4
Trees and Concrete Data

Concrete data consists of constructions that can be inspected, taken apart, or joined to form larger constructions. Lists are an example of concrete data. We can test whether or not a list is empty, and divide a non-empty list into its head and tail. New elements can be joined to a list. This chapter introduces several other forms of concrete data, including trees and logical propositions.

The ML `datatype` declaration defines a new type along with its ***constructors***. In an expression, constructors create values of a datatype; in patterns, constructions describe how to take such values apart. A datatype can represent a class consisting of distinct subclasses — like Pascal's variant records, but without their complications and insecurities. A recursive datatype typically represents a tree. Functions on datatypes are declared by pattern-matching.

The special datatype exn is the type of ***exceptions***, which stand for error conditions. Errors can be signalled and trapped. An exception handler tests for particular errors by pattern-matching.

Chapter outline

This chapter describes datatypes, pattern-matching, exception handling and trees. It contains the following sections:

The `datatype` *declaration.* Datatypes, constructors and pattern-matching are illustrated through examples. To represent the King and his subjects, a single type $person$ comprises four classes of individual and associates appropriate information with each.

Exceptions. These represent a class of error values. Exceptions can be declared for each possible error. Raising an exception signals an error; handling the exception allows an alternative computation to be performed.

Trees. A tree is a branching structure. Binary trees are a generalization of lists and have many applications.

Tree-based data structures. Dictionaries, flexible arrays and priority queues are implemented easily using binary trees. The update operation creates a new data structure, with minimal copying.

A tautology checker. This is an example of elementary theorem proving. A datatype of propositions (boolean expressions) is declared. Functions convert propositions to conjunctive normal form and test for tautologies.

The `datatype` declaration

A heterogeneous class consists of several distinct subclasses. A circle, a triangle and a square are all shapes, but of different kinds. A triangle might be represented by three points, a square by four points and a circle by its radius and centre.

For a harder problem, consider cataloguing all the inhabitants of the Kingdom by their class. These comprise the King, the Peers (or nobility), the Knights and the Peasants. For each we record appropriate information:

- The King is simply himself. There is nothing more to say.
- A Peer has a degree, territory and number in succession (as in 'the 7th Earl of Carlisle.')
- A Knight or Peasant has a name.

In weakly typed languages, these subclasses can be represented directly. We need only take care to distinguish Knights from Peasants; the others will differ naturally. In ML we could try

```
"King"
("Earl","Carlisle",7)       ("Duke","Norfolk",9)
("Knight","Gawain")         ("Knight","Galahad")
("Peasant","Jack Cade")     ("Peasant","Wat Tyler")
```

Unfortunately, these do not all have the same type! No ML function could handle both Kings and Peasants with this representation.

4.1 *The King and his subjects*

An ML type consisting of King, Peers, Knights and Peasants is created by a `datatype` declaration:

```
datatype person = King
                | Peer of string*string*int
                | Knight of string
                | Peasant of string;
> datatype  person
>   con King    : person
>   con Peer    : string * string * int -> person
>   con Knight  : string -> person
>   con Peasant : string -> person
```

Five things are declared, namely the type *person* and its four ***constructors*** *King*, *Peer*, *Knight* and *Peasant*.

The type *person* consists precisely of the values built by its constructors. Note that *King* has type *person*, while the other constructors are functions that return something of that type. Thus the following have type *person*:

```
King
Peer("Earl","Carlisle",7)    Peer("Duke","Norfolk",9)
Knight "Gawain"              Knight "Galahad"
Peasant "Jack Cade"          Peasant "Wat Tyler"
```

Furthermore, these values are distinct. No *person* can be both a *Knight* and a *Peasant*; no *Peer* can have two different degrees.

Values of type *person*, like other ML values, may be arguments and results of functions and may belong to data structures such as lists:

```
val persons = [King, Peasant "Jack Cade", Knight "Gawain"];
> val persons = [King, Peasant "Jack Cade",
>                Knight "Gawain"] : person list
```

Since each *person* is a unique construction, it can be taken apart. A function over a datatype can be declared through patterns involving the constructors. As with lists, there may be several cases. A person's title depends upon his class and is constructed using string concatenation (^):

```
fun title King               = "His Majesty the King"
  | title (Peer(deg,terr,_)) = "The " ^ deg ^ " of " ^ terr
  | title (Knight name)      = "Sir " ^ name
  | title (Peasant name)     = name;
> val title = fn : person -> string
```

Each case is governed by a pattern with its own set of pattern variables. The *Knight* and *Peasant* cases each involve a variable called *name*, but these variables have separate scopes.

```
title(Peer("Earl", "Carlisle", 7));
> "The Earl of Carlisle" : string
title(Knight "Galahad");
> "Sir Galahad" : string
```

Patterns may be as complicated as necessary, combining tuples and the list constructors with datatype constructors. The function *sirs* returns the names of all the Knights in a list of persons:

```
fun sirs []                  = []
  | sirs ((Knight s) :: ps) = s :: (sirs ps)
  | sirs (p :: ps)          = sirs ps;
> val sirs = fn : person list -> string list
sirs persons;
> ["Gawain"] : string list
```

The cases in a function are considered in order. The third case (with pattern $p::ps$) is not considered if p is a $Knight$, and therefore must not be taken out of context. Some people prefer that the cases should be disjoint in order to assist mathematical reasoning. But replacing the case for $p::ps$ with separate cases for $King$, $Peer$ and $Peasant$ would make the function longer, slower and less readable. The third case of $sirs$ makes perfect sense as a conditional equation, holding for all p not of the form $Knight(s)$.

The ordering of cases is even more important when one $person$ is compared with another. Rather than testing 16 cases, we test for each $true$ case and take all the others for $false$, a total of 7 cases. Note the heavy use of wildcards in patterns.

```
fun superior (King, Peer _)         = true
  | superior (King, Knight _)       = true
  | superior (King, Peasant _)      = true
  | superior (Peer _, Knight _)     = true
  | superior (Peer _, Peasant _)    = true
  | superior (Knight _, Peasant _)  = true
  | superior _                      = false;
> val superior = fn : person * person -> bool
```

Exercise 4.1 Write an ML function to map persons to integers, mapping Kings to 4, Peers to 3, Knights to 2 and Peasants to 1. Write a function equivalent to $superior$ that works by comparing the results of this mapping.

Exercise 4.2 Modify type $person$ to add the constructor $Esquire$, whose arguments are a name and a village (both represented by strings). What is the type of this constructor? Modify function $title$ to generate, for instance,

```
"John Smith, Esq., of Bottisham"
```

Modify $superior$ to rank $Esquire$ above $Peasant$ and below $Knight$.

Exercise 4.3 Declare a datatype of geometric figures such as triangles, rectangles, lines and circles. Declare a function to compute the area of a figure.

4.2 Enumeration types

Letting strings denote degrees of nobility may be inadvisable. It does not prevent spurious degrees like `"butcher"` and `"madman"`. There are only five valid degrees; let them be the constructors of a new datatype:

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron;
```

Now type *person* should be redeclared, giving *Peer* the type

$$degree \times string \times int \to person.$$

Functions on type *degree* are defined by case analysis. What is the title of a lady of quality?

```
fun lady Duke     = "Duchess"
  | lady Marquis  = "Marchioness"
  | lady Earl     = "Countess"
  | lady Viscount = "Viscountess"
  | lady Baron    = "Baroness";
> val lady = fn : degree -> string
```

Accuracy being paramount in the Court and Social column, we cannot overestimate the importance of this example for electronic publishing.

A type like *degree*, consisting of a finite number of constants, is an ***enumeration type***. Another example is the built-in type *bool*, which is declared by

```
datatype bool = false | true;
```

The function *not* is declared by cases:

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

The standard library declares the enumeration type *order* as follows:

```
datatype order = LESS | EQUAL | GREATER;
```

This captures the three possible outcomes of a comparison. The library structures for strings, integers, reals, times, dates, etc., each include a function *compare* that returns one of these three outcomes:

```
String.compare ("York", "Lancaster");
> GREATER : order
```

Relations that return a boolean value are more familiar. But we need two calls of < to get as much information as we can get from one call to *String*.*compare*. The first version of Fortran provided three-way comparisons! The more things change, the more they stay the same ...

Beware of redeclaring a datatype. Each `datatype` declaration creates a new type distinct from all others. Suppose we have declared the type $degree$ and the function $lady$. Now, repeat the declaration of $degree$. This declares a new type with new constructors. Asking for the value of $lady(Duke)$ will elicit the type error 'expected type $degree$, found type $degree$.' Two different types are now called $degree$. This exasperating situation can happen while a program is being modified interactively. The surest remedy is to terminate the ML session, start a new one, and load the program afresh.

Exercise 4.4 Declare an enumeration type consisting of the names of six different countries. Write a function to return the capital city of each country as a string.

Exercise 4.5 Write functions of type $bool \times bool \to bool$ for boolean conjunction and disjunction. Use pattern-matching rather than `andalso`, `orelse` or `if`. How many cases have to be tested explicitly?

4.3 *Polymorphic datatypes*

Recall that $list$ is a type operator taking one argument.[1] Thus $list$ is not a type, while $(int)list$ and $((string \times real)list)list$ are. A `datatype` declaration can introduce type operators.

The 'optional' type. The standard library declares the type operator $option$:

```
datatype 'a option = NONE | SOME of 'a;
> datatype 'a option
>   con NONE : 'a option
>   con SOME : 'a -> 'a option
```

The type operator $option$ takes one argument. Type $\tau\ option$ contains a copy of type τ, augmented with the extra value *NONE*. It can be used to supply optional data to a function, but its most obvious use is to indicate errors. For example, the library function $Real.fromString$ interprets its string argument as a real number, but it does not accept every way of expressing 60,000:

```
Real.fromString "6.0E5";
> SOME 60000.0 : real option
Real.fromString "full three score thousand";
```

[1] The correct term is ***type constructor*** (Milner *et al.*, 1990). I avoid it here to prevent confusion with a constructor of a datatype.

```
> NONE : real option
```

You can use a `case` expression to consider the alternatives separately; see Section 4.4 below.

The disjoint sum type. A fundamental operator forms the ***disjoint sum*** or ***union*** of two types:

```
datatype ('a,'b)sum = In1 of 'a | In2 of 'b;
```

The type operator sum takes two arguments. Its constructors are

$$In1 : \alpha \to (\alpha, \beta)sum$$
$$In2 : \beta \to (\alpha, \beta)sum$$

The type $(\sigma, \tau)sum$ is the disjoint sum of the types σ and τ. Its values have the form $In1(x)$ for x of type σ, or $In2(y)$ for y of type τ. The type contains a copy of σ and a copy of τ. Observe that $In1$ and $In2$ can be viewed as labels that distinguish σ from τ.

The disjoint sum allows values of several types to be present where normally only a single type is allowed. A list's elements must all have the same type. If this type is $(string, person)sum$ then an element could contain a string or a person, while type $(string, int)sum$ comprises strings and integers.

$$[In2(King),\ In1(\texttt{"Scotland"})] : ((string, person)sum)list$$
$$[In1(\texttt{"tyrant"}),\ In2(1040)] \quad : ((string, int)sum)list$$

Pattern-matching for the disjoint sum tests whether $In1$ or $In2$ is present. The function $concat1$ concatenates all the strings included by $In1$ in a list:

```
fun concat1 []           = ""
  | concat1 ((In1 s)::l) = s ^ concat1 l
  | concat1 ((In2 _)::l) =     concat1 l;
> val concat1 = fn : (string, 'a) sum list -> string
concat1 [ In1 "O!", In2 (1040,1057), In1 "Scotland" ];
> "O!Scotland" : string
```

The expression $In1$ `"Scotland"` has appeared with two different types, namely $(string, int \times int)sum$ and $(string, person)sum$. This is possible because its type is polymorphic:

```
In1 "Scotland";
> In1 "Scotland" : (string, 'a) sum
```

Representing other datatypes. The disjoint sum can express all other datatypes that are not recursive. The type $person$ can be represented by

$$((unit, string \times string \times int)sum, (string, string)sum)sum$$

with constructors

$$\begin{aligned} King &= In1(In1()) \\ Peer(d, t, n) &= In1(In2(d, t, n)) \\ Knight(s) &= In2(In1(s)) \\ Peasant(s) &= In2(In2(s)) \end{aligned}$$

These are valid as both expressions and patterns. Needless to say, type $person$ is pleasanter. Observe how $unit$, the type whose sole element is (), represents the one King.

Storage requirements. Datatypes require a surprising amount of space, at least with current compilers. A typical value takes four bytes for the tag (which identifies the constructor) and four bytes for each component of the associated tuple. The garbage collector requires a header consisting of a further four bytes. The total comes to twelve bytes for a $Knight$ or $Peasant$ and twenty bytes for a $Peer$. This would include the integer in a $Peer$, but the strings would be stored as separate objects.

The internal values of an enumeration type require no more space than integers, especially on those ML systems where integers have unlimited precision. List cells typically occupy eight to twelve bytes. With a generational garbage collector, the amount of space taken by an object can vary with its age!

Optimizations are possible. If the datatype has only one constructor, no tag needs to be stored. If all but one of the constructors are constants then sometimes the non-constant constructor does not require a tag; this holds for *list* but not for $option$, since the operand of *SOME* can be anything at all. Compared with Lisp, not having types at run-time saves storage. Appel (1992) discusses such issues. With advances in run-time systems, we can expect storage requirements to decrease.

Exercise 4.6 What are the types of $King$, $Peer$, $Knight$ and $Peasant$ as declared above?

Exercise 4.7 Exhibit a correspondence between values of type $(\sigma, \tau)sum$ and certain values of type $(\sigma\ list) \times (\tau\ list)$ — those of the form ([x], []) or ([], [y]).

4.4 *Pattern-matching with* `val`, `as`, `case`

A ***pattern*** is an expression consisting solely of variables, constructors and wildcards. The ***constructors*** comprise

- numeric, character and string constants

- pairing, tupling and record formation
- list and datatype constructors

In a pattern, all names except constructors are variables. Any meaning they may have outside the pattern is insignificant. The variables in a pattern must be distinct. These conditions ensure that values can be matched efficiently against the pattern and analysed uniquely to bind the variables.

Constructors absolutely must be distinguished from variables. In this book, constructors begin with a capital letter while most variables begin with a small letter.[2] However, the standard constructors nil, $true$ and $false$ are also in lower case. A constructor name may be symbolic or infix, such as :: for lists. The standard library prefers constructor names consisting of all capitals, such as *NONE*.

Mistakes in pattern-matching. Typographical errors in patterns can be hard to locate. The following version of the function $title$ contains several errors. Try to spot them before reading on:

```
fun title Kong               = "His Majesty the King"
  | title (Peer(deg,terr,_)) = "The " ^ deg ^ " of " ^ terr
  | title (Knightname)       = "Sir "^ name
  | title Peasant name       = name;
```

The first error is the misspelling of the constructor $King$ as $Kong$. This is a variable and matches all values, preventing further cases from being considered. ML compilers warn if a function has a redundant case; this warning must be heeded!

The second error is $Knightname$: the omission of a space again reduces a pattern to a variable. Since the error leaves the variable $name$ undefined, the compiler should complain.

The third error is the omission of parentheses around $Peasant\ name$. The resulting error messages could be incomprehensible.

Misspelled constructor functions are quickly detected, for

```
fun f (g x) = ...
```

is allowed only if g is a constructor. Other misspellings may not provoke any warning. Omitted spaces before a wildcard, as in $Peer\_$, are particularly obscure.

Exercise 4.8 Which simple mistake in $superior$ would alter the function's behaviour without making any case redundant?

[2] The language Haskell enforces this convention.

Patterns in value declarations. The declaration

```
val P = E
```

defines the variables in the pattern P to have the corresponding values of expression E. We have used this in Chapter 2 to select components from tuples:

```
val (xc,yc) = scalevec(4.0, a);
> val xc = 6.0 : real
> val yc = 27.2 : real
```

We may also write

```
val [x,y,z] = upto(1,3);
> val x = 1 : int
> val y = 2 : int
> val z = 3 : int
```

The declaration fails (raising an exception) if the value of the expression does not match the pattern. When the pattern is a tuple, type checking eliminates this danger.

The following declarations are valid: the values of their expressions match their patterns. They declare no variables.

```
val King = King;
val [1,2,3] = upto(1,3);
```

Constructor names cannot be declared for another purpose using `val`. In the scope of type $person$, the names $King$, $Peer$, $Knight$ and $Peasant$ are reserved as constructors. Declarations like these, regarded as attempts at pattern-matching, will be rejected with a type error message:

```
val King = "Henry V";
val Peer = 925;
```

Layered patterns. A variable in a pattern may have the form

```
Id as P
```

If the entire pattern (which includes the pattern P as a part) matches, then the value that matches P is also bound to the identifier Id. This value is viewed both through the pattern and as a whole. The function $nextrun$ (from Section 3.21) can be coded

```
fun nextrun(run, [])            = ···
  | nextrun(run as r::_, x::xs) =
          if  x < r then (rev run, x::xs)
                    else nextrun(x::run, xs);
```

Here run and r`::_` are the same list. We now may refer to its head as r instead of $hd\ run$. Whether it is more readable than the previous version is a matter for debate.

The `case` *expression.* This is another vehicle for pattern-matching and has the form

`case` E `of` P_1 `=>` E_1 `|` $\cdots$ `|` P_n `=>` E_n

The value of E is matched successively against the patterns $P_1, \ldots, P_n$; if P_i is the first pattern to match then the result is the value of E_i. Thus `case` is equivalent to an expression that declares a function by cases and applies it to E. A typical `case` expression tests for a few explicit values, concluding with a catch-all case:

```
case p-q of
    0 => "zero"
  | 1 => "one"
  | 2 => "two"
  | n => if n < 10 then "lots"  else  "lots and lots"
```

The function $merge$ (also from Section 3.21) can be recoded using `case` to test the first argument before the second:

```
fun merge(xlist,ylist) : real list =
  case xlist of
       []    => ylist
    | x::xs => (case ylist of
                    [] => xlist
                | y::ys => if x<=y then x::merge(xs, ylist)
                                   else y::merge(xlist, ys));
```

In the recursive call, $xlist$ and x`::`xs denote the same list — an effect also obtainable through the pattern $xlist$ `as` x`::`xs.as keyword@`as` keyword

The scope of `case`. No symbol terminates the `case` expression, so enclose it in parentheses unless you are certain there is no ambiguity. Below, the second line is part of the inner `case` expression, although the programmer may have intended it to belong to the outer:

```
case x of 1 => case y of 0 => true | 1 => false
        | 2 => true;
```

The following declaration is not syntactically ambiguous, but many ML compilers parse it incorrectly. The `case` expression should be enclosed in parentheses:

```
fun f [x] = case g x of 0 => true | 1 => false
  | f xs  = true;
```

Exercise 4.9 Express the function *title* using a `case` expression to distinguish the four constructors of type *person*.

Exercise 4.10 Describe a simple method for removing all `case` expressions from a program. Explain why your method does not affect the meaning of the program.

Exceptions

A hard problem may be tackled by various methods, each of which succeeds in a fraction of the cases. There may be no better way of choosing a method than to try one and see if it succeeds. If the computation reaches a dead end, then the method fails — or perhaps determines that the problem is impossible. A proof method may make no progress or may reduce its goal to $0 = 1$. A numerical algorithm may suffer overflow or division by zero.

These outcomes can be represented by a datatype whose values are $Success(s)$, where s is a solution, $Failure$ and $Impossible$. Dealing with multiple outcomes is complicated, as we saw with the topological sorting functions of Section 3.17. The function $cyclesort$, which returns information about success or failure, is more complex than $pathsort$, which expresses failure by (horribly!) calling $hd[]$.

ML deals with failure through ***exceptions***. An exception is ***raised*** where the failure is discovered and ***handled*** elsewhere — possibly far away.

4.5 *Introduction to exceptions*

Exceptions are a datatype of error values that are treated specially in order to minimize explicit testing. When an exception is raised, it is transmitted by all ML functions until it is detected by an ***exception handler***. Essentially a `case` expression, the exception handler specifies what to return for each kind of exception.

Suppose that functions $methodA$ and $methodB$ realize different methods for solving a problem, and that $show$ displays a solution as a string. Using a datatype with constructors $Success$, $Failure$ and $Impossible$, we can display the outcome of an attempted solution by nested case expressions. If $methodA$ fails

then $methodB$ is tried, while if either reports that the problem is impossible then it is abandoned. In all cases, the result has the same type: $string$.

```
case methodA(problem) of
    Success s => show s
  | Failure    => (case methodB(problem) of
                       Success s => show s
                     | Failure    => "Both methods failed"
                     | Impossible => "No solution exists")
  | Impossible => "No solution exists"
```

Now try exception handling. Instead of a datatype of possible outcomes, declare exceptions $Failure$ and $Impossible$:

```
exception Failure;
exception Impossible;
```

Functions $methodA$ and $methodB$ — and any functions they call within the scope of these exception declarations — can signal errors by code such as

```
if      ... then raise Impossible
else if ... then raise Failure
else (*compute successful result*)
```

The attempts to apply $methodA$ and $methodB$ involve two exception handlers:

```
show (methodA(problem)
      handle Failure => methodB(problem))
  handle Failure    => "Both methods failed"
       | Impossible => "No solution exists"
```

The first handler traps $Failure$ from $methodA$, and tries $methodB$. The second handler traps $Failure$ from $methodB$ and $Impossible$ from either method. Function $show$ is given the result of $methodA$, if successful, or else $methodB$.

Even in this simple example, exceptions give a shorter, clearer and faster program. Error propagation does not clutter our code.

4.6 *Declaring exceptions*

An exception name in Standard ML is a constructor of the built-in type exn. This is a datatype with a unique property: its set of constructors can be extended. The exception declaration

```
exception Failure;
```

makes $Failure$ a new constructor of type exn.

While $Failure$ and $Impossible$ are constants, constructors can also be functions:

```
exception Failedbecause of string;
exception Badvalue of int;
```

Constructor *Failedbecause* has the type $string \to exn$ while *Badvalue* has type $int \to exn$. They create exceptions $Failedbecause(msg)$, where msg is a message to be displayed, and $Badvalue(k)$, where k may determine the method to be tried next.

Exceptions can be declared locally using `let`, even inside a recursive function. This can result in different exceptions having the same name and other complications. Whenever possible, declare exceptions at top level. The type of a top level exception must be monomorphic.[3]

Values of type exn can be stored in lists, returned by functions, etc., like values of other types. In addition, they have a special rôle in the operations `raise` and `handle`.

Dynamic types and exn. Because type exn can be extended with new constructors, it potentially includes the values of any type. We obtain a weak form of dynamic typing. This is an accidental feature of ML; CAML treats dynamics in a more sophisticated manner (Leroy and Mauny, 1993).

For example, suppose we wish to provide a uniform interface for expressing arbitrary data as strings. All the conversion functions can have type $exn \to string$. To extend the system with a new type, say $Complex.t$, we declare a new exception for that type, and write a new conversion function of type $exn \to string$:

```
exception ComplexToString of Complex.t;
fun convert_complex (ComplexToString z) = ...
```

This function only works when it is applied to constructor *ComplexToString*. A collection of similar functions might be stored in a dictionary, identified by uniform keys such as strings. We obtain a basic form of object-oriented programming.

4.7 *Raising exceptions*

Raising an exception creates an ***exception packet*** containing a value of type exn. If Ex is an expression of type exn and Ex evaluates to e, then

```
raise Ex
```

evaluates to an exception packet containing e. Packets are not ML values; the only operations that recognize them are `raise` and `handle`. Type exn mediates between packets and ML values.

During evaluation, exception packets propagate under the call-by-value rule. If expression E returns an exception packet then that is the result of the applica-

[3] This restriction relates to imperative polymorphism; see Section 8.3.

tion $f(E)$, for any function f. Thus `f (raise` Ex`)` is equivalent to `raise` Ex. Incidentally, `raise` itself propagates exceptions, and so

```
raise (Badvalue (raise Failure))
```

raises exception *Failure*.

Expressions in ML are evaluated from left to right. If E_1 returns a packet then that is the result of the pair (E_1, E_2); expression E_2 is not evaluated at all. If E_1 returns a normal value and E_2 returns a packet, then that packet is the result of the pair. The evaluation order matters when E_1 and E_2 raise different exceptions.

The evaluation order is also visible in conditional expressions:

`if` E `then` E_1 `else` E_2

If E evaluates to $true$ then only E_1 is evaluated. Its result, whether normal or not, becomes that of the conditional. Similarly, if E evaluates to $false$ then only E_2 is evaluated. There is a third possibility. If the test E raises an exception then that is the result of the conditional.

Finally, consider the `let` expression

`let val` P `=` E_1 `in` E_2 `end`

If E_1 evaluates to an exception packet then so does the entire `let` expression.

Exception packets are not propagated by testing. The ML system efficiently jumps to the correct exception handler if there is one, otherwise terminating execution.

Standard exceptions. Failure of pattern-matching may raise the built-in exceptions $Match$ or $Bind$. A function raises exception $Match$ when applied to an argument matching none of its patterns. If a `case` expression has no pattern that matches, it also raises exception $Match$. The ML compiler warns in advance of this possibility when it encounters non-exhaustive patterns (not covering all values of the type).

Because many functions can raise $Match$, this exception conveys little information. When coding a function, have it reject incorrect arguments by raising a suitable exception explicitly; a final case can catch any values that fail to match the other patterns. Some programmers declare a new exception for every function, but having too many exceptions leads to clutter. The standard library follows a middle course, declaring exceptions for whole classes of errors. Here are some examples.

- $Overflow$ is raised for arithmetic operations whose result is out of range.

- *Div* is raised for division by zero.
- *Domain* is raised for errors involving the functions of structure *Math*, such as the square root or logarithm of a negative number.
- *Chr* is raised by *chr*(*k*) if *k* is an invalid character code.
- *Subscript* is raised if an index is out of range. Array, string and list operations can raise *Subscript*.
- *Size* is raised upon an attempt to create an array, string or list of negative or grossly excessive size.
- *Fail* is raised for miscellaneous errors; it carries an error message as a string.

The library structure *List* declares the exception *Empty*. Functions *hd* and *tl* raise this exception if they are applied to the empty list:

```
exception Empty;
fun hd (x::_) = x
  | hd []     = raise Empty;
fun tl (_::xs) = xs
  | tl []     = raise Empty;
```

Less trivial is the library function to return the *n*th element in a list, counting from 0:

```
exception Subscript;
fun nth(x::_,  0) = x
  | nth(x::xs, n) = if n>0 then nth(xs,n-1)
                           else raise Subscript
  | nth _         = raise Subscript;
```

Evaluating *nth*(*l*, *n*) raises exception *Subscript* if $n < 0$ or if the list *l* has no *n*th element. In the latter case, the exception propagates up the recursive calls to *nth*.

```
nth(explode "At the pit of Acheron", 5);
> #"e" : char
nth([1,2], 2);
> Exception: Subscript
```

The declaration `val` *P* = *E* raises exception *Bind* if the value of *E* does not match pattern *P*. This is usually poor style (but see Figure 8.4 on page 348). If there is any possibility that the value will not match the pattern, consider the alternatives explicitly using a `case` expression:

```
case E of P => ··· | P2 => ···
```

4.8 Handling exceptions

An exception handler tests whether the result of an expression is an exception packet. If so, the packet's contents — a value of type *exn* — may be examined by cases. An expression that has an exception handler resembles the `case` construct:

$$E \texttt{ handle } P_1 \texttt{ => } E_1 \texttt{ | } \cdots \texttt{ | } P_n \texttt{ => } E_n$$

If E returns a normal value, then the handler simply passes this value on. On the other hand, if E returns a packet then its contents are matched against the patterns. If P_i is the first pattern to match then the result is the value of E_i, for $i = 1, \ldots, n$.

There is one major difference from `case`. If no pattern matches, then the handler propagates the exception packet rather than raising exception $Match$. A typical handler does not consider every possible exception.

In Section 3.7 we considered the problem of making change. The greedy algorithm presented there (function *change*) could not express 16 using 5 and 2 because it always took the largest coin. Another function, *allChange*, treated such cases by returning the list of all possible results.

Using exceptions, we can easily code a backtracking algorithm. We declare the exception *Change* and raise it in two situations: if we run out of coins with a non-zero *amount* or if we cause *amount* to become negative. We always try the largest coin, undoing the choice if it goes wrong. The exception handler always undoes the most recent choice; recursion makes sure of that.

```
exception Change;
fun backChange (coinvals, 0)          = []
  | backChange ([],  amount)          = raise Change
  | backChange (c::coinvals, amount) =
      if amount<0 then raise Change
      else c :: backChange(c::coinvals, amount-c)
           handle Change => backChange(coinvals, amount);
> val change = fn : int list * int -> int list
```

Unlike *allChange*, this function returns at most one solution. Let us compare the two functions by redoing the examples from Section 3.7:

```
backChange([], [10,2], 27);
> Exception: Change
backChange([5,2], 16);
> [5, 5, 2, 2, 2] : int list
backChange(gbcoins, 16);
```

```
> [10, 5, 1] : int list
```

There are none, two and 25 solutions; we get at most one of them. Similar examples of exception handling occur later in the book, as we tackle problems like parsing and unification. Lazy lists (Section 5.19) are an alternative to exception handling. Multiple solutions can be computed on demand.

Pitfalls in exception handling. An exception handler must be written with care, as with other forms of pattern matching. Never misspell an exception name; it will be taken as a variable and match all exceptions.

Be careful to give exception handlers the correct scope. In the expression

```
if E then E1 else E2 handle ···
```

the handler will only detect exceptions raised by E_2. Enclosing the conditional expression in parentheses brings it entirely within the scope of the handler. Similarly, in

```
case E of P1 => E1 | ··· | Pn => En handle ···
```

the handler will only detect exceptions raised by E_n.

Exception handlers in `case` expressions can be syntactically ambiguous. Omitting the parentheses here would make the second line of the `case` expression become part of the handler:

```
case f u of [x] => (g x  handle _ => x)
          | xs  => g u
```

4.9 *Objections to exceptions*

Exceptions can be a clumsy alternative to pattern-matching, as in this function for computing the length of a list:

```
fun len l =  1 + len(tl l) handle _ => 0;
> val len = fn : 'a list -> int
```

Writing ▲ for the exception packet, the evaluation of $len[1]$ goes like this:

$$
\begin{aligned}
len[1] &\Rightarrow 1 + len(tl[1])\ \texttt{handle}\ \_ \Rightarrow 0 \\
&\Rightarrow 1 + len[]\ \texttt{handle}\ \_ \Rightarrow 0 \\
&\Rightarrow 1 + (1 + len(tl[])\ \texttt{handle}\ \_ \Rightarrow 0)\ \texttt{handle}\ \_ \Rightarrow 0 \\
&\Rightarrow 1 + (1 + len\blacktriangle\ \texttt{handle}\ \_ \Rightarrow 0)\ \texttt{handle}\ \_ \Rightarrow 0 \\
&\Rightarrow 1 + (1 + \blacktriangle\ \texttt{handle}\ \_ \Rightarrow 0)\ \texttt{handle}\ \_ \Rightarrow 0 \\
&\Rightarrow 1 + (\blacktriangle\ \texttt{handle}\ \_ \Rightarrow 0)\ \texttt{handle}\ \_ \Rightarrow 0 \\
&\Rightarrow 1 + 0\ \texttt{handle}\ \_ \Rightarrow 0 \\
&\Rightarrow 1
\end{aligned}
$$

This evaluation is more complicated than one for the obvious length function defined by pattern-matching. Test for different cases in advance, if possible, rather than trying them willy-nilly by exception handling.

Most proponents of lazy evaluation object to exception handling. Exceptions complicate the theory and can be abused, as we have just seen. The conflict is deeper. Exceptions are propagated under the call-by-value rule, while lazy evaluation follows call-by-need.

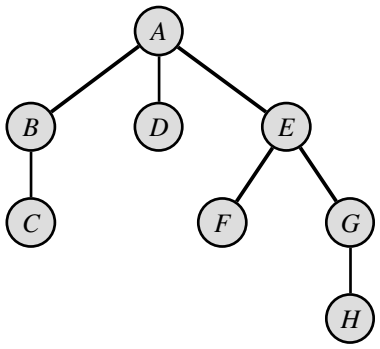
ML includes assignments and other commands, and exceptions can be hazardous in imperative programming. It is difficult to write correct programs when execution can be interrupted in arbitrary places. Restricted to the functional parts of a program, exceptions can be understood as dividing the value space into ordinary values and exception packets. They are not strictly necessary in a programming language and could be abused, but they can also promote clarity and efficiency.

Exercise 4.11 Type *exn* does not admit the ML equality operator. Is this restriction justified?

Exercise 4.12 Describe a computational problem from your experience where exception handling would be appropriate. Write the skeleton of an ML program to solve this problem. Include the exception declarations and describe where exceptions would be raised and handled.

Trees

A ***tree*** is a branching structure consisting of ***nodes*** with ***branches*** leading to subtrees. Nodes may carry values, called ***labels***. Despite the arboreal terminology, trees are usually drawn upside down:



The node labelled A is the ***root*** of the tree, while nodes C, D, F and H (which have no subtrees) are its ***leaves***.

The type of a node determines the type of its label and how many subtrees it may have. The type of a tree determines the types of its nodes. Two types of tree are especially important. The first has labelled nodes, each with one branch, terminated by an unlabelled leaf. Such trees are simply lists. The second type of tree differs from lists in that each labelled node has two branches instead of one. These are called ***binary trees***.

When functional programmers work with lists, they can draw on a body of techniques and a library of functions. When they work with trees, they usually make all their own arrangements. This is a pity, for binary trees are ideal for many applications. The following sections apply them to efficient table lookup, arrays and priority queues. We develop a library of polymorphic functions for binary trees.

4.10 *A type for binary trees*

A binary tree has branch nodes, each with a label and two subtrees. Its leaves are unlabelled. To define binary trees in ML requires a recursive `datatype` declaration:

```
datatype 'a tree = Lf
                 | Br of 'a * 'a tree * 'a tree;
```

Recursive datatypes are understood exactly like non-recursive ones. Type τ *tree* consists of all the values that can be made by Lf and Br. There is at least one τ *tree*, namely Lf; and given two trees and a label of type τ, we can make another tree. Thus Lf is the base case of the recursion.

Here is a tree labelled with strings:

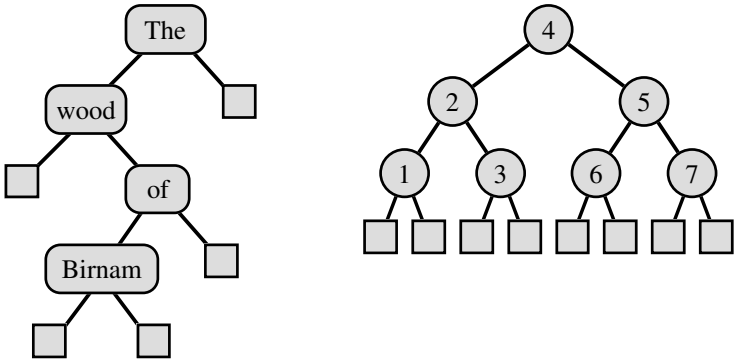
```
val birnam =
  Br("The", Br("wood", Lf,
                       Br("of", Br("Birnam", Lf, Lf),
                                Lf)),
            Lf);
> val birnam = Br ("The", ..., Lf) : string tree
```

Here are some trees labelled with integers. Note how trees can be combined to form bigger ones.

```
val tree2 = Br(2, Br(1,Lf,Lf), Br(3,Lf,Lf));
> val tree2 = Br (2, Br (1, Lf, Lf),
>                    Br (3, Lf, Lf)) : int tree
val tree5 = Br(5, Br(6,Lf,Lf), Br(7,Lf,Lf));
> val tree5 = Br (5, Br (6, Lf, Lf),
```

```
>                     Br (7, Lf, Lf)) : int tree
val tree4 = Br(4, tree2, tree5);
> val tree4 =
> Br (4, Br (2, Br (1, Lf, Lf),
>               Br (3, Lf, Lf)),
>        Br (5, Br (6, Lf, Lf),
>               Br (7, Lf, Lf))) : int tree
```

Trees *birnam* and *tree*4 can be pictured as follows:



Leaves are shown as squares above, but will henceforth be omitted.

Tree operations are expressed by recursive functions with pattern-matching. The polymorphic function *size* returns the number of labels in a tree:

```
fun size Lf            = 0
  | size (Br(v,t1,t2)) = 1 + size t1 + size t2;
> val size = fn : 'a tree -> int
size birnam;
> 4 : int
size tree4;
> 7 : int
```

Another measure of the size of a tree is its *depth*: the length of the longest path from the root to a leaf.

```
fun depth Lf            = 0
  | depth (Br(v,t1,t2)) = 1 + Int.max(depth t1, depth t2);
> val depth = fn : 'a tree -> int
depth birnam;
> 4 : int
depth tree4;
> 3 : int
```

Observe that *birnam* is rather deep for its size while *tree*4 is as shallow as

possible. If t is a binary tree then

$$size(t) \leq 2^{depth(t)} - 1$$

If t satisfies $size(t) = 2^{depth(t)} - 1$ then it is a ***complete binary tree***. For instance, $tree4$ is a complete binary tree of depth 3.

Informally speaking, a binary tree is ***balanced*** if at each node, both subtrees are of similar size. This concept can be made precise in various ways. The cost of reaching a node in a tree is proportional to its depth — for a balanced tree, to the logarithm of the number of elements. A complete binary tree of depth 10 contains 1,023 branch nodes, all reachable in at most nine steps. A tree of depth 20 can contain over 10^6 elements. Balanced trees permit efficient access to large quantities of data.

Calling $comptree(1, n)$ creates a complete binary tree of depth n, labelling the nodes from 1 to 2^n:

```
fun comptree (k,n) =
      if n=0 then Lf
             else Br(k, comptree(2*k,   n-1),
                        comptree(2*k+1, n-1));
> val comptree = fn : int * int -> int tree
comptree (1,3);
> Br (1, Br (2, Br (4, Lf, Lf),
>               Br (5, Lf, Lf)),
>        Br (3, Br (6, Lf, Lf),
>               Br (7, Lf, Lf))) : int tree
```

A function over trees, *reflect* forms the mirror image of a tree by exchanging left and right subtrees all the way down:

```
fun reflect Lf           = Lf
  | reflect (Br(v,t1,t2)) = Br(v, reflect t2, reflect t1);
> val reflect = fn : 'a tree -> 'a tree
reflect tree4;
> Br (4, Br (5, Br (7, Lf, Lf),
>               Br (6, Lf, Lf)),
>        Br (2, Br (3, Lf, Lf),
>               Br (1, Lf, Lf))) : int tree
```

Exercise 4.13 Write a function $compsame(x, n)$ to construct a complete binary tree of depth n, labelling all nodes with x. How efficient is your function?

Exercise 4.14 A binary tree is ***balanced*** (by size) if each node $Br(x, t_1, t_2)$ satisfies $|size(t_1) - size(t_2)| \leq 1$. The obvious recursive function to test whether

a tree is balanced applies *size* at every subtree, performing much redundant computation. Write an efficient function to test whether a tree is balanced.

Exercise 4.15 Write a function that determines whether two arbitrary trees t and u satisfy $t = reflect(u)$. The function should not build any new trees, so it should not call *reflect* or Br, although it may use Br in patterns.

Exercise 4.16 Lists need not have been built into ML. Give a `datatype` declaration of a type equivalent to $\alpha\ list$.

Exercise 4.17 Declare a datatype $(\alpha, \beta)ltree$ of labelled binary trees, where branch nodes carry a label of type α and leaves carry a label of type β.

Exercise 4.18 Declare a datatype of trees where each branch node may have any finite number of branches. (Hint: use *list*.)

4.11 *Enumerating the contents of a tree*

Consider the problem of making a list of a tree's labels. The labels must be arranged in some order. Three well-known orders, ***preorder***, ***inorder*** and ***postorder***, can be described by a recursive function over trees. Given a branch node, each puts the labels of the left subtree before those of the right; the orders differ only in the position of the label.

A preorder list places the label first:

```
fun preorder Lf            = []
  | preorder (Br(v,t1,t2)) = [v] @ preorder t1 @ preorder t2;
> val preorder = fn : 'a tree -> 'a list
preorder birnam;
> ["The", "wood", "of", "Birnam"] : string list
preorder tree4;
> [4, 2, 1, 3, 5, 6, 7] : int list
```

An inorder list places the label between the labels from the left and right subtrees, giving a strict left-to-right traversal:

```
fun inorder Lf            = []
  | inorder (Br(v,t1,t2)) = inorder t1 @ [v] @ inorder t2;
> val inorder = fn : 'a tree -> 'a list
inorder birnam;
> ["wood", "Birnam", "of", "The"] : string list
inorder tree4;
> [1, 2, 3, 4, 6, 5, 7] : int list
```

A postorder list places the label last:

```
fun postorder Lf            = []
  | postorder (Br(v,t1,t2)) = postorder t1 @ postorder t2 @ [v];
> val postorder = fn : 'a tree -> 'a list
postorder birnam;
> ["Birnam", "of", "wood", "The"] : string list
postorder tree4;
> [1, 3, 2, 6, 7, 5, 4] : int list
```

Although these functions are clear, they take quadratic time on badly unbalanced trees. The culprit is the appending (@) of long lists. It can be eliminated using an extra argument vs to accumulate the labels.The following versions perform exactly one cons (::) operation per branch node:

```
fun preord (Lf, vs)           = vs
  | preord (Br(v,t1,t2), vs)  = v :: preord(t1, preord(t2,vs));

fun inord (Lf, vs)            = vs
  | inord (Br(v,t1,t2), vs)   = inord(t1, v::inord(t2,vs));

fun postord (Lf, vs)          = vs
  | postord (Br(v,t1,t2), vs) = postord(t1, postord(t2,v::vs));
```

These definitions are worth study; many functions are declared similarly. For instance, logical terms are essentially trees. The list of all the constants in a term can be built as above.

Exercise 4.19 Describe how $inorder(birnam)$ and $inord(birnam, [])$ are evaluated, reporting how many cons operations are performed.

Exercise 4.20 Complete the following equations and explain why they are correct.

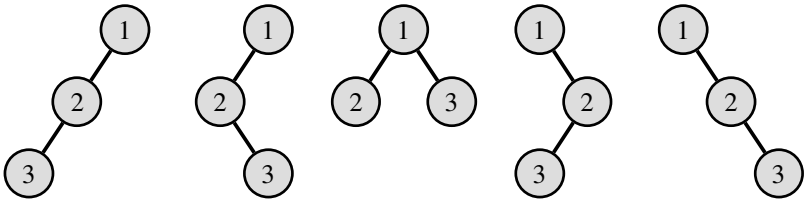
$$preorder(reflect(t)) = ?$$
$$inorder(reflect(t)) = ?$$
$$postorder(reflect(t)) = ?$$

4.12 *Building a tree from a list*

Now consider converting a list of labels to a tree. The concepts of preorder, inorder and postorder apply as well to this inverse operation. Even within a fixed order, one list can be converted to many different trees. The equation

$$preorder(t) = [1, 2, 3]$$

has five solutions in t:



Only one of these trees is balanced. To construct balanced trees, divide the list of labels roughly in half. The subtrees may differ in size (number of nodes) by at most 1.

To make a balanced tree from a preorder list of labels, the first label is attached to the root of the tree:

```
fun balpre  []    = Lf
  | balpre(x::xs) =
    let val k = length xs div 2
    in Br(x, balpre(List.take(xs,k)), balpre(List.drop(xs,k)))
    end;
> val balpre = fn : 'a list -> 'a tree
```

This function is an inverse of *preorder*.

```
balpre(explode "Macbeth");
> Br (#"M", Br (#"a", Br (#"c", Lf, Lf),
>                     Br (#"b", Lf, Lf)),
>           Br (#"e", Br (#"t", Lf, Lf),
>                     Br (#"h", Lf, Lf))) : char tree
implode(preorder it);
> "Macbeth" : string
```

To make a balanced tree from an inorder list, the label is taken from the middle. This resembles the top-down merge sort of Section 3.21:

```
fun balin [] = Lf
  | balin xs =
      let val k      = length xs div 2
          val y::ys  = List.drop(xs,k)
      in  Br(y, balin (List.take(xs,k)), balin ys)
      end;
> val balin = fn : 'a list -> 'a tree
```

This function is an inverse of *inorder*.

```
balin(explode "Macbeth");
> Br (#"b", Br (#"a", Br (#"M", Lf, Lf),
>                     Br (#"c", Lf, Lf)),
>           Br (#"t", Br (#"e", Lf, Lf),
>                     Br (#"h", Lf, Lf))) : char tree
implode(inorder it);
```

```
> "Macbeth" : string
```

Exercise 4.21 Write a function to convert a postorder list of labels to a balanced tree.

Exercise 4.22 The function *balpre* constructs one tree from a preorder list of labels. Write a function that, given a list of labels, constructs the list of all trees that have those labels in preorder.

4.13 *A structure for binary trees*

As usual in this book, we have been following an imaginary ML session in which we typed in the tree functions one at a time. Now we ought to collect the most important of those functions into a structure, called *Tree*. We really must do so, because one of our functions (*size*) clashes with a built-in function. One reason for using structures is to prevent such name clashes.

We shall, however, leave the `datatype` declaration of *tree* outside the structure. If it were inside, we should be forced to refer to the constructors by *Tree*`.`*Lf* and *Tree*`.`*Br*, which would make our patterns unreadable.[4] Thus, in the sequel, imagine that we have made the following declarations:

```
datatype 'a tree = Lf
                 | Br of 'a * 'a tree * 'a tree;

structure Tree =
  struct
  fun size Lf             = 0
    | size (Br(v,t1,t2)) = 1 + size t1 + size t2;

  fun depth   ...
  fun reflect ...
  fun preord  ...
  fun inord   ...
  fun postord ...
  fun balpre  ...
  fun balin   ...
  fun balpost ...
  end;
```

[4] There is a means — the `open` declaration — of making a structure's components available directly by names such as *Lf* and *Br*. Opening the whole of *Tree* would defeat the purpose of declaring the structure in the first place. Section 7.14 will discuss various ways of dealing with compound names.

Exercise 4.23 Let us put the `datatype` declaration inside the structure, then make the constructors available outside using these declarations:

```
val Lf = Tree.Lf;
val Br = Tree.Br;
```

What is wrong with this idea?

Tree-based data structures

Computer programming consists of implementing a desired set of high level operations in terms of a given set of primitive operations. Those high level operations become the primitive operations for coding the next level. Layered network protocols are a striking example of this principle, which can be seen in any modular system design.

Consider the simpler setting of data structure design. The task is to implement the desired data structure in terms of the programming language's primitive data structures. Here a data structure is described not by its internal representation, but by the operations it supports. To implement a new data structure, we must know the precise set of operations desired.

ML gives us two great advantages. Its primitive features easily describe trees; we do not have to worry about references or storage allocation. And we can describe the desired set of operations by a signature.

Typical operations on collections of data include inserting an item, looking up an item, removing an item or merging two collections. We shall consider three data structures that can be represented by trees:

- Dictionaries, where items are identified by name.
- Arrays, where items are identified by an integer.
- Priority queues, where items are identified by priority: only the highest priority item can be removed.

Unlike the data structures described in most texts, ours will be purely functional. Inserting or removing an item will not alter the collection, but create a new collection. It may come as a surprise to hear that this can be done efficiently.

4.14 *Dictionaries*

A ***dictionary*** is a collection of items, each identified by a unique key (typically a string). It supports the following operations:

- ***Lookup*** a key and return the item associated with it.
- ***Insert*** a new key (not already present) and an associated item.

- ***Update*** the item associated with an existing key (insert it if the key is not already present).

We can make this description more precise by writing an ML signature:

```
signature DICTIONARY =
  sig
  type key
  type 'a t
  exception E of key
  val empty  : 'a t
  val lookup  : 'a t * key -> 'a
  val insert  : 'a t * key * 'a -> 'a t
  val update  : 'a t * key * 'a -> 'a t
  end;
```

The signature has more than the three operations described above. What are the other things for?

- key is the type of search keys.[5]
- $\alpha\ t$ is the type of dictionaries whose stored items have type α.
- E is the exception raised when errors occur. Lookup fails if the key is not found, while insert fails if the key is already there. The exception carries the rejected key.
- $empty$ is the empty dictionary.

A structure matching this signature must declare the dictionary operations with appropriate types. For instance, the function $lookup$ takes a dictionary and a key, and returns an item. Nothing in signature *DICTIONARY* indicates that trees are involved. We may adopt any representation.

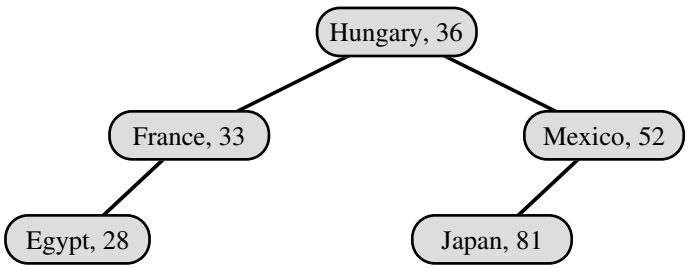
A binary search tree can implement a dictionary. A reasonably balanced tree (Figure 4.1) is considerably more efficient than an association list of (key, $item$) pairs. The time required to search for a key among n items is order n for lists and order $\log n$ for binary search trees. The time required to update the tree is also of order $\log n$. An association list can be updated in constant time, but this does not compensate for the long search time.

In the worst case, binary search trees are actually slower than association lists. A series of updates can create a highly unbalanced tree. Search and update can take up to n steps for a tree of n items.

The keys in an association list may have any type that admits equality, but the keys in a binary search tree must come with a linear ordering. Strings, with

[5] Here it is $string$; Section 7.10 will generalize binary search trees to take the type of search keys as a parameter.

Figure 4.1 *A balanced binary search tree*



alphabetic ordering, are an obvious choice. Each branch node of the tree carries a $(string, item)$ pair; its left subtree holds only lesser strings; the right subtree holds only greater strings. The inorder list of labels puts the strings in alphabetic order.

Unlike tree operations that might be coded in Pascal, the ***update*** and ***insert*** operations do not modify the current tree. Instead, they create a new tree. This is less wasteful than it sounds: the new tree shares most of its storage with the existing tree.

Figure 4.2 presents the structure $Dict$, which is an instance of signature *DICTIONARY*. It starts by declaring types key and $\alpha\ t$, and exception E. The type declarations are only abbreviations, but they must be present in order to satisfy the signature.

Lookup in a binary search tree is simple. At a branch node, look left if the item being sought is lesser than the current label, and right if it is greater. If the item is not found, the function raises exception E. Observe the use of the datatype $order$.

Insertion of a $(string, item)$ pair involves locating the correct position for the $string$, then inserting the $item$. As with $lookup$, comparing the string with the current label determines whether to look left or right. Here the result is a new branch node; one subtree is updated and the other borrowed from the original tree. If the string is found in the tree, an exception results.

In effect, $insert$ copies the path from the root of the tree to the new node. Function $update$ is identical apart from its result if the string is found in the tree.

The exception in $lookup$ is easily eliminated because that function is iterative. It could return a result of type $\alpha\ option$, namely $SOME\,x$ if the key is

Figure 4.2 *A structure for dictionaries as binary search trees*

```
structure Dict : DICTIONARY =
  struct

  type key  = string;
  type 'a t = (key * 'a) tree;

  exception E of key;

  val empty = Lf;

  fun lookup (Lf, b)                = raise E b
    | lookup (Br ((a,x),t1,t2), b) =
        (case String.compare(a,b) of
              GREATER => lookup(t1, b)
            | EQUAL   => x
            | LESS    => lookup(t2, b));

  fun insert (Lf, b, y)               = Br((b,y), Lf, Lf)
    | insert (Br((a,x),t1,t2), b, y) =
        (case String.compare(a,b) of
              GREATER => Br ((a,x),  insert(t1,b,y),  t2)
            | EQUAL   => raise E b
            | LESS    => Br ((a,x),  t1,  insert(t2,b,y)));

  fun update (Lf, b, y)               = Br((b,y), Lf, Lf)
    | update (Br((a,x),t1,t2), b, y) =
        (case String.compare(a,b) of
              GREATER => Br ((a,x),  update(t1,b,y),  t2)
            | EQUAL   => Br ((a,y),  t1,  t2)
            | LESS    => Br ((a,x),  t1,  update(t2,b,y)));

  end;
```

found and *NONE* otherwise. The exception in *insert* is another matter: since the recursive calls construct a new tree, returning *SOME t* or *NONE* would be cumbersome. Function *insert* could call *lookup* and *update*, eliminating the exception but doubling the number of comparisons.

Binary search trees are built from the empty tree (*Lf*) by repeated updates or inserts. We construct a tree *ctree*1 containing France and Egypt:

```
Dict.insert(Lf, "France", 33);
> Br (("France", 33), Lf, Lf) : int Dict.t
val ctree1 = Dict.insert(it, "Egypt", 20);
> val ctree1 = Br (("France", 33),
>                  Br (("Egypt", 20), Lf, Lf),
>                  Lf) : int Dict.t
```

We insert Hungary and Mexico:

```
Dict.insert(ctree1, "Hungary", 36);
> Br (("France", 33), Br (("Egypt", 20), Lf, Lf),
>     Br (("Hungary", 36), Lf, Lf)) : int Dict.t
Dict.insert(it, "Mexico", 52);
> Br (("France", 33), Br (("Egypt", 20), Lf, Lf),
>     Br (("Hungary", 36), Lf,
>         Br (("Mexico", 52), Lf, Lf))) : int Dict.t
```

By inserting Japan, we create the tree *ctree*2 consisting of 5 items.

```
val ctree2 = Dict.update(it, "Japan", 81);
> val ctree2 =
> Br (("France", 33), Br (("Egypt", 20), Lf, Lf),
>     Br (("Hungary", 36), Lf,
>         Br (("Mexico", 52),
>             Br (("Japan", 81), Lf, Lf),
>             Lf))) : int Dict.t
```

Note that *ctree*1 still exists, even though *ctree*2 has been constructed from it.

```
Dict.lookup(ctree1, "France");
> 33 : int
Dict.lookup(ctree2, "Mexico");
> 52 : int
Dict.lookup(ctree1, "Mexico");
> Exception: E
```

Inserting items at random can create unbalanced trees. If most of the insertions occur first, followed by many lookups, then it pays to balance the tree before the lookups. Since a binary search tree corresponds to a sorted inorder list, it can be balanced by converting it to inorder, then constructing a new tree:

```
Tree.inord (ctree2, []);
```

```
> [("Egypt", 20), ("France", 33), ("Hungary", 36),
>  ("Japan", 81), ("Mexico", 52)] : (Dict.key * int) list
val baltree = Tree.balin it;
> val baltree =
> Br (("Hungary", 36),
>     Br (("France", 33), Br (("Egypt", 20), Lf, Lf), Lf),
>     Br (("Mexico", 52), Br (("Japan", 81), Lf, Lf), Lf))
> : (Dict.key * int) tree
```

This is the tree illustrated in Figure 4.1.

Balanced tree algorithms. The balancing approach outlined above is limited. Using *inord* and *balin* relies on the internal representation of dictionaries as trees; the result type is now *tree* instead of $Dict.t$. Worse, the user must decide when to perform balancing.

There exist several forms of search trees that maintain balance automatically, typically by rearranging elements during updates or lookups. Adams (1993) presents ML code for self-balancing binary search trees. Reade (1992) presents a functional treatment of 2-3 trees, where each branch node may have two or three children.

Exercise 4.24 Give four examples of a binary search tree whose depth equals 5 and that contains only the 5 labels of $ctree2$. For each tree, show a sequence of insertions that creates it.

Exercise 4.25 Write a new version of structure $Dict$ where a dictionary is represented by a list of $(key, item)$ pairs ordered by the keys.

4.15 *Functional and flexible arrays*

What is an array? To most programmers, an array is a block of storage cells, indexed by integers, that can be updated. Conventional programming skill mainly involves using arrays effectively. Since most arrays are scanned sequentially, the functional programmer can use lists instead. But many applications — hash tables and histograms are perhaps the simplest — require random access.

In essence, an array is a mapping defined on a finite range of the integers. The element associated with the integer k is written $A[k]$. Conventionally, an array is modified by the assignment command

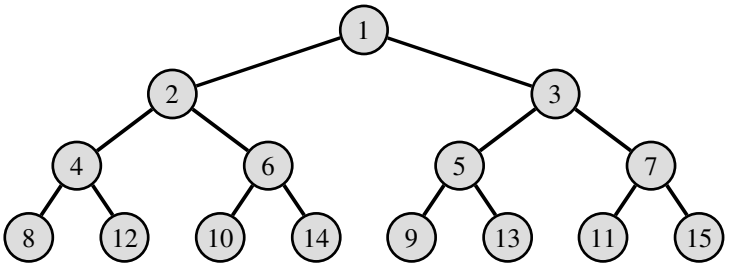
$$A[k] := x,$$

changing the machine state such that $A[k] = x$. The previous contents of $A[k]$ are lost. Updating in place is highly efficient, both in time and space, but it is hard to reconcile with functional programming.

A functional array provides a mapping from integers to elements, with an update operation that creates a new array

$$B = update(A, k, x)$$

such that $B[k] = x$ and $B[i] = A[i]$ for all $i \neq k$. The array A continues to exist and additional arrays can be created from it. Functional arrays can be implemented by binary trees. The position of subscript k in the tree is determined by starting at the root and repeatedly dividing k by 2 until it is reduced to 1. Each time the remainder equals 0, move to the left subtree; if the remainder equals 1, move to the right. For instance, subscript 12 is reached by left, left, right:



A ***flexible*** array augments the usual lookup and update with operations to insert or delete elements from either end of the array. A program starts with an empty array and inserts elements as required. Gaps are forbidden: element $n+1$ must be defined after element n, for $n > 0$. Let us examine the underlying tree operations, which have been credited to W. Braun.

The lookup function, sub, divides the subscript by 2 until 1 is reached. If the remainder is 0 then the function follows the left subtree, otherwise the right. If it reaches a leaf, it signals error by raising a standard exception.

```
fun sub (Lf, _)          = raise Subscript
  | sub (Br(v,t1,t2), k) =
      if k = 1 then v
      else if k mod 2 = 0
           then sub (t1, k div 2)
           else sub (t2, k div 2);
> val sub = fn : 'a tree * int -> 'a
```

The update function, $update$, also divides the subscript repeatedly by 2. When it reaches 1 it replaces the branch node by another branch with the new label. A leaf may be replaced by a branch, extending the array, provided no intervening nodes have to be generated. This suffices for arrays without gaps.

```
fun update (Lf, k, w) =
```

```
      if k = 1 then Br (w, Lf, Lf)
      else raise Subscript
  | update (Br(v,t1,t2), k, w) =
      if k = 1 then Br (w, t1, t2)
      else if k mod 2 = 0
           then Br (v,  update(t1, k div 2, w),  t2)
           else Br (v,  t1,  update(t2, k div 2, w));
> val update = fn : 'a tree * int * 'a -> 'a tree
```

Calling $delete(ta, n)$ replaces the subtree rooted at position n (if it exists) by a leaf. It resembles sub, but builds a new tree.

```
fun delete (Lf, n)          = raise Subscript
  | delete (Br(v,t1,t2), n) =
      if n = 1 then Lf
      else if n mod 2 = 0
           then Br (v,  delete(t1, n div 2),  t2)
           else Br (v,  t1,  delete(t2, n div 2));
> val delete = fn : 'a tree * int -> 'a tree
```

Letting a flexible array grow and shrink from above is simple. Just store the upper bound with the binary tree and use $update$ and $delete$. But how can we let the array grow and shrink from below? As the lower bound is fixed, this seems to imply shifting all the elements.

Consider extending a tree from below with the element w. The result has w at position 1, replacing the previous element v. Its right subtree (positions 3, 5, ...) is simply the old left subtree (positions 2, 4, ...). By a recursive call, its left subtree has v at position 2 and takes the rest (positions 4, 6, ...) from the old right subtree (positions 3, 5, ...).

```
fun loext (Lf, w)          = Br(w, Lf, Lf)
  | loext (Br(v,t1,t2), w) = Br(w, loext(t2,v), t1);
> val loext = fn : 'a tree * 'a -> 'a tree
```

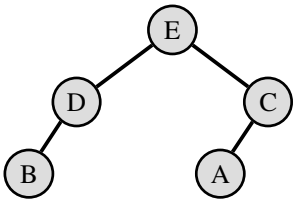
So we can extend a flexible array from below, in logarithmic time. To shorten the array, simply reverse the steps. Attempted deletion from the empty array raises the standard exception $Size$. Trees of the form $Br\,(\_, Lf, Br\ \_)$ need not be considered: at every node we have $L - 1 \leq R \leq L$, where L is the size of the left subtree and R is the size of the right subtree.

```
fun lorem Lf                              = raise Size
  | lorem (Br(_,Lf,Lf))                   = Lf
  | lorem (Br(_, t1 as Br(v,_,_), t2)) = Br(v, t2, lorem t1);
> val lorem = fn : 'a tree -> 'a tree
```

It is time for a demonstration. By repeatedly applying $loext$ to a leaf, we build an array of the letters A to E in reverse order.

```
loext(Lf,"A");
> Br ("A", Lf, Lf) : string tree
loext(it,"B");
> Br ("B", Br ("A", Lf, Lf), Lf) : string tree
loext(it,"C");
> Br ("C", Br ("B", Lf, Lf), Br ("A", Lf, Lf))
> : string tree
loext(it,"D");
> Br ("D", Br ("C", Br ("A", Lf, Lf), Lf),
>          Br ("B", Lf, Lf)) : string tree
val tlet = loext(it,"E");
> val tlet = Br ("E", Br ("D", Br ("B", Lf, Lf), Lf),
>                     Br ("C", Br ("A", Lf, Lf), Lf))
> : string tree
```

The tree *tlet* looks like this:



Updating elements of *tlet* does not affect that array, but creates a new array:

```
val tdag = update(update (tlet, 5, "Amen"),
                  2, "dagger");
> val tdag =
> Br ("E", Br ("dagger", Br ("B", Lf, Lf), Lf),
>                        Br ("C", Br ("Amen", Lf, Lf), Lf))
> : string tree
sub(tdag,5);
> "Amen" : string
sub(tlet,5);
> "A" : string
```

The binary tree remains balanced after each operation. Lookup and update with subscript k take order $\log k$ steps, the best possible time complexity for any data structure of unbounded size. Access to a million-element array will be twice as slow as access to a thousand-element array.

The standard library structure *Array* provides imperative arrays. They will be used in Chapter 8 to implement functional (but not flexible) arrays. That implementation gives fast, constant access time — if the array is used in an imperative style. Imperative arrays are so prevalent that functional applications require some imagination.

Here is a signature for flexible arrays. It is based on the structure *Array*,

Figure 4.3 *Structures for Braun trees and flexible arrays*

```
structure Braun =
  struct
  fun sub    ...
  fun update ...
  fun delete ...
  fun loext  ...
  fun lorem  ...
  end;


structure Flex : FLEXARRAY =
  struct
  datatype 'a array = Array of 'a tree * int;

  val empty = Array(Lf,0);

  fun length (Array(_,n)) = n;

  fun sub (Array(t,n), k) =
      if 0<=k andalso k<n then Braun.sub(t,k+1)
      else raise Subscript;

  fun update (Array(t,n), k, w) =
      if 0<=k andalso k<n then Array(Braun.update(t,k+1,w), n)
      else raise Subscript;

  fun loext (Array(t,n), w) = Array(Braun.loext(t,w), n+1);

  fun lorem (Array(t,n)) =
      if n>0 then Array(Braun.lorem t, n-1)
      else raise Size;

  fun hiext (Array(t,n), w) = Array(Braun.update(t,n+1,w), n+1);

  fun hirem (Array(t,n)) =
      if n>0 then Array(Braun.delete(t,n) , n-1)
      else raise Size;

  end;
```

but includes extension and removal from below ($loext$, $lorem$) and from above ($hiext$, $hirem$).

```
signature FLEXARRAY =
  sig
  type 'a array
  val empty : 'a array
  val length : 'a array -> int
  val sub    : 'a array * int -> 'a
  val update : 'a array * int * 'a -> 'a array
  val loext  : 'a array * 'a -> 'a array
  val lorem  : 'a array -> 'a array
  val hiext  : 'a array * 'a -> 'a array
  val hirem  : 'a array -> 'a array
  end;
```

Figure 4.3 presents the implementation. The basic tree manipulation functions are packaged as the structure $Braun$, to prevent name clashes with the analogous functions in structure $Flex$. Incidentally, $Braun$ subscripts range from 1 to n while $Flex$ subscripts range from 0 to $n-1$. The former arises from the representation, the latter from ML convention.

Structure $Flex$ represents a flexible array by a binary tree paired with an integer, its size. It might have declared type $array$ as a type abbreviation:

```
type 'a array = 'a tree * int;
```

Instead, it declares $array$ as a datatype with one constructor. Such a datatype costs nothing at run-time: the constructor occupies no space. The new type distinguishes flexible arrays from accidental pairs of a tree with an integer, as in the call to $Braun.sub$. The constructor is hidden outside the structure, preventing users from taking apart the flexible array.

Further reading. Dijkstra (1976), a classic work on imperative programming, introduces flexible arrays and many other concepts. Hoogerwoord (1992) describes flexible arrays in detail, including the operation to extend an array from below. Okasaki (1995) introduces ***random access lists***, which provide logarithmic time array access as well as constant time list operations (cons, head, tail). A random access list is represented by a list of complete binary trees. The code is presented in ML and is easy to understand.

Exercise 4.26 Write a function to create an array consisting of the element x in subscript positions 1 to n. Do not use $Braun.update$: build the tree directly.

Exercise 4.27 Write a function to convert the array consisting of the elements x_1, x_2, ..., x_n (in subscript positions 1 to n) to a list. Operate directly on the tree, without repeated subscripting.

Exercise 4.28 Implement sparse arrays, which may have large gaps between elements, by allowing empty labels in the tree.

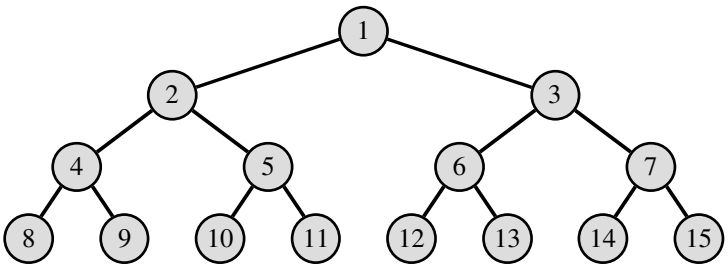
4.16 *Priority queues*

A ***priority queue*** is an ordered collection of items. Items may be inserted in any order, but only the highest priority item may be seen or deleted. Higher priorities traditionally mean lower numerical values, so the basic operations are called $insert$, min and $delmin$.

In simulations, a priority queue selects the next event according to its scheduled time. In Artificial Intelligence, priority queues implement ***best-first search***: attempted solutions to a problem are stored with priorities (assigned by a rating function) and the best attempt is chosen for further search.

If a priority queue is kept as a sorted list, $insert$ takes up to n steps for a queue of n items. This is unacceptably slow. With a binary tree, $insert$ and $delmin$ take order $\log n$ steps. Such a tree, called a ***heap***, underlies the well-known sorting algorithm ***heap sort***. The labels are arranged such that no label is less than a label above it in the tree. This ***heap condition*** puts the labels in no strict order, but does put the least label at the root.

Conventionally, the tree is embedded in an array with the labels indexed as follows:



An n-item heap consists of nodes 1 to n. This indexing scheme always creates a tree of minimum depth. But our functional priority queues are based on the indexing scheme for flexible arrays; it seems more amenable to functional programming, and also assures minimum depth. The resulting program is a hybrid of old and new ideas.

If the heap contains $n - 1$ items, then an insertion fills position n. However, the new item may be too small to go into that position without violating the heap

condition. It may end up higher in the tree, forcing larger items downwards. Function *insert* works on the same principle as *loext*, but maintains the heap condition while inserting the item. Unless the new item w exceeds the current label v, it becomes the new label and v is inserted further down; because v does not exceed any item in the subtrees, neither does w.

```
fun insert(w: real, Lf)        = Br(w, Lf, Lf)
  | insert(w, Br(v, t1, t2)) =
      if w <= v then Br(w, insert(v, t2), t1)
                else Br(v, insert(w, t2), t1);
> val insert = fn : real * real tree -> real tree
```

There seems to be no simple way of reversing the *insert* operation. Deletion must remove the item at the root, as that is the smallest. But deletion from an n-item heap must vacate position n. Item n may be too large to go into the root without violating the heap condition. We need two functions: one to remove item n and one to re-insert its label in a suitable place.

Function *leftrem* works on the same principle as *lorem*, but does not move labels up or down in the heap. It always removes the leftmost item, exchanging subtrees to give the new heap the correct form. It returns the removed item, paired with the new heap:

```
fun leftrem (Br(v,Lf,Lf)) = (v, Lf)
  | leftrem (Br(v,t1,t2))  =
      let val (w, t) = leftrem t1
      in  (w, Br(v,t2,t))  end;
> val leftrem = fn : 'a tree -> 'a * 'a tree
```

Function *siftdown* makes a heap from the displaced item and the two subtrees of the old heap. The item moves down the tree. At each branch node, it follows the smaller of the subtrees' labels. It stops when no subtree has a smaller label. Thanks to the indexing scheme, the cases considered below are the only ones possible. If the left subtree is empty then so is the right; if the right subtree is empty then the left subtree can have only one label.

```
fun siftdown (w:real, Lf, Lf)            = Br(w,Lf,Lf)
  | siftdown (w, t as Br(v,Lf,Lf), Lf) =
      if w <= v then Br(w, t, Lf)
                else Br(v, Br(w,Lf,Lf), Lf)
  | siftdown (w, t1 as Br(v1,p1,q1), t2 as Br(v2,p2,q2)) =
      if w <= v1 andalso w <= v2 then Br(w,t1,t2)
      else if v1 <= v2 then Br(v1, siftdown(w,p1,q1), t2)
          (*v2 < v1*)   else Br(v2, t1, siftdown(w,p2,q2));
> val siftdown = fn
> : real * real tree * real tree -> real tree
```

Now we can perform deletions. Function *delmin* calls *leftrem* to delete and return an item, then *siftdown* to put the item back in some suitable place. Deletion from the empty heap is an error, and the one-element heap is treated separately.

```
  fun delmin Lf             = raise Size
    | delmin (Br(v,Lf,_))   = Lf
    | delmin (Br(v,t1,t2))  =
        let val (w,t) = leftrem t1
        in  siftdown (w,t2,t)  end;
> val delmin = fn : real tree -> real tree
```

Our signature for priority queues specifies the primitives discussed above. It also specifies operations to convert between heaps and lists, and the corresponding sorting function. It specifies *item* as the type of items in the queue; for now this is *real*, but a functor can take any ordered type as a parameter (see Section 7.10).

```
signature PRIORITY_QUEUE =
  sig
  type item
  type t
  val empty    : t
  val null     : t -> bool
  val insert   : item * t -> t
  val min      : t -> item
  val delmin   : t -> t
  val fromList : item list -> t
  val toList   : t -> item list
  val sort     : item list -> item list
  end;
```

Figure 4.4 displays the structure for heaps. It uses the obvious definitions of *empty* and the predicate *null*. The function *min* merely returns the root.

Priority queues easily implement heap sort. We sort a list by converting it to a heap and back again. Function *heapify* converts a list into a heap in a manner reminiscent of top-down merge sort (Section 3.21). This approach, using *siftdown*, builds the heap in linear time; repeated insertions would require order $n \log n$ time. Heap sort's time complexity is optimal: it takes order $n \log n$ time to sort n items in the worst case. In practice, heap sort tends to be slower than other $n \log n$ algorithms. Recall our timing experiments of Chapter 3. Quick sort and merge sort can process 10,000 random numbers in 200 msec or less, but *Heap.sort* takes 500 msec.

Although there are better ways of sorting, heaps make ideal priority queues. To see how heaps work, let us build one and remove some items from it.

```
Heap.fromList [4.0, 2.0, 6.0, 1.0, 5.0, 8.0, 5.0];
```

Figure 4.4 *A structure for priority queues using heaps*

```
structure Heap : PRIORITY_QUEUE =
  struct
  type item = real;
  type t    = item tree;

  val empty = Lf;

  fun null Lf       = true
    | null (Br _) = false;

  fun min (Br(v,_,_)) = v;

  fun insert   ...

  fun leftrem  ...

  fun siftdown ...

  fun delmin   ...

  fun heapify (0, vs)     = (Lf, vs)
    | heapify (n, v::vs) =
        let val (t1, vs1) = heapify (n div 2, vs)
            val (t2, vs2) = heapify ((n-1) div 2, vs1)
        in  (siftdown (v,t1,t2), vs2)  end;

  fun fromList vs = #1 (heapify (length vs, vs));

  fun toList (t as Br(v,_,_)) = v :: toList(delmin t)
    | toList Lf                 = [];

  fun sort vs = toList (fromList vs);

  end;
```

```
> Br (1.0, Br (2.0, Br (6.0, Lf, Lf),
>                   Br (4.0, Lf, Lf)),
>           Br (5.0, Br (8.0, Lf, Lf),
>                   Br (5.0, Lf, Lf))) : Heap.t
Heap.delmin it;
> Br (2.0, Br (5.0, Br (8.0, Lf, Lf),
>                   Br (5.0, Lf, Lf)),
>           Br (4.0, Br (6.0, Lf, Lf), Lf)) : Heap.t
Heap.delmin it;
> Br (4.0, Br (6.0, Br (8.0, Lf, Lf), Lf),
>           Br (5.0, Br (5.0, Lf, Lf), Lf)) : Heap.t
```

Observe that the smallest item is removed first. Let us apply *delmin* twice more:

```
Heap.delmin it;
> Br (5.0, Br (5.0, Br (8.0, Lf, Lf), Lf),
>                   Br (6.0, Lf, Lf)) : Heap.t
Heap.delmin it;
> Br (5.0, Br (6.0, Lf, Lf), Br (8.0, Lf, Lf)) : Heap.t
```

ML's response has been indented to emphasize the structure of the binary trees.

Other forms of priority queues. The heaps presented here are sometimes called ***binary*** or ***implicit*** heaps. Algorithms textbooks such as Sedgewick (1988) describe them in detail. Other traditional representations of priority queues can be coded in a functional style. Leftist heaps (Knuth, 1973, page 151) and binomial heaps (Cormen *et al.*, 1990, page 400) are more complicated than binary heaps. However, they allow heaps to be merged in logarithmic time. The merge operation for binomial heaps works by a sort of binary addition. Chris Okasaki, who supplied most of the code for this section, has implemented many other forms of priority queues. Binary heaps appear to be the simplest and the fastest, provided we do not require merge.

Exercise 4.29 Draw diagrams of the heaps created by starting with the empty heap and inserting 4, 2, 6, 1, 5, 8 and 5 (as in the call to *heapoflist* above).

Exercise 4.30 Describe the functional array indexing scheme in terms of the binary notation for subscripts. Do the same for the conventional indexing scheme of heap sort.

Exercise 4.31 Write ML functions for lookup and update on functional arrays, represented by the conventional indexing scheme of heap sort. How do they compare with $Braun.sub$ and $Braun.update$?

A tautology checker

This section introduces elementary theorem proving. We define propositions and functions to convert them into various normal forms, obtaining a tautology checker for Propositional Logic. Rather than using binary trees, we declare a datatype of propositions.

4.17 *Propositional Logic*

Propositional Logic deals with ***propositions*** constructed from atoms a, b, c, ..., by the connectives $\wedge$, $\vee$, $\neg$. A proposition may be

$$\begin{array}{rl} \neg p & \text{a negation, 'not } p\text{'} \\ p \wedge q & \text{a conjunction, '}p \text{ and } q\text{'} \\ p \vee q & \text{a disjunction, '}p \text{ or } q\text{'} \end{array}$$

Propositions resemble boolean expressions and are represented by the datatype *prop*:

```
datatype prop = Atom of string
              | Neg  of prop
              | Conj of prop * prop
              | Disj of prop * prop;
```

The implication $p \rightarrow q$ is equivalent to $(\neg p) \vee q$. Here is a function to construct implications:

```
fun implies(p,q) = Disj(Neg p, q);
> val implies = fn : prop * prop -> prop
```

Our example is based on some important attributes — being rich, landed and saintly:

```
val rich    = Atom "rich"
and landed  = Atom "landed"
and saintly = Atom "saintly";
```

Here are two assumptions about the rich, the landed and the saintly.

- Assumption 1 is $landed \rightarrow rich$: the landed are rich.
- Assumption 2 is $\neg(saintly \wedge rich)$: one cannot be both saintly and rich.

A plausible conclusion is $landed \rightarrow \neg saintly$: the landed are not saintly.

Let us give these assumptions and desired conclusion to ML:

```
val assumption1 = implies(landed, rich)
and assumption2 = Neg(Conj(saintly,rich));
> val assumption1 = Disj (Neg (Atom "landed"),
```

```
>                          Atom "rich")  : prop
> val assumption2 = Neg (Conj (Atom "saintly",
>                              Atom "rich"))  : prop
val concl = implies(landed, Neg saintly);
> val concl = Disj (Neg (Atom "landed"),
>                   Neg (Atom "saintly"))  : prop
```

If the conclusion follows from the assumptions, then the following proposition is a propositional theorem — a ***tautology***. Let us declare it as a goal to be proved:

```
val goal = implies(Conj(assumption1,assumption2), concl);
> val goal =
> Disj (Neg (Conj (Disj (Neg (Atom "landed"),
>                        Atom "rich"),
>                  Neg (Conj (Atom "saintly",
>                             Atom "rich")))),
>       Disj (Neg (Atom "landed"), Neg (Atom "saintly")))
> : prop
```

In mathematical notation this is

$$((landed \to rich) \wedge \neg(saintly \wedge rich)) \to (landed \to \neg saintly)$$

For a more readable display, let us declare a function for converting a proposition to a string.

```
fun show (Atom a)     = a
  | show (Neg p)      = "(~" ^ show p ^ ")"
  | show (Conj(p,q)) = "(" ^ show p ^ " & " ^ show q ^ ")"
  | show (Disj(p,q))  = "(" ^ show p ^ " | " ^ show q ^ ")";
> val show = fn : prop -> string
```

Here is our goal:

```
show goal;
> "((~(((~landed) | rich) & (~(saintly & rich))))
>       | ((~landed) | (~saintly)))"   : string
```

Spaces and line breaks have been inserted in the output above to make it more legible, as elsewhere in this book.

Exercise 4.32 Write a version of *show* that suppresses needless parentheses. If $\neg$ has highest precedence and $\vee$ the lowest then all the parentheses in $((\neg a) \wedge b) \vee c$ are redundant. Since $\wedge$ and $\vee$ are associative, suppress parentheses in $(a \wedge b) \wedge (c \wedge d)$.

Exercise 4.33 Write a function to evaluate a proposition using the standard truth tables. One argument should be a list of the true atoms, all others to be assumed false.

4.18 *Negation normal form*

Any proposition can be converted into ***negation normal form*** (NNF), where $\neg$ is only applied to atoms, by pushing negations into conjunctions and disjunctions. Repeatedly replace

$$\neg\neg p \text{ by } p$$
$$\neg(p \wedge q) \text{ by } (\neg p) \vee (\neg q)$$
$$\neg(p \vee q) \text{ by } (\neg p) \wedge (\neg q)$$

Such replacements are sometimes called ***rewrite rules***. First, consider whether they make sense. Are they unambiguous? Yes, because the left sides of the rules cover distinct cases. Will the replacements eventually stop? Yes; although they can create additional negations, the negated parts shrink. How do we know when to stop? Here a single sweep through the proposition suffices.

Function *nnf* applies these rules literally. Where no rule applies, it simply makes recursive calls.

```
fun nnf (Atom a)            = Atom a
  | nnf (Neg (Atom a))      = Neg (Atom a)
  | nnf (Neg (Neg p))       = nnf p
  | nnf (Neg (Conj(p,q)))   = nnf(Disj(Neg p, Neg q))
  | nnf (Neg (Disj(p,q)))   = nnf(Conj(Neg p, Neg q))
  | nnf (Conj(p,q))         = Conj(nnf p, nnf q)
  | nnf (Disj(p,q))         = Disj(nnf p, nnf q);
> val nnf = fn : prop -> prop
```

Assumption 2, $\neg(saintly \wedge rich)$, is converted to $\neg saintly \vee \neg rich$. Function *show* displays the result.

```
nnf assumption2;
> Disj (Neg (Atom "saintly"), Neg (Atom "rich")) : prop
show it;
> "((~saintly) | (~rich))" : string
```

The function *nnf* can be improved. Given $\neg(p \wedge q)$ it evaluates

$$nnf\,(Disj\,(Neg\,p,\, Neg\,q))$$

The recursive call then computes

$$Disj\,(nnf\,(Neg\,p),\, nnf\,(Neg\,q))$$

Making the function evaluate this expression directly saves a recursive call — similarly for $\neg(p \vee q)$.

It can be faster still. A separate function to compute $nnf\,(Neg\ p)$ avoids the needless construction of negations. In mutual recursion, function $nnfpos\ p$ computes the normal form of p while $nnfneg\ p$ computes the normal form of $Neg\ p$.

```
fun nnfpos (Atom a)     = Atom a
  | nnfpos (Neg p)      = nnfneg p
  | nnfpos (Conj(p,q)) = Conj(nnfpos p, nnfpos q)
  | nnfpos (Disj(p,q)) = Disj(nnfpos p, nnfpos q)
and nnfneg (Atom a)     = Neg (Atom a)
  | nnfneg (Neg p)      = nnfpos p
  | nnfneg (Conj(p,q)) = Disj(nnfneg p, nnfneg q)
  | nnfneg (Disj(p,q)) = Conj(nnfneg p, nnfneg q);
```

4.19 *Conjunctive normal form*

Conjunctive normal form is the basis of our tautology checker, and also of the resolution method of theorem proving. Hardware designers know it as the maxterm representation of a boolean expression.

A ***literal*** is an atom or its negation. A proposition is in ***conjunctive normal form*** (CNF) if it has the form $p_1 \wedge \cdots \wedge p_m$, where each p_i is a disjunction of literals.

To check whether p is a tautology, reduce it to an equivalent proposition in CNF. Now if $p_1 \wedge \cdots \wedge p_m$ is a tautology then so is p_i for $i = 1, \ldots, m$. Suppose p_i is $q_1 \vee \cdots \vee q_n$, where $q_1, \ldots, q_n$ are literals. If the literals include an atom and its negation then p_i is a tautology. Otherwise the atoms can be given truth values to falsify each literal in p_i, and therefore p is not a tautology.

To obtain CNF, start with a proposition in negation normal form. Using the distributive law, push in disjunctions until they apply only to literals. Replace

$$p \vee (q \wedge r) \text{ by } (p \vee q) \wedge (p \vee r)$$
$$(q \wedge r) \vee p \text{ by } (q \vee p) \wedge (r \vee p)$$

These replacements are less straightforward than those that yield negation normal form. They are ambiguous — both of them apply to $(a \wedge b) \vee (c \wedge d)$ — but the resulting normal forms are logically equivalent. Termination is assured; although each replacement makes the proposition bigger, it replaces a disjunction by smaller disjunctions, and this cannot go on forever.

A disjunction may contain buried conjunctions; take for instance $a \vee (b \vee (c \wedge d))$. Our replacement strategy, given $p \vee q$, first puts p and q into CNF. This

brings any conjunctions to the top. Then applying the replacements distributes the disjunctions into the conjunctions.

Calling $distrib(p, q)$ computes the disjunction $p \vee q$ in CNF, given p and q in CNF. If neither is a conjunction then the result is $p \vee q$, the only case where $distrib$ makes a disjunction. Otherwise it distributes into a conjunction.

```
fun distrib (p, Conj(q,r)) = Conj(distrib(p,q), distrib(p,r))
  | distrib (Conj(q,r), p) = Conj(distrib(q,p), distrib(r,p))
  | distrib (p, q)         = Disj(p,q)   (*no conjunctions*);
> val distrib = fn : prop * prop -> prop
```

The first two cases overlap: if both p and q are conjunctions then $distrib(p, q)$ takes the first case, because ML matches patterns in order. This is a natural way to express the function. As we can see, $distrib$ makes every possible disjunction from the parts available:

```
distrib (Conj(rich,saintly), Conj(landed, Neg rich));
> Conj (Conj (Disj (Atom "rich", Atom "landed"),
>             Disj (Atom "saintly", Atom "landed")),
>       Conj (Disj (Atom "rich", Neg (Atom "rich")),
>             Disj (Atom "saintly", Neg (Atom "rich"))))
> : prop
show it;
> "(((rich | landed) & (saintly | landed)) &
>   ((rich | (~rich)) & (saintly | (~rich))))" : string
```

The conjunctive normal form of $p \wedge q$ is simply the conjunction of those of p and q. Function cnf is simple because $distrib$ does most of the work. The third case catches both $Atom$ and Neg.

```
fun cnf (Conj(p,q)) = Conj (cnf p, cnf q)
  | cnf (Disj(p,q)) = distrib (cnf p, cnf q)
  | cnf p           = p     (*a literal*) ;
> val cnf = fn : prop -> prop
```

Finally, we convert the desired goal into CNF using cnf and nnf:

```
val cgoal = cnf (nnf goal);
> val cgoal = Conj (..., ...) : prop
show cgoal;
> "((((landed | saintly) | ((~landed) | (~saintly))) &
>    (((~rich) | saintly) | ((~landed) | (~saintly)))) &
>   (((landed | rich) | ((~landed) | (~saintly))) &
>    (((~rich) | rich) | ((~landed) | (~saintly)))))"
> : string
```

This is indeed a tautology. Each of the four disjunctions contains some $Atom$ and its negation: $landed$, $saintly$, $landed$ and $rich$, respectively. To detect this,

function *positives* returns a list of the positive atoms in a disjunction, while *negatives* returns a list of the negative atoms. Unanticipated cases indicate that the proposition is not in CNF; an exception results.

```
exception NonCNF;
fun positives (Atom a)       = [a]
  | positives (Neg(Atom _))  = []
  | positives (Disj(p,q))    = positives p @ positives q
  | positives  _             = raise NonCNF;
> val positives = fn : prop -> string list
fun negatives (Atom _)       = []
  | negatives (Neg(Atom a))  = [a]
  | negatives (Disj(p,q))    = negatives p @ negatives q
  | negatives  _             = raise NonCNF;
> val negatives = fn : prop -> string list
```

Function *taut* performs the tautology check on any CNF proposition, using *inter* (see Section 3.15) to form the intersection of the positive and negative atoms. The final outcome is perhaps an anticlimax.

```
fun taut (Conj(p,q)) = taut p andalso taut q
  | taut p           = not (null (inter (positives p, negatives p)));
> val taut = fn : prop -> bool
taut cgoal;
> true : bool
```

Advanced tautology checkers. The tautology checker described above is not practical. Ordered binary decision diagrams (OBDDs) can solve serious problems in hardware design. They employ directed graphs, where each node represents an 'if-then-else' decision. Moore (1994) explains the ideas and the key optimizations, involving hashing and caching.

The Davis-Putnam procedure makes use of CNF. It can solve hard constraint satisfaction problems and has settled open questions in combinatorial mathematics. Zhang and Stickel (1994) describe an algorithm that could be coded in ML. Uribe and Stickel (1994) describe experimental comparisons between the procedure and OBDDs.

Exercise 4.34 A proposition in conjunctive normal form can be represented as a list of lists of literals. The outer list is a conjunction; each inner list is a disjunction. Write functions to convert a proposition into CNF using this representation.

Exercise 4.35 Modify the definition of *distrib* so that no two cases overlap.

Exercise 4.36 A proposition is in ***disjunctive normal form*** (DNF) if it has the form $p_1 \vee \cdots \vee p_m$, where each p_i is a conjunction of literals. A proposition is

inconsistent if its negation is a tautology. Describe a method of testing whether a proposition is inconsistent that involves DNF. Code this method in ML.

Summary of main points

- A `datatype` declaration creates a new type by combining several existing types.
- A pattern consists of constructors and variables.
- Exceptions are a general mechanism for responding to run-time errors.
- A recursive `datatype` declaration can define trees.
- Binary trees may represent many data structures, including dictionaries, functional arrays and priority queues.
- Pattern-matching can express transformations on logical formulæ.