: `a`)`, which is equivalent to `(rev `t`) @ [`h`] @ `a. But this is just `rev (`h`::`t`) @ `a, which was to be shown.

The principle of structural induction may be summarized as follows. To show that a function works correctly for every list l, it suffices to show

1. The correctness of the function for the empty list, `nil`, and

2. The correctness of the function for h`::`t, assuming its correctness for t.

As with mathematical induction over the natural numbers, structural induction over lists allows us to focus on the *basic* and *incremental* behavior of a function to establish its correctness for all lists.

Chapter 10

Concrete Data Types

10.1 Datatype Declarations

Lists are one example of the general notion of a *recursive type*. ML provides a general mechanism, the `datatype` declaration, for introducing programmer-defined recursive types. Earlier we introduced `type` declarations as an abbreviation mechanism. Types are given names as documentation and as a convenience to the programmer, but doing so is semantically inconsequential — one could replace all uses of the type name by its definition and not affect the behavior of the program. In contrast the `datatype` declaration provides a means of introducing a *new* type that is distinct from all other types and that does not merely stand for some other type. It is the means by which the ML type system may be extended by the programmer.

The `datatype` declaration in ML has a number of facets. A `datatype` declaration introduces

1. One or more new type constructors. The type constructors introduced may, or may not, be mutually recursive.

2. One or more new value constructors for each of the type constructors introduced by the declaration.

The type constructors may take zero or more arguments; a zero-argument, or *nullary*, type constructor is just a type. Each value constructor may also take zero or more arguments; a nullary value constructor is just a constant. The type and value constructors introduced by the declaration are "new" in the sense that they are distinct from all other type and value constructors previously introduced; if a datatype re-defines an "old" type or value constructor, then the old definition is shadowed by the new one, rendering the old ones inaccessible in the scope of the new definition.

10.2 Non-Recursive Datatypes

Here's a simple example of a nullary type constructor with four nullary value constructors.

```
datatype suit = Spades | Hearts | Diamonds | Clubs
```

This declaration introduces a new type `suit` with four nullary value constructors, `Spades`, `Hearts`, `Diamonds`, and `Clubs`. This declaration may be read as introducing a type `suit` such that a value of type `suit` is either `Spades`, or `Hearts`, or `Diamonds`, or `Clubs`. There is no significance to the ordering of the constructors in the declaration; we could just as well have written

```
datatype suit = Hearts | Diamonds | Spades | Clubs
```

(or any other ordering, for that matter). It is conventional to capitalize the names of value constructors, but this is not required by the language.

Given the declaration of the type suit, we may define functions on it by case analysis on the value constructors using a clausal function definition. For example, we may define the suit ordering in the card game of bridge by the function

```
fun outranks (Spades, Spades) = false
  | outranks (Spades, _) = true
  | outranks (Hearts, Spades) = false
  | outranks (Hearts, Hearts) = false
  | outranks (Hearts, _) = true
  | outranks (Diamonds, Clubs) = true
  | outranks (Diamonds, _) = false
  | outranks (Clubs, _) = false
```

This defines a function of type `suit * suit -> bool` that determines whether or not the first `suit` outranks the second.

Data types may be *parameterized* by a type. For example, the declaration

```
datatype 'a option = NONE | SOME of 'a
```

introduces the unary type constructor `'a option` with two value constructors, `NONE`, with no arguments, and `SOME`, with one. The values of type *typ* `option` are

1. The constant `NONE`, and

2. Values of the form `SOME` *val*, where *val* is a value of type *typ*.

For example, some values of type `string option` are `NONE`, `SOME "abc"`, and `SOME "def"`.

The option type constructor is pre-defined in Standard ML. One common use of option types is to handle functions with an optional argument. For example, here is a function to compute the base-b exponential function for natural number exponents that defaults to base 2:

```
fun expt (NONE, n) = expt (SOME 2, n)
  | expt (SOME b, 0) = 1
  | expt (SOME b, n) =
    if n mod 2 = 0 then
       expt (SOME b*b, n div 2)
    else
       b * expt (SOME b, n-1)
```

The advantage of the option type in this sort of situation is that it avoids the need to make a special case of a particular argument, *e.g.,* using `0` as first argument to mean "use the default exponent".

A related use of option types is in aggregate data structures. For example, an address book entry might have a record type with fields for various bits of data about a person. But not all data is relevant to all people. For example, someone may not have a spouse, but they all have a name. For this we might use a type definition of the form

```
type entry = { name:string, spouse:string option }
```

so that one would create an entry for an unmarried person with a `spouse` field of `NONE`.

Option types may also be used to represent an optional result. For example, we may wish to define a function `reciprocal` that returns the reciprocal of an integer, if it has one, and otherwise indicates that it has no reciprocal. This is achieve by defining `reciprocal` to have type `int -> int option` as follows:

```
fun reciprocal 0 = NONE
  | reciprocal n = SOME (1 div n)
```

To use the result of a call to *reciprocal* we must perform a case analysis of the form

```
case (reciprocal exp
  of NONE => exp1
   | SOME r => exp2
```

where exp_1 covers the case that exp has no reciprocal, and exp_2 covers the case that exp has reciprocal `r`.

10.3 Recursive Datatypes

The next level of generality is the recursive type definition. For example, one may define a type *typ* `tree` of binary trees with values of type *typ* at the nodes using the following declaration:

```
datatype 'a tree =
  Empty |
  Node of 'a tree * 'a * 'a tree
```

This declaration corresponds to the informal definition of binary trees with values of type *typ* at the nodes:

1. The empty tree `Empty` is a binary tree.

2. If *tree_1* and *tree_2* are binary trees, and *val* is a value of type *typ*, then `Node (`*tree_1*`, `*val*`, `*tree_2*`)` is a binary tree.

3. Nothing else is a binary tree.

The distinguishing feature of this definition is that it is *recursive* in the sense that binary trees are constructed out of other binary trees, with the empty tree serving as the base case.

(Incidentally, a *leaf* in a binary tree is here represented as a node both of whose children are the empty tree. This definition of binary trees is analogous to starting the natural numbers with zero, rather than one. One can think of the children of a node in a binary tree as the "predecessors" of that node, the only difference compared to the usual definition of predecessor being that a node has two, rather than one, predecessors.)

To compute with a recursive type, use a recursive function. For example, here is the function to compute the *height* of a binary tree:

```
fun height Empty = 0
  | height (Node (lft, _, rht)) =
    1 + max (height lft, height rht)
```

Notice that `height` is called recursively on the children of a node, and is defined outright on the empty tree. This pattern of definition is another instance of structural induction (on the `tree` type). The function `height` is said to be defined by induction on the structure of a tree. The general idea is to define the function directly for the base cases of the recursive type (*i.e.*, value constructors with no arguments or whose arguments do not involve values of the type being defined), and to define it for non-base cases in terms of its definitions for the constituent values of that type. We will see numerous examples of this as we go along.

Here's another example. The *size* of a binary tree is the number of nodes occurring in it. Here's a straightforward definition in ML:

```
fun size Empty = 0
  | size (Node (lft, _, rht)) =
    1 + size lft + size rht
```

The function size is defined by structural induction on trees.

A word of warning. One reason to capitalize value constructors is to avoid a pitfall in the ML syntax that we mentioned in chapter 2. Suppose we gave the following definition of `size`:

```
fun size empty = 0
  | size (Node (lft, _, rht)) =
    1 + size lft + size rht
```

The compiler will warn us that the second clause of the definition is *redundant*! Why? Because `empty`, spelled with a lower-case "e", is a *variable*, not a *constructor*, and hence matches *any* tree whatsoever. Consequently the second clause never applies. By capitalizing constructors we can hope to make mistakes such as these more evident, but in practice you are bound to run into this sort of mistake.

The `tree` data type is appropriate for binary trees: those for which every node has exactly two children. (Of course, either or both children might be the empty tree, so we may consider this to define the type of trees with *at most* two children; it's a matter of terminology which interpretation you prefer.) It should be obvious how to define the type of *ternary* trees, whose nodes have at most three children, and so on for other fixed arities. But what if we wished to define a type of trees with a *variable* number of children? In a so-called *variadic tree* some nodes might have three children, some might have two, and so on. This can be achieved in at least two ways. One way combines lists and trees, as follows:

```
datatype 'a tree =
  Empty |
  Node of 'a * 'a tree list
```

Each node has a *list* of children, so that distinct nodes may have different numbers of children. Notice that the empty tree is distinct from the tree with one node and no children because there is no data associated with the empty tree, whereas there is a value of type `'a` at each node.

Another approach is to simultaneously define trees and "forests". A variadic tree is either empty, or a node gathering a "forest" to form a tree; a forest is either empty or a variadic tree together with another forest. This leads to the following definition:

```
datatype 'a tree =
  Empty |
  Node of 'a * 'a forest
and 'a forest =
  Nil |
  Cons of 'a tree * 'a forest
```

This example illustrates the introduction of two *mutually recursive datatypes*.

Mutually recursive datatypes beget mutually recursive functions. Here's a definition of the size (number of nodes) of a variadic tree:

```
fun size_tree Empty = 0
  | size_tree (Node (_, f)) = 1 + size_forest f
and size_forest Nil = 0
  | size_forest (Cons (t, f')) = size_tree t + size_forest f'
```

Notice that we define the size of a tree in terms of the size of a forest, and *vice versa*, just as the type of trees is defined in terms of the type of forests.

Many other variations are possible. Suppose we wish to define a notion of binary tree in which data items are associated with branches, rather than nodes. Here's a `datatype` declaration for such trees:

```
datatype 'a tree =
    Empty |
    Node of 'a branch * 'a branch
and 'a branch =
    Branch of 'a * 'a tree
```

In contrast to our first definition of binary trees, in which the branches from a node to its children were *implicit*, we now make the branches themselves *explicit*, since data is attached to them.

For example, we can collect into a list the data items labelling the branches of such a tree using the following code:

```
fun collect Empty = nil
  | collect (Node (Branch (ld, lt), Branch (rd, rt))) =
    ld :: rd :: (collect lt) @ (collect rt)
```

10.4 Heterogeneous Data Structures

Returning to the original definition of binary trees (with data items at the nodes), observe that the *type* of the data items at the nodes must be the same for every node of the tree. For example, a value of type `int tree` has an integer at every node, and a value of type string tree has a string at every node. Therefore an expression such as

```
Node (Empty, 43, Node (Empty, "43", Empty))
```

is ill-typed. The type system insists that trees be *homogeneous* in the sense that the type of the data items is the same at every node.

It is quite rare to encounter heterogeneous data structures in real programs. For example, a dictionary with strings as keys might be represented as a binary search tree with strings at the nodes; there is no need for heterogeneity to represent such a data structure. But occasionally one might wish to work with a *heterogeneous* tree, whose data values at each node are of different types. How would one represent such a thing in ML?

To discover the answer, first think about how one might manipulate such a data structure. When accessing a node, we would need to check at run-time whether the data item is an integer or a string; otherwise we would not know whether to, say, add 1 to it, or concatenate `"1"` to the end of it. This suggests that the data item must be *labelled* with sufficient information so that we may determine the type of the item at run-time. We must also be able to recover the underlying data item itself so that familiar operations (such as addition or string concatenation) may be applied to it.

The required labelling and discrimination is neatly achieved using a datatype declaration. Suppose we wish to represent the type of integer-or-string trees. First, we define the type of values to be integers or strings, marked with a constructor indicating which:

```
datatype int_or_string =
  Int of int |
  String of string
```

Then we define the type of interest as follows:

```
type int_or_string_tree =
  int_or_string tree
```

Voila! Perfectly natural and easy — heterogeneity is really a special case of homogeneity!

10.5 Abstract Syntax

`Datatype` declarations and pattern matching are extremely useful for defining and manipulating the *abstract syntax* of a language. For example, we may define a small language of arithmetic expressions using the following declaration:

```
datatype expr =
  Numeral of int |
  Plus of expr * expr |
  Times of expr * expr
```

This definition has only three clauses, but one could readily imagine adding others. Here is the definition of a function to evaluate expressions of the language of arithmetic expressions written using pattern matching:

```
fun eval (Numeral n) = Numeral n
  | eval (Plus (e1, e2)) =
    let
        val Numeral n1 = eval e1
        val Numeral n2 = eval e2
    in
        Numeral (n1+n2)
    end
  | eval (Times (e1, e2)) =
    let
        val Numeral n1 = eval e1
        val Numeral n2 = eval e2
    in
        Numeral (n1*n2)
    end
```

The combination of datatype declarations and pattern matching contributes enormously to the readability of programs written in ML. A less obvious, but more important, benefit is the error checking that the compiler can perform for you if you use these mechanisms in tandem. As an example, suppose that we extend the type expr with a new component for the reciprocal of a number, yielding the following revised definition:

```
datatype expr =
  Numeral of int |
  Plus of expr * expr |
  Times of expr * expr |
  Recip of expr
```

First, observe that the "old" definition of eval is no longer applicable to values of type expr! For example, the expression

```
eval (Plus (Numeral 1, Numeral 2))
```

is ill-typed, even though it doesn't use the Recip constructor. The reason is that the re-declaration of expr introduces a "new" type that just happens to have the same name as the "old" type, but is in fact distinct from it. This is a boon because it reminds us to recompile the old code relative to the new definition of the expr type.

Second, upon recompiling the definition of eval we encounter an *inexhaustive match* warning: the old code no longer applies to every value of type expr according to its new definition! We are of course lacking a case for Recip, which we may provide as follows:

```
fun eval (Numeral n) = Numeral n
  | eval (Plus (e1, e2)) = ... as before ...
  | eval (Times (e1, e2)) = ... as before ...
  | eval (Recip e) =
    let
        val Numeral n = eval e
    in
        Numeral (1 div n)
    end
```

The value of the checks provided by the compiler in such cases cannot be overestimated. When recompiling a large program after making a change to a datatype declaration the compiler will automatically point out *every line of code* that must be changed to conform to the new definition; it is impossible to forget to attend to even a single case. This is a tremendous help to the developer, especially if she is not the original author of the code being modified and is another reason why the static type discipline of ML is a positive benefit, rather than a hindrance, to programmers.

Chapter 11

Higher-Order Functions

11.1 Functions as Values

Values of function type are *first-class*, which means that they have the same rights and privileges as values of any other type. In particular, functions may be passed as arguments and returned as results of other functions, and functions may be stored in and retrieved from data structures such as lists and trees. We will see that first-class functions are an important source of expressive power in ML.

Functions which take functions as arguments or yield functions as results are known as *higher-order functions* (or, less often, as *functionals* or *operators*). Higher-order functions arise frequently in mathematics. For example, the differential operator is the higher-order function that, when given a (differentiable) function on the real line, yields its first derivative as a function on the real line. We also encounter functionals mapping functions to real numbers, and real numbers to functions. An example of the former is provided by the definite integral viewed as a function of its integrand, and an example of the latter is the definite integral of a given function on the interval $[a, x]$, viewed as a function of a, that yields the area under the curve from a to x as a function of x.

Higher-order functions are less familiar tools for many programmers since the best-known programming languages have only rudimentary mechanisms to support their use. In contrast higher-order functions play a prominent role in ML, with a variety of interesting applications. Their use may be classified into two broad categories:

1. *Abstracting patterns of control.* Higher-order functions are design patterns that "abstract out" the details of a computation to lay bare the

skeleton of the solution. The skeleton may be fleshed out to form a solution of a problem by applying the general pattern to arguments that isolate the specific problem instance.

2. *Staging computation.* It arises frequently that computation may be *staged* by expending additional effort "early" to simplify the computation of "later" results. Staging can be used both to improve efficiency and, as we will see later, to control sharing of computational resources.

11.2 Binding and Scope

Before discussing these programming techniques, we will review the critically important concept of *scope* as it applies to function definitions. Recall that Standard ML is a *statically scoped* language, meaning that identifiers are resolved according to the static structure of the program. A *use* of the variable *var* is considered to be a reference to the *nearest lexically enclosing declaration of var*. We say "nearest" because of the possibility of shadowing; if we re-declare a variable *var*, then subsequent uses of *var* refer to the "most recent" (lexically!) declaration of it; any "previous" declarations are temporarily shadowed by the latest one.

This principle is easy to apply when considering sequences of declarations. For example, it should be clear by now that the variable `y` is bound to `32` after processing the following sequence of declarations:

```
val x = 2            (* x=2 *)
val y = x*x          (* y=4 *)
val x = y*x          (* x=8 *)
val y = x*y          (* y=32 *)
```

In the presence of function definitions the situation is the same, but it can be a bit tricky to understand at first.

Here's an example to test your grasp of the lexical scoping principle:

```
val x = 2
fun f y = x+y
val x = 3
val z = f 4
```

After processing these declarations the variable `z` is bound to `6`, not to `7`! The reason is that the occurrence of `x` in the body of `f` refers to the *first* declaration of `x` since it is the nearest lexically enclosing declaration of the occurence, *even though* it has been subsequently re-declared.

This example illustrates three important points:

1. Binding is not assignment! If we were to view the second binding of `x` as an assignment statement, then the value of `z` would be `7`, not `6`.

2. Scope resolution is *lexical*, not *temporal.* We sometimes refer to the "most recent" declaration of a variable, which has a temporal flavor, but we always mean "nearest lexically enclosing at the point of occurrence".

3. "Shadowed" bindings are not lost. The "old" binding for `x` is still available (through calls to `f`), even though a more recent binding has shadowed it.

One way to understand what's going on here is through the concept of a *closure*, a technique for implementing higher-order functions. When a function expression is evaluated, a copy of the environment is attached to the function. Subsequently, all free variables of the function (*i.e.*, those variables not occurring as parameters) are resolved with respect to the environment attached to the function; the function is therefore said to be "closed" with respect to the attached environment. This is achieved at function application time by "swapping" the attached environment of the function for the environment active at the point of the call. The swapped environment is restored after the call is complete. Returning to the example above, the environment associated with the function `f` contains the declaration `val x = 2` to record the fact that at the time the function was evaluated, the variable `x` was bound to the value `2`. The variable `x` is subsequently re-bound to `3`, but when `f` is applied, we temporarily reinstate the binding of `x` to `2`, add a binding of `y` to `4`, then evaluate the body of the function, yielding `6`. We then restore the binding of `x` and drop the binding of `y` before yielding the result.

11.3 Returning Functions

While seemingly very simple, the principle of lexical scope is the source of considerable expressive power. We'll demonstrate this through a series of examples.

To warm up let's consider some simple examples of passing functions as arguments and yielding functions as results. The standard example of passing a function as argument is the `map'` function, which applies a given function to every element of a list. It is defined as follows:

```
fun map' (f, nil) = nil
  | map' (f, h::t) = (f h) :: map' (f, t)
```

For example, the application

```
map' (fn x => x+1, [1,2,3,4])
```

evaluates to the list `[2,3,4,5]`.

Functions may also yield functions as results. What is surprising is that we can create *new* functions during execution, not just return functions that have been previously defined. The most basic (and deceptively simple) example is the function `constantly` that creates constant functions: given a value `k`, the application `constantly k` yields a function that yields `k` whenever it is applied. Here's a definition of `constantly`:

```
val constantly = fn k => (fn a => k)
```

The function constantly has type `'a -> ('b -> 'a)`. We used the `fn` notation for clarity, but the declaration of the function `constantly` may also be written using `fun` notation as follows:

```
fun constantly k a = k
```

Note well that a *white space* separates the two successive arguments to `constantly`! The meaning of this declaration is precisely the same as the earlier definition using `fn` notation.

The value of the application `constantly 3` is the function that is constantly `3`; *i.e.*, it always yields `3` when applied. Yet nowhere have we defined the function that always yields `3`. The resulting function is "created" by the application of `constantly` to the argument `3`, rather than merely "retrieved" off the shelf of previously-defined functions. In implementation terms the result of the application `constantly 3` is a closure consisting of the function `fn a => k` with the environment `val k = 3` attached to it. The closure is a data structure (a pair) that is created by each application of `constantly` to an argument; the closure is the representation of the "new" function yielded by the application. Notice, however, that the *only* difference between any two results of applying the function `constantly` lies in the attached environment; the underlying function is *always* `fn a => k`. If we think of the lambda as the "executable code" of the function, then this amounts to the observation that no new *code* is created at run-time, just new *instances* of existing code.

This also points out why functions in ML are not the same as code pointers in C. You may be familiar with the idea of passing a pointer to a C function to another C function as a means of passing functions as arguments or yielding functions as results. This may be considered to be a form of "higher-order" function in C, but it must be emphasized that code pointers are significantly less expressive than closures because in C there are only *statically many* possibilities for a code pointer (it must point to one of the functions defined in your code), whereas in ML we may generate *dynamically many* different instances of a function, differing in the bindings of the variables in its environment. The non-varying part of the closure, the code, is directly analogous to a function pointer in C, but there is no counterpart in C of the varying part of the closure, the dynamic environment.

The definition of the function `map'` given above takes a function and list as arguments, yielding a new list as result. Often it occurs that we wish to map the same function across several different lists. It is inconvenient (and a tad inefficient) to keep passing the same function to `map'`, with the list argument varying each time. Instead we would prefer to create a instance of map specialized to the given function that can then be applied to many different lists. This leads to the following definition of the function `map`:

```
fun map f nil = nil
  | map f (h::t) = (f h) :: (map f t)
```

The function `map` so defined has type `('a->'b) -> 'a list -> 'b list`. It takes a function of type `'a -> 'b` as argument, and yields another function of type `'a list -> 'b list` as result.

The passage from `map'` to `map` is called *currying*. We have changed a two-argument function (more properly, a function taking a pair as argument) into a function that takes two arguments in succession, yielding after the first a function that takes the second as its sole argument. This passage can be codified as follows:

```
fun curry f x y = f (x, y)
```

The type of `curry` is

```
('a*'b->'c) -> ('a -> ('b -> 'c)).
```

Given a two-argument function, `curry` returns another function that, when applied to the first argument, yields a function that, when applied to the second, applies the original two-argument function to the first and second arguments, given separately.

Observe that `map` may be alternately defined by the binding

```
fun map f l = curry map' f l
```

Applications are implicitly left-associated, so that this definition is equivalent to the more verbose declaration

```
fun map f l = ((curry map') f) l
```

11.4 Patterns of Control

We turn now to the idea of abstracting patterns of control. There is an obvious similarity between the following two functions, one to add up the numbers in a list, the other to multiply them.

```
fun add_up nil = 0
  | add_up (h::t) = h + add_up t
fun mul_up nil = 1
  | mul_up (h::t) = h * mul_up t
```

What precisely is the similarity? We will look at it from two points of view.

One view is that in each case we have a binary operation and a unit element for it. The result on the empty list is the unit element, and the result on a non-empty list is the operation applied to the head of the list and the result on the tail. This pattern can be abstracted as the function `reduce` defined as follows:

```
fun reduce (unit, opn, nil) =
    unit
  | reduce (unit, opn, h::t) =
    opn (h, reduce (unit, opn, t))
```

Here is the type of `reduce`:

```
val reduce : 'b * ('a*'b->'b) * 'a list -> 'b
```

The first argument is the unit element, the second is the operation, and the third is the list of values. Notice that the type of the operation admits the possibility of the first argument having a different type from the second argument and result.

Using reduce, we may re-define `add_up` and `mul_up` as follows:

```
fun add_up l = reduce (0, op +, l)
fun mul_up l = reduce (1, op *, l)
```

To further check your understanding, consider the following declaration:

```
fun mystery l = reduce (nil, op ::, l)
```

(Recall that "`op ::`" is the function of type `'a * 'a list -> 'a list` that adds a given value to the front of a list.) What function does `mystery` compute?

Another view of the commonality between `add_up` and `mul_up` is that they are both defined by induction on the structure of the list argument, with a base case for `nil`, and an inductive case for `h::t`, defined in terms of its behavior on `t`. But this is really just another way of saying that they are defined in terms of a unit element and a binary operation! The difference is one of perspective: whether we focus on the pattern part of the clauses (the inductive decomposition) or the result part of the clauses (the unit and operation). The recursive structure of `add_up` and `mul_up` is abstracted by the `reduce` functional, which is then specialized to yield `add_up` and `mul_up`. Said another way, *the function reduce abstracts the pattern of defining a function by induction on the structure of a list*.

The definition of `reduce` leaves something to be desired. One thing to notice is that the arguments `unit` and `opn` are carried unchanged through the recursion; only the list parameter changes on recursive calls. While this might seem like a minor overhead, it's important to remember that multi-argument functions are really single-argument functions that take a tuple as argument. This means that each time around the loop we are constructing a new tuple whose first and second components remain fixed, but whose third component varies. Is there a better way? Here's another definition that isolates the "inner loop" as an auxiliary, tail-recursive function:

```
fun better_reduce (unit, opn, l) =
    let
        fun red nil = unit
          | red (h::t) = opn (h, red t)
    in
        red l
    end
```

Notice that each call to `better_reduce` creates a *new* function `red` that uses the parameters `unit` and `opn` of the call to `better_reduce`. This means that `red` is bound to a closure consisting of the code for the function together with the environment active at the point of definition, which will provide bindings for `unit` and `opn` arising from the application of `better_reduce` to its arguments. Furthermore, the recursive calls to `red` no longer carry bindings for `unit` and `opn`, saving the overhead of creating tuples on each iteration of the loop.

11.5 Staging

An interesting variation on `reduce` may be obtained by *staging* the computation. The motivation is that `unit` and `opn` often remain fixed for many different lists (*e.g.*, we may wish to sum the elements of many different lists). In this case `unit` and `opn` are said to be "early" arguments and the list is said to be a "late" argument. The idea of staging is to perform as much computation as possible on the basis of the early arguments, yielding a function of the late arguments alone.

In the case of the function `reduce` this amounts to building `red` on the basis of `unit` and `opn`, yielding it as a function that may be later applied to many different lists. Here's the code:

```
fun staged_reduce (unit, opn) =
    let
        fun red nil = unit
          | red (h::t) = opn (h, red t)
    in
        red
    end
```

The definition of `staged_reduce` bears a close resemblance to the definition of `better_reduce`; the only difference is that the creation of the closure bound to `red` occurs *as soon as* `unit` *and* `opn` *are known*, rather than each time the list argument is supplied. Thus the overhead of closure creation is "factored out" of multiple applications of the resulting function to list arguments.

We could just as well have replaced the body of the let expression with the function

```
fn l => red l
```

but a moment's thought reveals that the meaning is the same.

Note well that we would *not* obtain the effect of staging were we to use the following definition:

```
fun curried_reduce (unit, opn) nil = unit
  | curried_reduce (unit, opn) (h::t) =
    opn (h, curried_reduce (unit, opn) t)
```

If we unravel the `fun` notation, we see that while we are taking two arguments in succession, we are *not* doing any useful work in between the arrival of the first argument (a pair) and the second (a list). A *curried* function does not take significant advantage of staging. Since `staged_reduce` and `curried_reduce` have the same iterated function type, namely

```
('b * ('a * 'b -> 'b)) -> 'a list -> 'b
```

the contrast between these two examples may be summarized by saying *not every function of iterated function type is curried*. Some are, and some aren't. The "interesting" examples (such as `staged_reduce`) are the ones that *aren't* curried. (This directly contradicts established terminology, but it is necessary to deviate from standard practice to avoid a serious misapprehension.)

The time saved by staging the computation in the definition of `staged_reduce` is admittedly minor. But consider the following definition of an append function for lists that takes both arguments at once:

```
fun append (nil, l) = l
  | append (h::t, l) = h :: append(t,l)
```

Suppose that we will have occasion to append many lists to the end of a given list. What we'd like is to build a specialized appender for the first list that, when applied to a second list, appends the second to the end of the first. Here's a naive solution that merely curries append:

```
fun curried_append nil l = l
  | curried_append (h::t) l = h :: append t l
```

Unfortunately this solution doesn't exploit the fact that the first argument is fixed for many second arguments. In particular, each application of the result of applying `curried_append` to a list results in the first list being traversed so that the second can be appended to it.

We can improve on this by staging the computation as follows:

```
fun staged_append nil = fn l => l
  | staged_append (h::t) =
    let
        val tail_appender = staged_append t
    in
        fn l => h :: tail_appender l
    end
```

Notice that the first list is traversed *once* for all applications to a second argument. When applied to a list `[`v_1`, ..., `v_n`]`, the function `staged_append` yields a function that is equivalent to, but not quite as efficient as, the function

`fn l =>` v_1 `::` v_2 `:: ... ::` v_n `:: l.`

This still takes time proportional to n, but a substantial savings accrues from avoiding the pattern matching required to destructure the original list argument on each call.

Chapter 12

Exceptions

In the first chapter of these notes we mentioned that expressions in Standard ML always have a type, may have a value, and may have an effect. So far we've concentrated on typing and evaluation. In this chapter we will introduce the concept of an *effect*. While it's hard to give a precise general definition of what we mean by an effect, the idea is that an effect is any action resulting from evaluation of an expression other than returning a value. From this point of view we might consider non-termination to be an effect, but we don't usually think of failure to terminate as a positive "action" in its own right, rather as a failure to take any action.

The main examples of effects in ML are these:

1. *Exceptions*. Evaluation may be aborted by signaling an exceptional condition.

2. *Mutation*. Storage may be allocated and modified during evaluation.

3. *Input/output*. It is possible to read from an input source and write to an output sink during evaluation.

4. *Communication*. Data may be sent to and received from communication channels.

This chapter is concerned with exceptions; the other forms of effects will be considered later.

12.1 Exceptions as Errors

ML is a *safe* language in the sense that its execution behavior may be understood entirely in terms of the constructs of the language itself. Behavior such as "dumping core" or incurring a "bus error" are extra-linguistic notions that may only be explained by appeal to the underlying implementation of the language. These cannot arise in ML. This is ensured by a combination of a static type discipline, which rules out expressions that are manifestly ill-defined (*e.g.*, adding a string to an integer or casting an integer as a function), and by dynamic checks that rule out violations that cannot be detected statically (*e.g.*, division by zero or arithmetic overflow). Static violations are signalled by type checking errors; dynamic violations are signalled by *raising exceptions*.

12.1.1 Primitve Exceptions

The expression `3 + "3"` is ill-typed, and hence cannot be evaluated. In contrast the expression `3 div 0` is well-typed (with type `int`), but incurs a run-time fault that is signalled by raising the exception `Div`. We will indicate this by writing

$$\texttt{3 div 0} \Downarrow \texttt{raise Div}$$

An exception is a form of "answer" to the question "what is the value this expression?". In most implementations an exception such as this is reported by an error message of the form "`Uncaught exception Div`", together with the line number (or some other indication) of the point in the program where the exception occurred.

Exceptions have names so that we may distinguish different sources of error in a program. For example, evaluation of the expression `maxint * maxint` (where `maxint` is the largest representable integer) causes the exception `Overflow` to be raised, indicating that an arithmetic overflow error arose in the attempt to carry out the multiplication. This is usefully distinguished from the exception `Div`, corresponding to division by zero.

(You may be wondering about the overhead of checking for arithmetic faults. The compiler must generate instructions that ensure that an overflow fault is caught before any subsequent operations are performed. This can be quite expensive on pipelined processors, which sacrifice precise delivery of arithmetic faults in the interest of speeding up execution in the non-faulting case. Unfortunately it is necessary to incur this overhead if we are to avoid having the behavior of an ML program depend on the underlying processor on which it is implemented.)

Another source of run-time exceptions is an inexhaustive match. Suppose we define the function `hd` as follows

```
fun hd (h::_) = h
```

This definition is inexhaustive since it makes no provision for the possibility of the argument being `nil`. What happens if we apply `hd` to `nil`? The exception `Match` is raised, indicating the failure of the pattern-matching process:

hd nil $\Downarrow$ raise Match

The occurrence of a `Match` exception at run-time is indicative of a violation of a pre-condition to the invocation of a function somewhere in the program. Recall that it is often sensible for a function to be inexhaustive, provided that we take care to ensure that it is never applied to a value outside of its domain. Should this occur (because of a programming mistake, evidently), the result is nevertheless well-defined because ML checks for, and signals, pattern match failure. That is, ML programs are implicitly "bullet-proofed" against failures of pattern matching. The flip side is that if no inexhaustive match warnings arise during type checking, then the exception Match can never be raised during evaluation (and hence no run-time checking need be performed).

A related situation is the use of a pattern in a `val` binding to destructure a value. If the pattern can fail to match a value of this type, then a `Bind` exception is raised at run-time. For example, evaluation of the binding

```
val h::_ = nil
```

raises the exception `Bind` since the pattern `h::_` does not match the value `nil`. Here again observe that a `Bind` exception cannot arise unless the compiler has previously warned us of the possibility: no warning, no `Bind` exception.

12.1.2 User-Defined Exceptions

So far we have considered examples of pre-defined exceptions that indicate fatal error conditions. Since the built-in exceptions have a built-in meaning, it is generally inadvisable to use these to signal program-specific error conditions. Instead we introduce a *new* exception using an `exception` declaration, and signal it using a `raise` expression when a run-time violation occurs. That way we can associate specific exceptions with specific pieces of code, easing the process of tracking down the source of the error.

Suppose that we wish to define a "checked factorial" function that ensures that its argument is non-negative. Here's a first attempt at defining such a function:

```
exception Factorial

fun checked_factorial n =
    if n < 0 then
       raise Factorial
    else if n=0 then
       1
    else n * checked_factorial (n-1)
```

The declaration `exception Factorial` introduces an exception `Factorial`, which we raise in the case that `checked_factorial` is applied to a negative number.

The definition of `checked_factorial` is unsatisfactory in at least two respects. One, relatively minor, issue is that it does not make effective use of pattern matching, but instead relies on explicit comparison operations. To some extent this is unavoidable since we wish to check explicitly for negative arguments, which cannot be done using a pattern. A more significant problem is that `checked_factorial` *repeatedly* checks the validity of its argument on each recursive call, even though we can prove that if the initial argument is non-negative, then so must be the argument on each recursive call. This fact is not reflected in the code. We can improve the definition by introducing an auxiliary function:

```
exception Factorial

local
   fun fact 0 = 1
     | fact n = n * fact (n-1)
in
   fun checked_factorial n =
       if n >= 0 then
          fact n
       else
          raise Factorial
end
```

Notice that we perform the range check exactly once, and that the auxiliary function makes effective use of pattern-matching.

12.2 Exception Handlers

The use of exceptions to signal error conditions suggests that raising an exception is fatal: execution of the program terminates with the raised exception. But signaling an error is only one use of the exception mechanism. More generally, exceptions can be used to effect *non-local transfers of control*. By using an *exception handler* we may "catch" a raised exception and continue evaluation along some other path. A very simple example is provided by the following driver for the factorial function that accepts numbers from the keyboard, computes their factorial, and prints the result.

```
fun factorial_driver () =
    let
        val input = read_integer ()
        val result =
            toString (checked_factorial input)
    in
        print result
    end
    handle Factorial => print "Out of range."
```

An expression of the form *exp* `handle` *match* is called an *exception handler*. It is evaluated by attempting to evaluate *exp*. If it returns a value, then that is the value of the entire expression; the handler plays no role in this case. If, however, *exp* raises an exception *exc*, then the exception value is matched against the clauses of the match (exactly as in the application of a clausal function to an argument) to determine how to proceed. If the pattern of a clause matches the exception *exc*, then evaluation resumes with the expression part of that clause. If no pattern matches, the exception *exc* is *re-raised* so that outer exception handlers may dispatch on it. If no handler handles the exception, then the uncaught exception is signaled as the final result of evaluation. That is, computation is aborted with the uncaught exception *exc*.

In more operational terms, evaluation of *exp* `handle` *match* proceeds by installing an exception handler determined by *match*, then evaluating *exp*. The previous binding of the exception handler is preserved so that it may be restored once the given handler is no longer needed. Raising an exception consists of passing a value of type `exn` to the current exception handler. Passing an exception to a handler de-installs that handler, and re-installs the previously active handler. This ensures that if the handler itself raises an exception, or fails to handle the given exception, then the exception is propagated to the handler active prior to evaluation of the `handle` expression. If the expression does not raise an exception, the previous handler is restored as part of completing the evaluation of the `handle` expression.

Returning to the function `factorial_driver`, we see that evaluation proceeds by attempting to compute the factorial of a given number (read from the keyboard by an unspecified function `read_integer`), printing the result if the given number is in range, and otherwise reporting that the number is out of range. The example is trivialized to focus on the role of exceptions, but one could easily imagine generalizing it in a number of ways that also make use of exceptions. For example, we might repeatedly read integers until the user terminates the input stream (by typing the end of file character). Termination of input might be signaled by an `EndOfFile` exception, which is handled by the driver. Similarly, we might expect that the function `read_integer` raises the exception `SyntaxError` in the case that the input is not properly formatted. Again we would handle this exception, print a suitable message, and resume.

Here's a sketch of a more complicated factorial driver:

```
fun factorial_driver () =
    let
        val input = read_integer ()
        val result =
            toString (checked_factorial input)
        val _ = print result
    in
        factorial_driver ()
    end
    handle EndOfFile => print "Done."
         | SyntaxError =>
           let
               val _ = print "Syntax error."
           in
               factorial_driver ()
           end
         | Factorial =>
           let
               val _ = print "Out of range."
           in
               factorial_driver ()
           end
```

We will return to a more detailed discussion of input/output later in these notes. The point to notice here is that the code is structured with a completely uncluttered "normal path" that reads an integer, computes its factorial, formats it, prints it, and repeats. The exception handler takes care of the exceptional cases: end of file, syntax error, and domain error. In the latter two cases we report an error, and resume reading. In the former we simply report completion and we are done.

The reader is encouraged to imagine how one might structure this program *without* the use of exceptions. The primary benefits of the exception mechanism are as follows:

1. They *force* you to consider the exceptional case (if you don't, you'll get an uncaught exception at run-time), and

2. They *allow* you to segregate the special case from the normal case in the code (rather than clutter the code with explicit checks).

These aspects work hand-in-hand to facilitate writing robust programs.

A typical use of exceptions is to implement *backtracking*, a programming technique based on exhaustive search of a state space. A very simple, if somewhat artificial, example is provided by the following function to compute change from an arbitrary list of coin values. What is at issue is that the obvious "greedy" algorithm for making change that proceeds by doling out as many coins as possible in decreasing order of value does not always work. Given only a 5 cent and a 2 cent coin, we cannot make 16 cents in change by first taking three 5's and then proceeding to dole out 2's. In fact we must use two 5's and three 2's to make 16 cents. Here's a method that works for any set of coins:

```
exception Change

fun change _ 0 = nil
  | change nil _ = raise Change
  | change (coin::coins) amt =
    if coin > amt then
       change coins amt
    else
       (coin :: change (coin::coins) (amt-coin))
       handle Change => change coins amt
```

The idea is to proceed greedily, but if we get "stuck", we undo the most recent greedy decision and proceed again from there. Simulate evaluation of the example of `change [5,2] 16` to see how the code works.

12.3 Value-Carrying Exceptions

So far exceptions are just "signals" that indicate that an exceptional condition has arisen. Often it is useful to attach additional information that is passed to the exception handler. This is achieved by attaching values to exceptions.

For example, we might associate with a `SyntaxError` exception a string indicating the precise nature of the error. In a parser for a language we might write something like

```
raise SyntaxError "Integer expected"
```

to indicate a malformed expression in a situation where an integer is expected, and write

```
raise SyntaxError "Identifier expected"
```

to indicate a badly-formed identifier.

To associate a string with the exception `SyntaxError`, we declare it as

```
exception SyntaxError of string.
```

This declaration introduces the exception `SyntaxError` as an exception carrying a string as value. This declaration introduces the *exception constructor* `SyntaxError`.

Exception constructors are in many ways similar to value constructors. In particular they can be used in patterns, as in the following code fragment:

```
... handle SyntaxError msg => print "Syntax error: " ^ msg
```

Here we specify a pattern for `SyntaxError` exceptions that also binds the string associated with the exception to the identifier `msg` and prints that string along with an error indication.

Recall that we may use value constructors in two ways:

1. We may use them to create values of a datatype (perhaps by applying them to other values).

2. We may use them to match values of a datatype (perhaps also matching a constituent value).

The situation with exception constructors is symmetric.

1. We may use them to create an exception (perhaps with an associated value).

2. We may use them to match an exception (perhaps also matching the associated value).

Value constructors have types, as we previously mentioned. For example, the list constructors `nil` and `::` have types `'a list` and `'a * 'a list -> 'a list`, respectively. What about exception constructors? A "bare" exception constructor (such as `Factorial` above) has type `exn` and a value-carrying exception constructor (such as `SyntaxError`) has type `string -> exn` Thus `Factorial` is a value of type `exn`, and

```
SyntaxError "Integer expected"
```

is a value of type `exn`.

The type `exn` is the type of *exception packets*, the data values associated with an exception. The primitive operation `raise` takes any value of type `exn` as argument and raises an exception with that value. The clauses of a handler may be applied to any value of type `exn` using the rules of pattern matching described earlier; if an exception constructor is no longer in scope, then the handler cannot catch it (other than via a wild-card pattern).

The type `exn` may be thought of as a kind of built-in datatype, *except that* the constructors of this type are not determined once and for all (as they are with a `datatype` declaration), but rather are *incrementally* introduced as needed in a program. For this reason the type `exn` is sometimes called an *extensible datatype.*

Chapter 13

Mutable Storage

In this chapter we consider a second form of effect, called a *storage effect*, the allocation or mutation of storage during evaluation. The introduction of storage effects has profound consequences, not all of which are desirable. Indeed, they are sometimes called *side effects*, by analogy with the unintended actions of some medications. While it is excessive to dismiss storage effects as completely undesirable, it is advantageous to minimize the use of storage effects except in situations where the task clearly demands them. We will explore some techniques for programming with storage effects later in this chapter, but first we introduce the primitive mechanisms for programming with mutable storage in ML.

13.1 Reference Cells

To support mutable storage the execution model that we described in chapter 2 is enriched with a *memory* consisting of a finite set of *mutable cells*. A mutable cell may be thought of as a container in which a data value of a specified type is stored. During execution of a program the contents of a cell may be retrieved or replaced by any other value of the appropriate type. Since cells are used by issuing "commands" to modify and retrieve their contents, programming with cells is called *imperative programming*.

Changing the contents of a mutable cell introduces a temporal aspect to evaluation. We speak of the *current* contents of a cell, meaning the value *most recently* assigned to it. We also speak of *previous* and *future* values of a reference cell when discussing the behavior of a program. This is in sharp contrast to the effect-free fragment of ML, for which no such concepts apply. For example, the binding of a variable does not change while evaluating within the scope of that variable, lending a "permanent" quality to statements about variables — the "current" binding is the only binding that variable will ever have.

The type *typ* `ref` is the type of reference cells containing values of type `typ`. Reference cells are, like all values, first class — they may be bound to variables, passed as arguments to functions, returned as results of functions, appear within data structures, and even be stored within other reference cells.

A reference cell is *created*, or *allocated*, by the function `ref` of type *typ* `->` *typ* `ref`. When applied to a value *val* of type *typ*, `ref` allocates a "new" cell, initializes its content to *val*, and returns a reference to the cell. By "new" we mean that the allocated cell is distinct from all other cells previously allocated, and does not share storage with them.

The *contents* of a cell of type *typ* is retrieved using the function `!` of type *typ* `ref ->` *typ*. Applying `!` to a reference cell yields the current contents of that cell. The contents of a cell is changed by applying the assignment operator `op :=`, which has type *typ* `ref *` *typ* `-> unit`. Assignment is usually written using infix syntax. When applied to a cell and a value, it replaces the content of that cell with that value, and yields the null-tuple as result.

Here are some examples:

```
val r = ref 0
val s = ref 0
val _ = r := 3
val x = !s + !r
val _ = t := 5
val y = !s + !r
val z = !t + !r
```

After execution of these bindings, the variable `x` is bound to `3`, the variable `y` is bound to `5`, and `z` is bound to `10`.

Notice the use of a `val` binding of the form `val _ =` *exp* when *exp* is to be evaluated purely for its effect. The value of *exp* is discarded by the binding, since the left-hand side is a wildcard pattern. In most cases the expression *exp* has type `unit`, so that its value is guaranteed to be the null-tuple, `()`, if it has a value at all.

A wildcard binding is used to define *sequential composition* of expressions in ML. The expression

exp_1 `;` exp_2

is shorthand for the expression

```
let
    val _ = exp1
in
    exp2
end
```

that first evaluates exp_1 for its effect, then evaluates exp_2.

Functions of type *typ*`->unit` are sometimes called *procedures*, because they are executed purely for their effect. This is apparent from the type: it is assured that the value of applying such a function is the null-tuple, `()`, so the only point of applying it is for its effects on memory.

13.2 Reference Patterns

It is a common mistake to omit the exclamation point when referring to the content of a reference, especially when that cell is bound to a variable. In more familiar languages such as C all variables are implicitly bound to reference cells, and they are implicitly *de-referenced* whenever they are used so that a variable always stands for its current contents. This is both a boon and a bane. It is obviously helpful in many common cases since it alleviates the burden of having to explicitly dereference variables whenever their content is required. However, it shifts the burden to the programmer in the case that the address, and not the content, is intended. In C one writes `& x` for the address of (the cell bound to) `x`. Whether explicit or implicit de-referencing is preferable is to a large extent a matter of taste. The burden of explicit de-referencing is not nearly so onerous in ML as it might be in other languages simply because reference cells are used so infrequently in ML programs, whereas they are the sole means of binding variables in more familiar languages.

An alternative to explicitly de-referencing cells is to use *ref patterns*. A pattern of the form `ref` *pat* matches a reference cell whose content matches the pattern *pat*. This means that the cell's contents are implicitly retrieved during pattern matching, and may be subsequently used without explicit de-referencing. For example, the second implementation of `rot3` above might be written using ref patterns as follows:

```
fun rot3 (a, b, c) =
    let
        val (ref x, ref y, ref z) = (a, b, c)
    in
        a := y; b := z; c := x
    end
```

In practice it is common to use both explicit de-referencing and ref patterns, depending on the situation.

13.3 Identity

Reference cells raise delicate issues of equality that considerably complicate reasoning about programs. In general we say that two expressions (of the same type) are equal iff they cannot be distinguished by any operation in the language. That is, two expressions are distinct iff there is some way within the language to tell them apart. This is called *Leibniz's Principle* of *identity of indiscernables* — we equate everything that we cannot tell apart. As a consequence we obtain the principle of *indiscernability of identicals* — that which we deem equal cannot be told apart. (Think about this for a minute. One can have the latter without the former. Leibniz's Principle is a principle of maximality for equality.)

What makes Leibniz's Principle tricky to grasp is that it hinges on what we mean by a "way to tell expressions apart". The crucial idea is that we can tell two expressions apart iff there is a *complete program* containing one of the expressions whose *observable behavior* changes when we replace that expression by the other. That is, two expressions are considered equal iff there is no such scenario that distinguishes them. But what do we mean by "complete program"? And what do we mean by "observable behavior"?

For the present purposes we will consider a complete program to be any expression of basic type (say, `int` or `bool` or `string`). The idea is that a complete program is one that computes a concrete result such as a number. The observable behavior of a complete program includes at least these aspects:

1. Its value, or lack thereof, either by non-termination or by raising an uncaught exception.

2. Its visible side effects, include visible modifications to mutable storage or any input/output it may perform.

In contrast here are some behaviors that we will not count as observations:

1. Execution time or space usage.
2. "Private" uses of storage (*e.g.*, internally-allocated reference cells).
3. The name of uncaught exceptions (*i.e.*, we will not distinguish between terminating with the uncaught exception `Bind` and the uncaught exception `Match`.

With these ideas in mind, it should be plausible that if we evaluate these bindings

```
val r = ref 0
val s = ref 0
```

then `r` and `s` are *not* equivalent. Consider the following usage of `r` to compute an integer result:

```
(s := 1 ; !r)
```

Clearly this expression evaluates to `0`, and mutates the binding of `s`. Now replace `r` by `s` to obtain

```
(s := 1 ; !s)
```

This expression evaluates to `1`, and mutates `s` as before. These two complete programs distinguish `r` from `s`, and therefore must be considered distinct.

Had we replaced the binding for `s` by the binding

```
val s = r
```

then the two expressions that formerly distinguished `r` from `s` no long do so — they are, after all, bound to the *same* reference cell! In fact, no program can be concocted that would distinguish them. In this case `r` and `s` are equivalent.

Now consider a third, very similar scenario. Let us declare `r` and `s` as follows:

```
val r = ref ()
val s = ref ()
```

Are r and s equivalent or not? We might first try to distinguish them by a variant of the experiment considered above. This breaks down because there is only one possible value we can assign to a variable of type `unit ref`! Indeed, one may suspect that r and s are equivalent in this case, but in fact there is a way to distinguish them! Here's a complete program involving r that we will use to distinguish r from s:

```
if r=r then "it's r" else "it's not"
```

Now replace the first occurrence of r by s to obtain

```
if s=r then "it's r" else "it's not"
```

and the result is different.

This example hinges on the fact that ML defines equality for values of reference type to be *reference equality* (or, occasionally, *pointer equality*). Two reference cells (of the same type) are equal in this sense iff they both arise from the exact same use of the `ref` operation to allocate that cell; otherwise they are distinct. Thus the two cells bound to r and s above are observably distinct (by testing reference equality), even though they can only ever hold the value `()`. Had equality not been included as a primitive, any two reference cells of `unit` type would have been equal.

Why does ML provide such a fine-grained notion of equality? "True" equality, as defined by Leibniz's Principle, is, unfortunately, undecidable — there is no computer program that determines whether two expressions are equivalent in this sense. ML provides a useful, conservative approximation to true equality that in some cases is not defined (you cannot test two functions for equality) and in other cases is too picky (it distinguishes reference cells that are otherwise indistinguishable). Such is life.

13.4 Aliasing

To see how reference cells complicate programming, let us consider the problem of *aliasing*. Any two variables of the same reference type might be bound to the *same* reference cell, or to two different reference cells. For example, after the declarations

```
val r = ref 0
val s = ref 0
```

the variables r and s are not aliases, but after the declaration

```
val r = ref 0
val s = r
```

the variables `r` and `s` are aliases for the same reference cell.

These examples show that we must be careful when programming with variables of reference type. This is particularly problematic in the case of functions, because we cannot assume that two different argument variables are bound to different reference cells. They might, in fact, be bound to the same reference cell, in which case we say that the two variables are *aliases* for one another. For example, in a function of the form

```
fn (x:typ ref, y:typ ref) => exp
```

we may not assume that `x` and `y` are bound to different reference cells. We must always ask ourselves whether we've properly considered aliasing when writing such a function. This is harder to do than it sounds. Aliasing is a huge source of bugs in programs that work with reference cells.

A classic example is a program to "rotate" the contents of three cells: given reference cells a, b, and c, with initial contents x, y, and z, set their contents to y, z, and x, respectively. Here's a candidate implementation:

```
fun rot3 (a, b, c) =
    let
        val t = !a
    in
        a := !b; b := !c; c := t
    end
```

This code works fine if a, b, and c are three distinct reference cells. But suppose that a and c are the same cell. Afterwards the contents of a, b, and c are y, y, and x! Here's a solution that works correctly, even in the presence of aliasing:

```
fun rot3 (a, b, c) =
    let
        val (x, y, z) = (!a, !b, !c)
    in
        a := y; b := z; c := x
    end
```

Notice that we use ordinary *immutable* variables to temporarily hold the initial contents of the cells while their values are being updated.

13.5 Programming Well With References

Using references it is possible to mimic the style of programming used in imperative languages such as C. For example, we might define the factorial function in imitation of such languages as follows:

```
fun imperative_fact (n:int) =
    let
        val result = ref 1
        val i = ref 0
        fun loop () =
            if !i = n then
               ()
            else
               (i := !i + 1;
                result := !result * !i;
                loop ())
    in
        loop (); !result
    end
```

Notice that the function `loop` is essentially just a while loop; it repeatedly executes its body until the contents of the cell bound to `i` reaches `n`. The tail call to `loop` is essentially just a `goto` statement to the top of the loop.

It is (appallingly) bad style to program in this fashion. The purpose of the function `imperative_fact` is to compute a simple function on the natural numbers. There is nothing about its definition that suggests that state must be maintained, and so it is senseless to allocate and modify storage to compute it. The definition we gave earlier is shorter, simpler, more efficient, and hence more suitable to the task. This is not to suggest, however, that there are no good uses of references. We will now discuss some important uses of state in ML.

13.5.1 Private Storage

The first example is the use of higher-order functions to manage shared private state. This programming style is closely related to the use of objects to manage state in object-oriented programming languages. Here's an example to frame the discussion:

```
local
   val counter = ref 0
in
   fun tick () = (counter := !counter + 1; !counter)
   fun reset () = (counter := 0)
end
```

This declaration introduces two functions, `tick` of type `unit -> int` and `reset` of type `unit -> unit`. Their definitions share a *private* variable `counter` that is bound to a mutable cell containing the current value of a shared counter. The `tick` operation increments the counter and returns its new value, and the `reset` operation resets its value to zero. The types of the operations suggest that implicit state is involved. In the absence of exceptions and implicit state, there is only one useful function of type `unit->unit`, namely the function that always returns its argument (and it's debatable whether this is really useful!).

The declaration above defines two functions, `tick` and `reset`, that share a single private counter. Suppose now that we wish to have several different *instances* of a counter — different pairs of functions `tick` and `reset` that share different state. We can achieve this by defining a *counter generator* (or *constructor*) as follows:

```
fun new_counter () =
    let
        val counter = ref 0
        fun tick () = (counter := !counter + 1; !counter)
        fun reset () = (counter := 0)
    in
        { tick = tick, reset = reset }
    end
```

The type of `new_counter` is

```
unit -> { tick : unit->int, reset : unit->unit }.
```

We've packaged the two operations into a record containing two functions that share private state. There is an obvious analogy with class-based object-oriented programming. The function `new_counter` may be thought of as a *constructor* for a class of counter *objects*. Each object has a private *instance variable* `counter` that is shared between the *methods* `tick` and `reset` of the object represented as a record with two fields.

Here's how we use counters.

```
val c1 = new_counter ()
val c2 = new_counter ()
#tick c1 ();
(* 1 *)
#tick c1 ();
(* 2 *)
#tick c2 ();
(* 1 *)
#reset c1 ();
#tick c1 ();
(* 1 *)
#tick c2 ();
(* 2 *)
```

Notice that `c1` and `c2` are *distinct* counters that increment and reset independently of one another.

13.5.2 Mutable Data Structures

A second important use of references is to build *mutable data structures*. The data structures (such as lists and trees) we've considered so far are *immutable* in the sense that it is impossible to *change* the structure of the list or tree without building a modified copy of that structure. This is both a benefit and a drawback. The principal benefit is that immutable data structures are *persistent* in that operations performed on them do not destroy the original structure — in ML we can eat our cake and have it too. For example, we can simultaneously maintain a dictionary both before and after insertion of a given word. The principal drawback is that if we aren't really relying on persistence, then it is wasteful to make a copy of a structure if the original is going to be discarded anyway. What we'd like in this case is to have an "update in place" operation to build an *ephemeral* (opposite of persistent) data structure. To do this in ML we make use of references.

A simple example is the type of *possibly circular lists*, or *pcl's*. Informally, a pcl is a finite graph in which every node has at most one neighbor, called its *predecessor*, in the graph. In contrast to ordinary lists the predecessor relation is not necessarily well-founded: there may be an infinite sequence of nodes arranged in descending order of predecession. Since the graph is finite, this can only happen if there is a cycle in the graph: some node has an ancestor as predecessor. How can such a structure ever come into existence? If the predecessors of a cell are needed to construct a cell, then the ancestor that is to serve as predecessor in the cyclic case can never be created! The "trick" is to employ *backpatching*: the predecessor is initialized to `Nil`, so that the node and its ancestors can be constructed, then it is reset to the appropriate ancestor to create the cycle.

This can be achieved in ML using the following `datatype` declaration:

```
datatype 'a pcl = Pcl of 'a pcell ref
and 'a pcell = Nil | Cons of 'a * 'a pcl;
```

A value of type *typ* `pcl` is essentially a reference to a value of type *typ* `pcell`. A value of type *typ* `pcell` is either `Nil`, the cell at the end of a non-circular possibly-circular list, or `Cons (`*h*`, `*t*`)`, where *h* is a value of type *typ* and `t` is another such possibly-circular list.

Here are some convenient functions for creating and taking apart possibly-circular lists:

```
fun cons (h, t) = Pcl (ref (Cons (h, t)));
fun nill () = Pcl (ref Nil);
fun phd (Pcl (ref (Cons (h, _)))) = h;
fun ptl (Pcl (ref (Cons (_, t)))) = t;
```

To implement backpatching, we need a way to "zap" the tail of a possibly-circular list.

```
fun stl (Pcl (r as ref (Cons (h, _))), u) =
    (r := Cons (h, u));
```

If you'd like, it would make sense to require that the tail of the `Cons` cell be the empty pcl, so that you're only allowed to backpatch at the end of a finite pcl.

Here is a finite and an infinite pcl.

```
val finite = cons (4, cons (3, cons (2, cons (1, nill ()))))
val tail = cons (1, nill());
val infinite = cons (4, cons (3, cons (2, tail)));
val _ = stl (tail, infinite)
```

The last step backpatches the tail of the last cell of `infinite` to be `infinite` itself, creating a circular list.

Now let us define the size of a pcl to be the number of distinct nodes occurring in it. It is an interesting problem is to define a `size` function for pcls that makes *no* use of auxiliary storage (*e.g.,* no set of previously-encountered nodes) and runs in time proportional to the number of cells in the pcl. The idea is to think of running a long race between a tortoise and a hare. If the course is circular, then the hare, which quickly runs out ahead of the tortoise, will eventually come from behind and pass it! Conversely, if this happens, the course must be circular.

```
local
  fun race (Nil, Nil) = 0
    | race (Cons (_, Pcl (ref c)), Nil) =
      1 + race (c, Nil)
    | race (Cons (_, Pcl (ref c)), Cons (_, Pcl (ref Nil))) =
      1 + race (c, Nil)
    | race (Cons (_, l), Cons (_, Pcl (ref (Cons (_, m))))) =
      1 + race' (l, m)
  and race' (Pcl (r as ref c), Pcl (s as ref d)) =
      if r=s then 0 else race (c, d)
in
  fun size (Pcl (ref c)) = race (c, c)
end
```

The hare runs twice as fast as the tortoise. We let the tortoise do the counting; the hare's job is simply to detect cycles. If the hare reaches the finish line, it simply waits for the tortoise to finish counting. This covers the first three clauses of `race`. If the hare has not yet finished, we must continue with the hare running at twice the pace, checking whether the hare catches the tortoise from behind. Notice that it can never arise that the tortoise reaches the end before the hare does! Consequently, the definition of `race` is inexhaustive.

13.6 Mutable Arrays

In addition to reference cells, ML also provides mutable arrays as a primitive data structure. The type *typ* `array` is the type of arrays carrying values of type *typ*. The basic operations on arrays are these:

```
val array : int * 'a -> 'a array
val size : 'a array -> int
val sub : 'a array * int -> 'a
val update : 'a array * int * 'a -> unit
```

The function `array` creates a new array of a given size, with the given value as the initial value of every element of the array. The function `size` returns the size of an array. The function `sub` performs a subscript operation, returning the ith element of an array `A`, where $0 \leq i \leq$ `size(A)`. (These are just the basic operations on arrays; please see V for complete information.)

One simple use of arrays is for *memoization*. Here's a function to compute the nth Catalan number, which may be thought of as the number of distinct ways to parenthesize an arithmetic expression consisting of a sequence of n consecutive multiplication's. It makes use of an auxiliary summation function that you can easily define for yourself. (Applying `sum` to f and n computes the sum of $f\,0 + \cdots + f\,n$.)

```
fun C 1 = 1
  | C n = sum (fn k => (C k) * (C (n-k))) (n-1)
```

This definition of `C` is hugely inefficient because a given computation may be repeated exponentially many times. For example, to compute `C 10` we must compute `C 1`, `C 2`, ..., `C 9`, and the computation of `C` i engenders the computation of `C 1`, ..., `C` $i-1$ for each $1 \leq i \leq 9$. We can do better by caching previously-computed results in an array, leading to an enormous improvement in execution speed. Here's the code:

```
local
       val limit : int = 100
       val memopad : int option array =
           Array.array (limit, NONE)
in
       fun C' 1 = 1
         | C' n = sum (fn k => (C k)*(C (n-k))) (n-1)
       and C n =
           if n < limit then
              case Array.sub (memopad, n)
                of SOME r => r
                 | NONE =>
                   let
                       val r = C' n
                   in
                       Array.update (memopad, n, SOME r);
                       r
                   end
           else
              C' n
```

```
end
```

Note carefully the structure of the solution. The function C is a memoized version of the Catalan number function. When called it consults the memopad to determine whether or not the required result has already been computed. If so, the answer is simply retrieved from the memopad, otherwise the result is computed, stored in the cache, and returned. The function C′ looks superficially similar to the earlier definition of C, with the important difference that the recursive calls are to C, rather than C′ itself. This ensures that sub-computations are properly cached and that the cache is consulted whenever possible.

The main weakness of this solution is that we must fix an upper bound on the size of the cache. This can be alleviated by implementing a more sophisticated cache management scheme that dynamically adjusts the size of the cache based on the calls made to it.

Chapter 14

Input/Output

The Standard ML Basis Library (described in V) defines a three-layer input and output facility for Standard ML. These modules provide a rudimentary, platform-independent text I/O facility that we summarize briefly here. The reader is referred to V for more details. Unfortunately, there is at present no standard library for graphical user interfaces; each implementation provides its own package. See your compiler's documentation for details.

14.1 Textual Input/Output

The text I/O primitives are based on the notions of an *input stream* and an *output stream*, which are values of type `instream` and `outstream`, respectively. An input stream is an unbounded sequence of characters arising from some source. The source could be a disk file, an interactive user, or another program (to name a few choices). Any source of characters can be attached to an input stream. An input stream may be thought of as a buffer containing zero or more characters that have already been read from the source, together with a means of requesting more input from the source should the program require it. Similarly, an output stream is an unbounded sequence of characters leading to some sink. The sink could be a disk file, an interactive user, or another program (to name a few choices). Any sink for characters can be attached to an output stream. An output stream may be thought of as a buffer containing zero or more characters that have been produced by the program but have yet to be flushed to the sink.

Each program comes with one input stream and one output stream, called `stdIn` and `stdOut`, respectively. These are ordinarily connected to the user's keyboard and screen, and are used for performing simple text I/O in a program. The output stream `stdErr` is also pre-defined, and is used for error reporting. It is ordinarily connected to the user's screen.

Textual input and output are performed on streams using a variety of primitives. The simplest are `inputLine` and `print`. To read a line of input from a stream, use the function `inputLine` of type `instream -> string`. It reads a line of input from the given stream and yields that line as a string whose last character is the line terminator. If the source is exhausted, return the empty string. To write a line to `stdOut`, use the function `print` of type `string -> unit`. To write to a specific stream, use the function `output` of type `outstream * string -> unit`, which writes the given string to the specified output stream. For interactive applications it is often important to ensure that the output stream is flushed to the sink (*e.g.*, so that it is displayed on the screen). This is achieved by calling `flushOut` of type `outstream -> unit`, which ensures that the output stream is flushed to the sink. The print function is a composition of `output` (to `stdOut`) and `flushOut`.

A new input stream may be created by calling the function `openIn` of type `string -> instream`. When applied to a string, the system attempts to open a file with that name (according to operating system-specific naming conventions) and attaches it as a source to a new input stream. Similarly, a new output stream may be created by calling the function `openOut` of type `string -> outstream`. When applied to a string, the system attempts to create a file with that name (according to operating system-specific naming conventions) and attaches it as a sink for a new output stream. An input stream may be closed using the function `closeIn` of type `instream -> unit`. A closed input stream behaves as if there is no further input available; request for input from a closed input stream yield the empty string. An output stream may be closed using `closeOut` of type `outstream -> unit`. A closed output stream is unavailable for further output; an attempt to write to a closed output stream raises the exception `TextIO.IO`.

The function `input` of type `instream -> string` is a blocking read operation that returns a string consisting of the characters currently available from the source. If none are currently available, but the end of source has not been reached, then the operation blocks until at least one character is available from the source. If the source is exhausted or the input stream is closed, `input` returns the null string. To test whether an `input` operation would block, use the function `canInput` of type `instream * int -> int option`. Given a stream s and a bound n, the function `canInput` determines whether or not a call to `input` on s would immediately yield up to n characters. If the `input` operation would block, `canInput` yields `NONE`; otherwise it yields `SOME` k, with $0 \leq k \leq n$ being the number of characters immediately available on the input stream. If `canInput` yields `SOME 0`, the stream is either closed or exhausted. The function `endOfStream` of type `instream -> bool` tests whether the input stream is currently at the end (no further input is available from the source). This condition is transitive since, for example, another process might append data to an open file in between calls to `endOfStream`.

The function `output` of type `outstream * string -> unit` writes a string to an output stream. It may block until the sink is able to accept the entire string. The function `flushOut` of type `outstream -> unit` forces any pending output to the sink, blocking until the sink accepts the remaining buffered output.

This collection of primitive I/O operations is sufficient for performing rudimentary textual I/O. For further information on textual I/O, and support for binary I/O and Posix I/O primitives, see the Standard ML Basis Library.

Chapter 15

Lazy Data Structures

In ML all variables are bound *by value*, which means that the bindings of variables are *fully evaluated* expressions, or *values*. This general principle has several consequences:

1. The right-hand side of a `val` binding is evaluated before the binding is effected. If the right-hand side has no value, the `val` binding does not take effect.

2. In a function application the argument is evaluated before being passed to the function by binding that value to the parameter of the function. If the argument does not have a value, then neither does the application.

3. The arguments to value constructors are evaluated before the constructed value is created.

According to the by-value discipline, the bindings of variables are evaluated, regardless of whether that variable is ever needed to complete execution. For example, to compute the result of applying the function `fn x => 1` to an argument, we never actually need to evaluate the argument, but we do anyway. For this reason ML is sometimes said to be an *eager* language.

An alternative is to bind variables *by name*,[1] which means that the binding of a variable is an *unevaluated* expression, known as a *computation* or a *suspension* or a *thunk*.[2] This principle has several consequences:

[1]The terminology is historical, and not well-motivated. It is, however, firmly established.
[2]For reasons that are lost in the mists of time.

1. The right-hand side of a `val` binding is not evaluated before the binding is effected. The variable is bound to a computation (unevaluated expression), not a value.

2. In a function application the argument is passed to the function in unevaluated form by binding it directly to the parameter of the function. This holds regardless of whether the argument has a value or not.

3. The arguments to value constructor are left unevaluated when the constructed value is created.

According to the by-name discipline, the bindings of variables are only evaluated (if ever) when their values are required by a primitive operation. For example, to evaluate the expression `x+x`, it is necessary to evaluate the binding of `x` in order to perform the addition. Languages that adopt the by-name discipline are, for this reason, said to be *lazy*.

This discussion glosses over another important aspect of lazy evaluation, called *memoization*. In actual fact laziness is based on a refinement of the *by-name* principle, called the *by-need* principle. According to the by-name principle, variables are bound to unevaluated computations, and are evaluated only as often as the value of that variable's binding is required to complete the computation. In particular, to evaluate the expression `x+x` the value of the binding of `x` is need *twice*, and hence it is evaluated twice. According to the by-need principle, the binding of a variable is evaluated *at most once* — not at all, if it is never needed, and exactly once if it ever needed at all. Re-evaluation of the same computation is avoided by *memoization*. Once a computation is evaluated, its value is saved for future reference should that computation ever be needed again.

The advantages and disadvantages of lazy *vs.* eager languages have been hotly debated. We will not enter into this debate here, but rather content ourselves with the observation that *laziness is a special case of eagerness*. (Recent versions of) ML have *lazy data types* that allow us to treat unevaluated computations as values of such types, allowing us to incorporate laziness into the language without disrupting its fundamental character on which so much else depends. This affords the benefits of laziness, but on a controlled basis — we can use it when it is appropriate, and ignore it when it is not.

The main benefit of laziness is that it supports *demand-driven* computation. This is useful for representing *on-line* data structures that are created only insofar as we examine them. *Infinite* data structures, such as the sequence of *all* prime numbers in order of magnitude, are one example of an on-line data structure. Clearly we cannot every "finish" creating the sequence of all prime numbers, but we can create as much of this sequence as we need for a given run of a program. *Interactive* data structures, such as the sequence of inputs provided by the user of an interactive system, are another example of on-line data structure. In such a system the user's inputs are not pre-determined at the start of execution, but rather are created "on demand" in response to the progress of computation up to that point. The demand-driven nature of on-line data structures is precisely what is needed to model this behavior.

Note: Lazy evaluation is a non-standard feature of ML that is supported only by the SML/NJ compiler. The lazy evaluation features must be enabled by executing the following at top level:

```
Compiler.Control.lazysml := true;
open Lazy;
```

15.1 Lazy Data Types

SML/NJ provides a general mechanism for introducing lazy data types by simply attaching the keyword `lazy` to an ordinary `datatype` declaration. The ideas are best illustrated by example. We will focus attention on the type of *infinite streams*, which may be declared as follows:

```
datatype lazy 'a stream = Cons of 'a * 'a stream
```

Notice that this type definition has no "base case"! Had we omitted the keyword `lazy`, such a `datatype` would not be very useful, since there would be no way to create a value of that type!

Adding the keyword `lazy` makes all the difference. Doing so specifies that the values of type *typ* `stream` are *computations* of values of the form

`Cons (`val`, `val'`),`

where val is of type *typ*, and val' is another such computation. Notice how this description captures the "incremental" nature of lazy data structures. The computation is not evaluated until we examine it. When we do, its structure is revealed as consisting of an element val together with another suspended computation of the same type. Should we inspect that computation, it will again have this form, and so on *ad infinitum*.

Values of type *typ* `stream` are created using a `val rec lazy` declaration that provides a means for building a "circular" data structure. Here is a declaration of the infinite stream of `1`'s as a value of type `int stream`:

```
val rec lazy ones = Cons (1, ones)
```

The keyword `lazy` indicates that we are binding `ones` to a computation, rather than a value. The keyword `rec` indicates that the computation is *recursive* (or *self-referential* or *circular*). It is the computation whose underlying value is constructed using `Cons` (the only possibility) from the integer `1` and *the very same computation itself*.

We can inspect the underlying value of a computation by pattern matching. For example, the binding

```
val Cons (h, t) = ones
```

extracts the "head" and "tail" of the stream `ones`. This is performed by evaluating the computation bound to `ones`, yielding `Cons (1, ones)`, then performing ordinary pattern matching to bind `h` to `1` and `t` to `ones`.

Had the pattern been "deeper", further evaluation would be required, as in the following binding:

```
val Cons (h, (Cons (h', t'))) = ones
```

To evaluate this binding, we evaluate ones to `Cons (1, ones)`, binding `h` to `1` in the process, then evaluate `ones` again to `Cons (1, ones)`, binding `h'` to `1` and `t'` to `ones`. The general rule is *pattern matching forces evaluation of a computation to the extent required by the pattern*. This is the means by which lazy data structures are evaluated only insofar as required.

15.2 Lazy Function Definitions

The combination of (recursive) `lazy` function definitions and decomposition by pattern matching are the core mechanisms required to support lazy evaluation. However, there is a subtlety about function definitions that requires careful consideration, and a third new mechanism, the *lazy function* declaration.

Using pattern matching we may easily define functions over lazy data structures in a familiar manner. For example, we may define two functions to extract the head and tail of a stream as follows:

```
fun shd (Cons (h, _)) = h
fun stl (Cons (_, s)) = s
```

These are functions that, when applied to a stream, evaluate it, and match it against the given patterns to extract the head and tail, respectively.

While these functions are surely very natural, there is a subtle issue that deserves careful discussion. The issue is whether these functions are "lazy enough". From one point of view, what we are doing is decomposing a computation by evaluating it and retrieving its components. In the case of the `shd` function there is no other interpretation — we are extracting a value of type *typ* from a value of type *typ* `stream`, which is a computation of a value of the form `Cons` $(exp_h,\ exp_t)$. We can adopt a similar viewpoint about `stl`, namely that it is simply extracting a component value from a computation of a value of the form `Cons` $(exp_h,\ exp_t)$.

However, in the case of `stl`, another point of view is also possible. Rather than think of `stl` as *extracting* a value from a stream, we may instead think of it as *creating* a stream out of another stream. Since streams are computations, the stream created by `stl` (according to this view) should also be suspended until its value is required. Under this interpretation the argument to `stl` should not be evaluated until its result is required, rather than at the time `stl` is applied. This leads to a variant notion of "tail" that may be defined as follows:

```
fun lazy lstl (Cons (_, s)) = s
```

The keyword `lazy` indicates that an application of `lstl` to a stream does *not* immediately perform pattern matching on its argument, but rather *sets up* a stream computation that, when forced, forces the argument and extracts the tail of the stream.

The behavior of the two forms of tail function can be distinguished using `print` statements as follows:

```
val rec lazy s = (print "."; Cons (1, s))
val _ = stl s    (* prints "." *)
val _ = stl s    (* silent *)

val rec lazy s = (print "."; Cons (1, s));
val _ = lstl s   (* silent *)
val _ = stl s    (* prints "." *)
```

Since `stl` evaluates its argument when applied, the "`.`" is printed when it is first called, but not if it is called again. However, since `lstl` only sets up a computation, its argument is not evaluated when it is called, but only when its result is evaluated.

15.3 Programming with Streams

Let's define a function `smap` that applies a function to every element of a stream, yielding another stream. The type of `smap` should be `('a -> 'b) -> 'a stream -> 'b stream`. The thing to keep in mind is that the application of `smap` to a function and a stream should set up (but not compute) another stream that, when forced, forces the argument stream to obtain the head element, applies the given function to it, and yields this as the head of the result.

Here's the code:

```
fun smap f =
    let
        fun lazy loop (Cons (x, s)) =
            Cons (f x, loop s)
    in
        loop
    end
```

We have "staged" the computation so that the partial application of `smap` to a function yields a function that loops over a given stream, applying the given function to each element. This loop is a lazy function to ensure that it merely sets up a stream computation, rather than evaluating its argument when it is called. Had we dropped the keyword `lazy` from the definition of the loop, then an application of `smap` to a function and a stream would immediately force the computation of the head element of the stream, rather than merely set up a future computation of the same result.

To illustrate the use of smap, here's a definition of the infinite stream of natural numbers:

```
val one_plus = smap (fn n => n+1)
val rec lazy nats = Cons (0, one_plus nats)
```

Now let's define a function `sfilter` of type

```
('a -> bool) -> 'a stream -> 'a stream
```

that filters out all elements of a stream that do not satisfy a given predicate.

```
fun sfilter pred =
    let
        fun lazy loop (Cons (x, s)) =
            if pred x then
```

```
               Cons (x, loop s)
            else
               loop s
    in
        loop
    end
```

We can use `sfilter` to define a function `sieve` that, when applied to a stream of numbers, retains only those numbers that are not divisible by a preceding number in the stream:

```
fun m mod n = m - n * (m div n)
fun divides m n = n mod m = 0
fun lazy sieve (Cons (x, s)) =
    Cons (x, sieve (sfilter (not o (divides x)) s))
```

We may now define the infinite stream of primes by applying `sieve` to the natural numbers greater than or equal to 2:

```
val nats2 = stl (stl nats)
val primes = sieve nats2
```

To inspect the values of a stream it is often useful to use the following function that takes $n \geq 0$ elements from a stream and builds a list of those n values:

```
fun take 0 _ = nil
  | take n (Cons (x, s)) = x :: take (n-1) s
```

Here's an example to illustrate the effects of memoization:

```
val rec lazy s = Cons ((print "."; 1), s)
val Cons (h, _) = s;
(* prints ".", binds h to 1 *)
val Cons (h, _) = s;
(* silent, binds h to 1 *)
```

Replace `print ".";1` by a time-consuming operation yielding `1` as result, and you will see that the second time we force `s` the result is returned instantly, taking advantage of the effort expended on the time-consuming operation induced by the first force of `s`.

Chapter 16

Equality and Equality Types

Chapter 17

Concurrency

Concurrent ML (CML) is an extension of Standard ML with mechanisms for concurrent programming. It is available as part of the Standard ML of New Jersey compiler. The eXene Library for programming the X windows system is based on CML.