through the details here. We did not prove Progress for STLC, because it is trivially true in that case. Here, however, proof is required. We will consider just the cases for simply typed lambda calculus extended with booleans and *if-then-else* (see Sections 11.1.1 and 11.1.1 above). The proof is by induction on the structure of the assumed typing derivation.

Case:

$$\frac{}{\cdot \vdash \mathit{true} : \mathit{bool}}$$

The term is a value in this case, so the required result holds. The case for *false* is exactly similar.

Case:

$$\frac{\cdot \vdash t_1 : \mathit{bool} \quad \cdot \vdash t_2 : T \quad \cdot \vdash t_3 : T}{\cdot \vdash \mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3 : T}$$

By the induction hypothesis, t_1 either steps to some t_1', in which case the whole *if-then-else* term steps to *if* t_1' *then* t_2 *else* t_3; or else t_1 is a value. If it is a value, then by inversion on the assumed derivation of $\cdot \vdash t_1 : \mathit{bool}$, it must either be *true* or *false*. In either case, the whole *if-then-else* term steps.

Case:

$$\frac{\cdot, x : T \vdash t : T'}{\cdot \vdash \lambda x : T.t : T \to T'}$$

The term in question is already a value, as required.

Case:

$$\frac{\cdot \vdash t_1 : T \to T' \qquad \cdot \vdash t_2 : T}{\cdot \vdash t_1 t_2 : T'}$$

By the induction hypothesis, either t_1 steps, in which case the whole application steps, also; or else t_1 is a value. By inversion on the assumed typing derivation for t_1, this value must be a λ-abstraction, in which case the whole application β-reduces, as required. **End proof.**

11.3 Connection to practice: eager functional programming in OCAML

The OCAML programming language supports ideas similar to those discussed above in an eminently usable and performant implementation, with excellent documentation, freely available online. This section gives a quick tutorial to central features of OCAML. For more information, see various resources, including an excellent reference manual with thorough documentation of standard library functions, linked from `http://caml.inria.fr` (the OCAML compiler can also be downloaded from that site).

11.3.1 Compiling and running OCAML programs

File structure. An OCAML file contains non-recursive definitions of the form

```
let a x1 ... xn = t ;;
```

where x_1 through x_n are input variables to a (or omitted, if a is not a function or is just defined to be an explicit functional term), and t is the body of the function. There are also recursive definitions of the form

```
let rec a x1 ... xn = t ;;
```

These are similar, but a may be used in t to make recursive calls. OCAML files can also just contain terms by themselves:

```
t;;
```

which will be evaluated when the program is executed (note that their values will not be printed from output compiled as described next). For example, to write a hello-world program, it is sufficient to put the following in a file called `test.ml` and compile it as described below.

```
print_string "Hello, world.\n";;
```

This calls the standard-library function `print_string`. OCAML files can also contain several other kinds of top-level commands, including type declarations, discussed below.

Compiling to bytecode. OCAML can be easily compiled to OCAML bytecode format, which is then efficiently executed by an OCAML virtual machine, on many platforms, including Linux, Windows, and Mac (I have personally tried the former two with OCAML version 3.11.1). Native-code compilation is also supported on some platforms, but in my experience can be harder to get working on Windows (though it is easy on Linux). To compile a single OCAML source file to a bytecode executable, run

```
ocamlc -o file file.ml
```

To compile multiple sources files `a.ml`, `b.ml`, and `c.ml`, use the following commands:

```
ocamlc -c a.ml
ocamlc -c b.ml
ocamlc -c c.ml
```

This will generate files ending in `.cmo` (also ones ending in `.cmi`). To link these together into an executable called `test`, use this command:

```
ocamlc -o test a.cmo b.cmo c.cmo
```

Note that the order of these `.cmo` files matters: if file `b.ml` depends on file `a.ml`, then one must list `a.cmo` earlier than `b.cmo`, as shown.

Running online. At the time of this writing, you can also run OCAML programs at the `http://codepad.org` web site. You just enter your program text into a provided input pane, select "OCAML" from the list of supported programming languages, and submit the code for compilation and execution.

Using the `ocaml` interpreter. To evaluate expressions directly, just start the OCAML interpreter `ocaml`. On Linux, this can be done from the shell like this (on Windows, one can start OCAML from the `cmd` program, or by launching the OCAML interpreter that is included with the distribution):

```
ephesus:~/papers/plf-book$ ocaml
        Objective Caml version 3.11.2

#
```

Now one can enter expressions to evaluate, after the # sign:

```
ephesus:~/papers/plf-book$ ocaml
        Objective Caml version 3.11.2

# 3+4+5;;
- : int = 12
#
```

The interpreter prints out the type `int` and the value `12` to which this expression evaluates.

11.3.2 Language basics

Basic top-level functions. We can define non-recursive top-level functions in OCAML like this:

```
let square x = x * x;;
```

This defines the function `square` to take in an input `x`. The value returned by the function is then `x * x`. In keeping with its connection to mathematics, functional languages do not explicitly use `return` to state that something is the ordinary return-value for a function (HASKELL does use `return`, but only for a more advanced functional-programming design pattern called a *monad*). Notice that it is not necessary to state any type information for an OCAML program like this. Type information is inferred automatically by the OCAML type checker, and is thus purely optional. If we want to include type information, we can include it like this for function definitions:

```
let square (x:int) : int = x * x;;
```

This states that the type of the input `x` is `int`. Also, the second "`: int`" indicates that the return type of the function is also `int`. Note that OCAML supports basic arithmetic operations like the multiplication used here. It has operations for 32-bit integers and also floating point numbers. The type `int` is for 32-bit integers. See the OCAML Reference Manual for complete details [23].

Inductive datatypes. OCAML allows programmers to define their own datatypes, called inductive because each piece of data is uniquely and incrementally built by applying constructors to other data – central characteristics of inductive definitions. Members of these datatypes can be thought of as trees, storing different kinds of data. For example, we might wish to define a datatype for abstract syntax trees for a language with addition and integers. An example of the kind of abstract syntax tree we want to support for this language is in Figure 11.1. The tree shown might be the one a parser generates for the string "`1+2+3`". To declare the type for abstract syntax trees like this one, we can use the following OCAML code:

```
type expr = Plus of expr * expr | Num of int;;
```

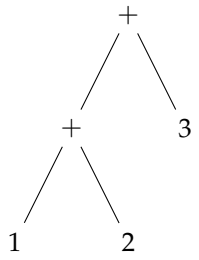

Figure 11.1: Example abstract syntax tree

This declares a new OCAML type called `expr`, with constructors `Plus` and `Num` for building nodes of the abstract syntax tree. The code "`of int`" for `Num` expresses that the `Num` constructor holds an `int`. Similarly, the `of expr * expr` code expresses that `Plus` holds a pair of two `expr`'s.

Pattern matching. OCAML supports pattern-matching on members of datatypes like the `expr` datatype shown above. For example, the following top-level non-recursive function uses pattern-matching to check whether or not an `expr` has `Plus` at its root (we call such an `expr` a `Plus-expr`), or not:

```
let isPlus e =
  match e with
    Plus(_,_) -> true
  | _ -> false;;
```

This code defines a function called `isPlus` which accepts an input `e`. The function pattern-matches on `e`. If `e` matches the pattern `Plus(_,_)`, then the value returned by the function is `true`, which is declared in OCAML's standard library as a `bool`. The underscores tell OCAML that we do not care about the left and right parts of the tree while matching against this pattern. If the tree `e` does not have `Plus` at its root, then OCAML will consider the next case. Here, we have just an underscore, indicating that we do not care what the tree looks like in this case. The boolean value `false` is then the return value for this case.

For another simple example, the following function returns the left subtree of a `Plus-expr`. It uses pattern variables `l` and `r` to refer to the left and right subtrees, respectively, of any matching `Plus-expr`. We could also have used an underscore instead of `r`, since the code for that case does not use `r`, but only `l` (for which we then could not have used an underscore, since we wish to refer to the matching value in that case).

```
let getLeft e =
  match e with
    Plus(l,r) -> l;;
```

Pairs and tuples. To put two pieces of data `x` and `y` together into a pair, we just use the standard ordered pair notation `(x,y)`. To take apart a pair `p` into its first and second components `x` and `y`, we can use pattern matching. For example, the following code defines a function `addComponents` which takes a pair of two `int`'s and returns their sum:

```
let addComponents (p : int * int) : int =
  match p with
    (x,y) -> x + y;;
```

Here we see that the type of the input (i.e., the ordered pair) is `int * int`, which demonstrates the OCAML syntax for types of pairs. Note that the type `int * (int * int)` is not considered the same, in OCAML, as the type `int * int * int` of triples.

Lists. OCAML provides some special syntax for lists. The empty list is denoted `[]`, and adding (or "consing") a new element `a` to the start of a list `L` is denoted `a::L`. This notation can be used in pattern-matching terms as well. For example, a recursive function to compute the length of a list can be written like this:

```
let rec length l =
  match l with
    [] -> 0
  | x::l' -> 1 + (length l')
;;
```

Here, as above, "`let rec`" introduces a top-level recursive function. The name of the function is `length`, and the input argument is named `l`. The function does pattern matching on `l`, with two cases. In the first case, `l` is `[]`, the empty list. In that case, the returned value is 0. In the second case, the list `l` matches the pattern `x::l'`. This means that its first element is `x`, and the rest of it is the list `l'`. We return one plus the result of recursively computing the length of the list `l'`.

OCAML has some other notation related to lists. First, if we wish to write down the list of the first four numbers starting from 0, we could write `0::1::2::3::[]`. That is the list we get by putting three onto the front of the empty list, then two onto the front of that, then 1, and finally 0. Alternative, slightly more readable notation in OCAML for this same list is `[0 ; 1 ; 2 ; 3]`. The general form of this alternative notation is to list elements in between square brackets, separated by semi-colons. The empty list `[]` can then be seen as a special case of that notation. Similarly, a *singleton* list containing exactly one element is also: we can write `[2]` for the list containing the single element 2. Finally, the operation which appends two lists can be written using infix `@`. So `[ 1 ; 2 ] @ [ 3 ; 4 ; 5]` is notation for calling the append function on the two given lists. This will result, of course, in the list `[ 1 ; 2 ; 3 ; 4 ; 5]`.

Recursive top-level functions To define a recursive function at the top level, as mentioned above, we just have to write "`let rec`" instead of "`let`" when writing our top-level definition. For example, here is a function to compute the length of a list (this is provided already in OCAML's standard library as `List.length`):

```
let rec length l =
  match l with
    [] -> 0
  | x::xs -> 1 + (length xs);;
```

In the body of a recursive function like this one, we can refer recursively, on the right hand side of the definition, to the function (`length`, in this case) which we are defining.

Let-terms. To give a name for the value computed by some expression, OCAML provides `let`-notation. For example:

```
let x = 10 * 10 in
  x * x
```

This makes `x` refer to the value of `10 * 10` in its *body*, which is the subexpression following the `in`-keyword. So this whole expression has value ten thousand. The type of the `let`-term is the type of its body. Functions, both non-recursive and recursive, can be defined using `let`-terms. For example, here is some code which uses a `let`-term to abstract out some code for logging from a bigger function `foo`:

```
let foo (log:out_channel) arg1 ... argn =
  let write_log (msg:string) =
    output_string log msg in
  ...
  write_log "some message";
  write_log "another";
  ...
;;
```

The `let`-term defining the function `write_log` uses a similar syntax as for top-level functions (see above). Notice how the definition of `write_log` refers to a variable in the surrounding context, namely `log`, without requiring it as an extra input. This helps keep calls to `write_log` more concise.

Mutually inductive datatypes, mutually recursive functions. In addition to the inductive datatypes and recursive functions explained above, OCAML also supports the definition of mutually inductive datatypes, and mutually recursive functions. Two or more types (respectively, two or more functions) are defined, and the definition of each can refer to the other. The keyword `and` is used to separate the definitions, for both types and recursive functions. Here is a simple example:

```
type even = Z | Se of odd
 and odd = So of even;;
```

This declares two mutually inductive types, of even and odd numbers in unary notation (see Section 6.2.1). OCAML reports the following typings for some example terms built using the constructors for these types:

```
Z : even
(So Z) : odd
(Se (So Z)) : even
(So (Se (So Z))) : odd
(Se (So (Se (So Z)))) : even
```

The odd numbers indeed have type `odd`, and the even numbers `even`. The following definitions define addition functions for all possible combinations of even and odd inputs. Note how the return types correctly capture the behavior of the usual definition of unary addition: for example, adding two odd numbers produces an even number.

```
let rec plusee (e1:even) (e2:even) : even =
  match e1 with
    Z -> e2
  | Se(o1) -> Se(plusoe o1 e2)
and plusoe (o:odd) (e:even) : odd =
  match o with
    So(e1) -> So(plusee e1 e)
and pluseo (e:even) (o:odd) : odd =
  match e with
    Z -> o
  | Se o1 -> So(plusoo o1 o)
and plusoo (o1:odd) (o2:odd) : even =
  match o1 with
   So e -> Se(pluseo e o2)
;;
```

11.3.3 Higher-Order Functions

As essential aspect of functional programming, well supported by OCAML, is the use of higher-order functions. These are functions that accept other functions as inputs, or produce them as outputs. Anonymous functions are also commonly used, and can be nested inside other functions. The notation for an anonymous function in OCAML is

```
fun x -> t
```

So the squaring function we defined in the previous section as a top-level function can be written as an anonymous function like this:

```
fun x -> x * x
```

Functions as inputs. Here is a top-level function called `applyTwice`, which accepts a function `f` and argument `x` as inputs, and applies `f` twice: first to `x`, and then to the result of the first application:

```
let applyTwice f x = f (f x)
```

Note, in passing, the notation for a nested function call: parentheses are placed around the call of `f` on `x`; `f` is then applied again to that parenthesized expression.

We can call `applyTwice` with our anonymous squaring function and the argument 3 to raise 3 to the fourth power:

```
applyTwice (fun x -> x * x) 3
```

If we were using our top-level definition of the squaring function, we could just as well write

```
applyTwice square 3
```

Partial applications. In OCAML, functions defined with N input variables can be called with fewer than N arguments. An application of a function to fewer than the number of input variables stated in its definition is called a partial application. For example, the `applyTwice` function we just defined has two input variables, `f` and `x`. But we are allowed to call it with just the first one; for example,

```
applyTwice (fun x -> x * x)
```

What value is it that we get back from a partial application like this one? We get back a new function, which is waiting for the remaining arguments. In this case, we get back a function which, when given the remaining needed argument `x`, will return the square of the square of `x`. Suppose we write a top-level definition like this:

```
let pow4 = applyTwice (fun x -> x * x);;
```

If we can call `pow4` on an argument like 3, we will get 81. If we call it on 4, we will likewise get the expected result (256). So by using a partial application, we have abstracted out an interesting piece of functionality, namely raising to the fourth power by applying squaring twice. This abstracted value, `pow4`, can now be used repeatedly with different arguments; for example:

```
pow4 3;;
pow4 4;;
```

This is more concise than writing the following, for example, to process several numbers:

```
applyTwice (fun x -> x * x) 3;;
applyTwice (fun x -> x * x) 4;;
```

Syntax for lists. Lists are used frequently in functional programming as a basic data structure, and both OCAML and HASKELL (which we will discuss in Section 11.4) have special syntax for common operations on lists. In OCAML, the empty list is written `[]`; and the list `cons x l` (which constructs a new list which begins with element `x` and continues with the sublist `l`) is written `x::l`. OCAML supports a form of polymorphism, which we will study in Chapter 10, so lists are allowed with any (single) type of element. A list of integers has type `int list` in OCAML, and a list of booleans `bool list`. In general, a list of elements of type `'a` (OCAML uses names beginning with a single quotation mark for type variables) has type `'a list`. So `list` is a type constructor. OCAML generally writes type constructors in postfix notation. So we have the following typings:

```
1::2::3::4::[]                        :   int list
true::true::false::[]                 :   bool list
(fun x -> x)::(fun y -> y + y)::[]    :  (int -> int) list
```

True to its nature as a functional language, OCAML allows functions to be manipulated much like any other data, as demonstrated in the third example just above.

Pattern matching. OCAML supports pattern-matching on data like lists. For example, the following function uses pattern-matching to check whether or not a list is null:

```
fun l -> match l with
           [] -> true
         | x::xs -> false
```

The variables `x` and `xs` are **pattern variables**, whose scope is to the right of the arrow symbol. When the pattern-matching expression is evaluated, they will be bound to the values of the first element in the list and the immediate sublist, respectively, if indeed the list matched is a `cons`-list.

Defining top-level functions. We could define the function just above at the top level in an OCAML file like this:

```
let null = fun l -> match l with
                      [] -> true
                    | x::xs -> false
```

OCAML provides a little syntactic sugar for such definitions, where we can write the argument to the function to the left of the equality symbol:

```
let null l = match l with
               [] -> true
             | x::xs -> false
```

This is equivalent to the definition using `fun`.

11.4 Lazy Programming with Call-By-Name Evaluation

In this section, we consider how call-by-name evaluation (first considered in Section 5.4.4) can be used for practical programming. The main benefit of call-by-name evaluation is that terms do not have to be evaluated to be passed as arguments to functions. This opens up the possibility of **lazy evaluation**, where computation is deferred until it absolutely must be performed in order to satisfy an immediate need for a value, such as to print it or communicate it over a network channel. For better performance, implementations of lazy programming use an optimized form of call-by-name evaluation, known as **call-by-need**. The basic idea of this optimization is as follows. When doing call-by-name evaluation, unevaluated terms can be duplicated by β-reduction. While lazy evaluation will try to avoid evaluating any of those copies of the term, it could happen that several of them do need to be evaluated. In that case, it is wasteful to evaluate each of the duplicate copies of the term separately, since the computation is exactly the same in each case. Call-by-need evaluation caches the result the first time that term is evaluated. If subsequent computation requires evaluating a copy of the term, call-by-need evaluation just uses the cached result, instead of re-evaluating the term. Languages like HASKELL are typically implemented using call-by-need evaluation. We will consider lazy programming using just call-by-name evaluation, since except for possibly (much) slower execution, this gives the same results as call-by-need. We will consider HASKELL (Section 11.5) as an example lazy-programming language.

11.4.1 Syntax and typing for lazy programming

Here, we develop a small lazy-programming language based on call-by-name evaluation. The language has basic arithmetic, λ-abstractions, list constructs, and a recursion operator, and is thus quite similar to the eager (call-by-value) language we studied in Section 11.1. The typing rules for these constructs are just as they were