Solutions to Exercises

Solutions for Chapter 2

Exercise 2.1

Provide a function to check if a character is alphanumeric, that is lower case, upper case or numeric.

One solution is to follow the same approach as in the function `isupper` for each of the three possibilities and link them with the special operator `\/` :

```
isalpha c = (c >= 'A' & c <= 'Z')
            \/
            (c >= 'a' & c <= 'z')
            \/
            (c >= '0' & c <= '9')
```

An second approach is to use continued relations:

```
isalpha c = ('A' <= c <= 'Z')
            \/
            ('a' <= c <= 'z')
            \/
            ('0' <= c <= '9')
```

A final approach is to define the functions `isupper`, `islower` and `isdigit` and combine them:

```
isalpha c = (isupper c) \/ (islower c) \/ (isdigit c)
```

This approach shows the advantage of reusing existing simple functions to build more complex functions.

Exercise 2.2

What happens in the following application and why?

```
myfst (3, (4 div 0))
```

The function evaluates to 3, the potential divide by zero error is ignored because Miranda only evaluates as much of its parameter as it needs.

Exercise 2.3

Define a function dup which takes a single element of any type and returns a tuple with the element duplicated.

The answer is just a direct translation of the specification into Miranda:

```
dup :: * -> (*,*)
dup x = (x, x)
```

Exercise 2.4

Modify the function `solomonGrundy` so that `Thursday` and `Friday` may be treated with special significance.

The pattern matching version is easily modified; all that is needed is to insert the extra cases somewhere before the default pattern:

```
solomonGrundy "Monday"   = "Born"
solomonGrundy "Thursday" = "Ill"
solomonGrundy "Friday"   = "Worse"
solomonGrundy "Sunday"   = "Buried"
solomonGrundy anyday     = "Did something else"
```

By contrast, a guarded conditional version is rather messy:

```
solomonGrundy day = "Born", if day = "Monday"
                  = "Ill", if day = "Thursday"
                  = "Worse", if day = "Friday"
                  = "Buried", if day = "Sunday"
                  = "Did something else", otherwise
```

Exercise 2.5

Define a function `intmax` which takes a number pair and returns the greater of its two components.

```
intmax :: (num,num) -> num
intmax (x, y) = x, if x > y
              = y, otherwise
```

Exercise 2.6

Define a recursive function to add up all the integers from 1 to a given upper limit.

```
addints :: num -> num
addints 1  = 1
addints n  = n + addints (n - 1)
```

The terminating condition is the first pattern (the integer `1`) and the parameter of recursion is `n`, which converges towards `1` by repeated subtraction. Note that `addints` fails if it is applied to a number less than 1. See also Chapter 2.9.

Exercise 2.7

Write `printdots` in an accumulative recursive style. This will require more than one function.

The accumulator will hold the growing sequence of dots; since the number `n` cannot be used for this purpose, another parameter is needed. This involves the definition of an auxiliary function to incorporate the accumulator:

```
printdots :: num -> [char]
printdots n = xprintdots (n, "")

xprintdots :: (num, [char]) -> [char]
xprintdots (0, dotstring) = dotstring
xprintdots (n, dotstring)
    = xprintdots (n - 1,  dotstring ++ ".")
```

Notice that the function `printdots` initializes the accumulator `dotstring` with an empty string.

Exercise 2.8

Write the function `plus` in a stack recursive style.

The parameter of recursion is `y` and the terminating condition is when `y` is `0`. In this version, `x` no longer serves as an accumulator, but as the second operand to the final addition:

```
plus :: (num,num) -> num
plus (x, 0) = x
plus (x, y) = 1 + plus (x, y - 1)
```

Exercise 2.9

Write a function called `int_divide` which divides one whole number by another; the function should not use any arithmetic operators except for subtraction, addition and unary minus.

The division is straightforward: what requires some thought is the handling of positive and negative values of the operands. Not every problem has an elegant pattern matching solution!

```
int_divide :: (num,num) -> num
int_divide (n, 0) = error "Division by zero"
int_divide (n, m) = error "Division: operands must be integers",
                    if ~ ((integer n) & (integer m))
                  = posdiv (-n, -m),    if (n < 0) & (m < 0)
                  = - (posdiv (n, -m)), if m < 0
                  = - (posdiv (-n, m)),if n < 0
                  = posdiv (n,m), otherwise

posdiv :: (num,num) -> num
posdiv (n, m) = 0, if n < m
              = 1 + posdiv (n - m, m), otherwise
```

Note that the applications `-(posdiv (n, -m))` and `-(posdiv (-n, m))` must be bracketed to evaluate to a numeric result for the unary negation operator `-`. If the brackets were omitted then Miranda would attempt to apply `-` to the function `posdiv` rather than to its result.

Solutions for Chapter 3

Exercise 3.1

Give the two possible correct versions of `wrong_y`.

Either the first operand should be an integer or the second operand should be a list of integer lists:

```
correct_y1 = 1 : [2,3]
correct_y2 = [1] : [[2,3]]
```

Exercise 3.2

Which of the following are legal list constructions?

```
list1 = 1 : []
list2 = 1 : [] : []
list3 = 1 : [1]
```

```
list4 = [] : [1]
list5 = [1] : [1] : []
```

The correct constructions are `list1`, `list3` and `list5`.

The construction `list2` fails because `:` is right-associative. Thus, `list2` is defined to be `(1 : ([] : []))`, which is the same as `(1 : [ [] ])`, which in turn is a type error because it attempts to join an integer to a list of lists.

The construction `list4` fails because it attempts to add a list (in this case the empty list) to a list of integers, which causes a type error.

Exercise 3.3

Miranda adopts the view that it is meaningless to attempt to extract something from nothing; generating an error seems a reasonable treatment for such an attempt. What would be the consequences if `hd` and `tl` were to evaluate to `[]` when applied to an empty list?

The following equality would no longer hold for all values of `alist`:

```
alist = hd alist : tl alist
```

The equality would not hold when `alist` was `[]`, since the right-hand side would evaluate to `[[]]`.

Furthermore, such definitions for `hd` and `tl` would be totally incompatible with the Miranda type system; for example, any function which applied `hd` to a list of integers could not be sure whether the value returned was going to be an integer or a list!

Exercise 3.4

At first sight it would appear that `show` can be bypassed by defining a function that quotes its numeric parameter:

```
numbertostring :: num -> [char]
numbertostring n = "n"
```

Explain what the above function *actually* does.

All it does is produce the string `"n"`. The quotation marks are *not* constructors, unlike the square brackets which denote the list aggregate format.

Exercise 3.5

Write a stack recursive function to add all numbers less than 3 which appear in a list of numbers.

```
addlessthanthree :: [num] -> num
addlessthanthree [] = 0
addlessthanthree (front : rest)
    = front + addlessthanthree rest, if (front < 3)
    = addlessthanthree rest, otherwise
```

Exercise 3.6

The following function `listmax` is accumulative recursive. Rather than using an explicit accumulator, it uses the front of the list to hold the current maximum value.

```
numlist == [num]

listmax :: numlist -> num
listmax [] = error "listmax: empty list"
listmax (front : []) = front
listmax (front : next : rest)
    = listmax (front : rest), if front > rest
    = listmax (next : rest), otherwise
```

Rewrite `listmax` so that it uses an auxiliary function and an explicit accumulator to store the current largest item in the list.

The explicit accumulator is initialized with the front of a non-empty list; the rest of the code is remarkably similar:

```
numlist == [num]

listmax :: numlist -> num
listmax [] = error "listmax: empty list"
listmax (front : rest) = xlistmax (rest, front)

xlistmax :: (numlist, num) -> num
xlistmax ([], maxvalue) = maxvalue
xlistmax (front : rest, maxvalue)
      = xlistmax (rest, front), if  front > maxvalue
      = xlistmax (rest, maxvalue), otherwise
```

Exercise 3.7

What happens if a negative value of `n` is supplied to the first version of `mydrop` ?

Eventually `(front :  rest)` will converge to `[]` and an error will be reported.

Exercise 3.8

Write the function `shorterthan` used by the final version of `mydrop`.

The approach taken is similar to that in defining the function `startswith` in Chapter 3.7.2. Both the number and the list converge towards terminating conditions, respectively, by the integer one and by an element at a time. Hence, zero indicates that there may still be items in the list, in which case the list cannot be shorter than the specified number. Conversely, `[]` indicates that the list is shorter than the number of items to be discarded.

```
shorterthan :: (num, [*]) -> bool

shorterthan (0, alist) = False
shorterthan (n, []) = True
shorterthan (n, front : rest) = shorterthan (n - 1, rest)
```

Exercise 3.9

Use structural induction to design the function `mytake`, which works similarly to `mydrop` but takes the first `n` items in a list and discards the rest.

The type of the function is:

```
(num, [*]) -> [*]
```

The general case is:

```
mytake (n, front : rest) = ??
```

There are two parameters of recursion; the inductive hypothesis must therefore assume that `take (n - 1, rest)` evaluates to an appropriate list. The inductive step is to construct a list of the `front` value (which must be retained) with that list:

```
mytake (n, front : rest)
    = front : mytake (n - 1, rest)
```

The terminating cases are:

1. Taking no elements; this must just give an empty list:

   ```
   mytake (0, alist) = []
   ```

2. Attempting to take some items from an empty list; this is an error:

   ```
   mytake (n, []) = error "take: list too small"
   ```

Notice that asking for zero items from an empty list is covered by `mytake (0, alist)` and therefore this pattern must appear first.

The final code is:

```
mytake :: (num, [*]) -> [*]

mytake (0, alist) = []
mytake (n, []) = error "mytake: list too small"
mytake (n, front : rest)
   = front : mytake (n - 1, rest)
```

This approach deals with negative numbers in the same manner as the first definition of `mydrop`.

Exercise 3.10

Write a function `fromto` which takes two numbers and a list and outputs all the elements in the list starting from the position indicated by the first integer up to the position indicated by the second integer. For example:

```
Miranda fromto (3, 5, ['a','b','c','d','e','f'])
['d','e','f']
```

To meet this specification, it is necessary to assume that it is possible to extract the first `n` elements from a list and that it is also possible to drop the first `m` elements from a list. Of course, it is quite feasible to write this function from first principles but a lot easier to reuse existing code:

```
fromto :: (num,num,[*]) -> [*]
fromto (m, n, alist) = mydrop (m, mytake (n,alist))
```

Exercise 3.11

Modify the `skipbrackets` program to cater for nested brackets.

```
stringtostring == [char] -> [char]

skipbrackets :: stringtostring
skipbrackets [] = []
skipbrackets ('(' : rest) = skipbrackets (inbrackets rest)
skipbrackets (front : rest) = front : skipbrackets rest

inbrackets :: stringtostring
inbrackets (')' : rest) = rest
inbrackets ('(' : rest) = inbrackets (inbrackets rest)
inbrackets (front : rest) = inbrackets rest
```

Notice the adjustment is minor; the nesting of brackets is a recursive requirement and its treatment is recursively achieved by matching the start of a nested bracket within `inbrackets`, which itself ignores brackets.

An alternative solution, though not recommended, would have been to use mutual recursion within `skipbrackets`, without changing the original definition of `inbrackets`:

```
notrecommended_skipbrackets [] = []
notrecommended_skipbrackets ('(' : rest)
    = inbrackets (notrecommended_skipbrackets rest)
notrecommended_skipbrackets (front : rest)
    = front : notrecommended_skipbrackets rest
```

As with mutually defined functions, it is quite difficult to reason how and why this function succeeds.

Exercise 3.12

It would appear that `sublist` no longer needs its first function pattern because this is checked as the first pattern in `startswith`. Explain why this is incorrect, and also whether the second pattern of `sublist` can safely be removed.

It is not safe to remove the first pattern because matching the empty regular expression with an empty line to be searched would now be met by the second pattern and incorrectly evaluate to `False`. The second pattern cannot be removed because it serves as the terminating condition for recursion along the line to be searched.

Exercise 3.13

An incorrect attempt to optimize the `startswith` program would combine `startswith` and `sublist` in one function:

```
stringpair = ([char], [char])

sublist :: stringpair -> bool
sublist ([], alist) = True
sublist (alist, []) = False
sublist ((regfront : regrest), (lfront : lrest))
    = ((regfront = lfront) & sublist (regrest, lrest))
      \/ sublist ((regfront : regrest), lrest)
```

This follows the general inductive case that the result is `True` if the front two items of the lists are equal and the result of a sublist search of the rest of the two lists is also `True`. Alternatively the entire regular expression matches the rest of the search line. Show why this approach is wrong.

The following application would erroneously evaluate to `True`:

```
sublist ("abc", "ab_this_will_be_ignored_c")
```

Exercise 3.14

Explain the presence of the final pattern in the function `startswith`, even though it should never be encountered.

The type `regtype` could have any number of possible strings, rather than just the strings `"ONCE"` and `"ZERO_MORE"`; the final pattern is intended to suppress the system warning message. A safer solution is presented in Chapter 6.2.4.

Exercise 3.15

What would happen if the second and third pattern in `startswith` were swapped?

The function would still produce the same results; the two patterns are mutually exclusive and it does not matter which appears first.

Exercise 3.16

Alter the `sublist` function so that `"A*"` matches the empty string.

The naive solution is to introduce an extra pattern as the new first pattern:

```
sublist ([("ZERO_MORE", alist)], []) = True
...
```

However, this does not cater for regular expressions of the form `"A*B*"`. An easy solution is to ensure that `startswith` is always applied at least once; this solution is presented in Chapter 9.

Solutions for Chapter 4

Exercise 4.1

Give the types of the following compositions:

```
tl . (++ [])
abs . fst
code . code
show . tl
```

The first composition has the type: `[*] -> [*]` which shows that it is legitimate to compose partially applied functions. The second expression has the type `(num,*) -> num`, which shows that it is valid to compose functions which have tupled parameters. Notice also that `fst` is now only polymorphic in the second component of its parameter because `abs` expects a `num` parameter. The third composition is invalid because the `code` that will be applied second expects its input parameter to be a `char` but the `code` that is applied first produces a value of type `num`.

The final composition is interesting in that the expression could be used at the Miranda prompt:

```
Miranda  (show . tl) "abc"
bc
```

However, it would be illegal to attempt to use the expression on its own, as the right-hand side of a script file definition:

```
wrongshowtail = show . tl
```

This will result in an error because Miranda could not resolve the type of the overloaded `show` function.

Exercise 4.2

Theoreticians claim that all of the combinators (and consequently all functions) can be written in terms of the combinator K (`cancel`) and the following combinator S (`distribute`):

```
distribute f g x = f x (g x)
```

Define the combinator `identity` (which returns its argument unaltered) using only the functions `distribute` and `cancel` in the function body. Provide a similar definition for a curried version of `snd`.

```
identity = distribute cancel cancel
|| I = SKK

curried_snd = distribute cancel
|| curried_snd = SK
```

Whilst the above are relatively straightforward, it is not always so easy to provide combinator expressions; for example `compose, (B)` is defined as:

```
compose = distribute (cancel distribute) cancel
|| B = S (KS) K
```

and the simplest combinator expression for `swap, (C)` is:

```
swap = distribute (compose compose distribute) (cancel cancel)
|| C = S (BBS) (KK)
```

Exercise 4.3

Explain why `make_curried` cannot be generalized to work for functions with an arbitrary number of tuple components.

This is because all functions must have a well-defined source type and therefore must have either a fixed number of curried arguments or a tuple of fixed size. A separate conversion function is therefore necessary for all the uncurried functions with two arguments, another for all the uncurried functions of three arguments, and so on.

Exercise 4.4

Write the function `make_uncurried` which will allow a curried, dyadic function to accept a tuple as its argument.

This is the mirror of `make_curried`:

```
make_uncurried :: (* -> ** -> ***) -> (*,**) -> ***

make_uncurried ff (x,y) = ff x y
```

Exercise 4.5

The built-in function `fst` can be written using `make_uncurried` and the `cancel` combinator:

```
myfst = make_uncurried cancel
```

Provide a similar definition for the built-in function `snd`.

The function `snd` can be thought of as `fst` with its parameters swapped, that is:

```
mysnd = make_uncurried (swap cancel)
```

A hand evaluation reveals:

```
mysnd (1,2)
==> make_uncurried (swap cancel) (1,2)
==> (swap cancel) 1 2
==> swap cancel 1 2
==> cancel 2 1
==> 2
```

Exercise 4.6

Explain why `myiterate` is non-robust and provide a robust version.

The function should check that `n` is a *positive integer*; the best way to achieve this is to separate the validation from the processing:

```
myiterate :: num -> (* -> *) -> * -> *
myiterate n ff result
    = error "myiterate", if  n < 0  \/ not (integer n)
    = xmyiterate n ff result, otherwise
```

where the auxiliary function `xmyiterate` is the same as the original version of `myiterate`. Chapter 5 shows how the auxiliary function can be tightly coupled to the new version of `myiterate`.

Exercise 4.7

Imperative programming languages generally have a general-purpose iterative control structure known as a "while" loop. This construct will repeatedly apply a function to a variable whilst the variable satisfies some predefined condition. Define an equivalent function in Miranda.

This function can be written directly from its informal specification:

```
whiletrue :: (* -> bool) -> (* -> *) -> * -> *
whiletrue pred ff state
   = whiletrue pred ff (ff state), if pred state
   = state, otherwise
```

Here the predefined condition is the guard `pred state`, the repeated application is the recursive application of `while`, with the application `ff state` representing the change in the variable.

Note that there is no guarantee that the condition `(pred state)` will ever be satisfied. This is a general problem of computing, known as the *halting problem*. Stated briefly, it is impossible to write a program that will infallibly determine whether an arbitrary function (given some arbitrary input) will terminate or loop forever. One consequence is that there is no point in attempting excessive and unnecessary validation of such general-purpose iterative constructs as `whiletrue`.

Exercise 4.8

In the definition of `map_two`, source lists of unequal length have been treated as an error. It is an equally valid design decision to truncate the longer list; amend the definition to meet this revised specification.

This is a trivial task; the last error-handling pattern can be converted to have an action that returns the empty list. This pattern also caters for the case of both lists being empty.

```
map_two (* -> ** -> ***) -> [*] -> [**] -> [***]
map_two ff (front1 : rest1) (front2 : rest2)
     = (ff front1 front2) : map_two ff rest1 rest2
map_two ff alist blist = []
```

Note that this function has the same behaviour as the Standard Environment function `zip2`.

Exercise 4.9

Write a function `applylist` which takes a list of functions (each of type `*->**`) and an item (of type `*`) and returns a list of the results of applying each function to the item. For example: `applylist [(+ 10),(* 3)] 2` will evaluate to `[12,6]`.

This function has a similar shape to `map`, only here it is the head of the list that is applied to the first parameter rather than vice versa:

```
applylist :: [* -> **] -> * -> [**]
applylist [] item = []
applylist (ffront : frest) item
        = (ffront item) : applylist frest item
```

Exercise 4.10

Explain why the following definitions are equivalent:

```
f1 x alist = map (plus x) alist
f2 x = map (plus x)
f3 = (map . plus)
```

The definition of `f2` is that of a partially applied function; Miranda can infer that it requires an extra list parameter from the type of `map`.

The definition of `f3` makes use of the following equivalence:

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

In this case `f` is `map` and `g` is `plus` and the `x` can be discarded for the same reason as in the definition of `f2`.

To a certain extent, it is a matter of taste which function should be used; `f1` has the advantage that *all* of its arguments are visible at the function definition, whilst `f3` has the advantage of brevity and perhaps the fact that the programmer can concentrate on what the function does rather than what it does it to. There are no absolute guidelines, but it is important to be able to read all three kinds of definitions. Finally, notice that the type of each function is the same, that is:

```
num -> [num] -> [num]
```

Exercise 4.11

Rewrite the function `map` in terms of `reduce`.

This is simply done by generalizing the code in the text:

```
map_inc = reduce ((:) . inc) [])
```

That is, by replacing the specific function `inc` with a polymorphic parameter:

```
reduce_map :: (*->**) -> [*] -> [**]
reduce_map ff = reduce ((:) . ff) []
```

Now `reduce_map` can be used in exactly the same way as `map`:

```
map_inc :: (num->num)->[num]->[num]
map_inc = reduce_map inc
```

Exercise 4.12

Some functions cannot be generalized over lists, as they have no obvious default value for the empty list; for example, it does not make sense to take the maximum value of an empty list. Write the function `reduce1` to cater for functions that require at least one list item.

It is clear from the specification that an empty list must be considered an error, otherwise the first item in the list can be used as the default value to `reduce`. This is a good example of reusing code.

```
reduce1 :: (* -> * -> *) -> [*] -> [*]
reduce1 ff [] = error "reduce1"
reduce1 ff (front : rest) = reduce ff front rest
```

Exercise 4.13

Write two curried versions of `mymember` (as specified in Chapter 3.6), using `reduce` and `accumulate`, respectively, and discuss their types and differences.

The `reduce` version is straightforward and uses the prefix, curried functions defined in Chapter 4.1.2:

```
reduce_member alist item
     = reduce (either . (equal item)) False alist
```

A hand evaluation of `(reduce_member [1,2,3] 1)` shows:

```
reduce_member [1,2,3] 1
==> reduce (either . (equal 1)) False [1,2,3]
==> ((either . (equal 1)) 1) (reduce (either . (equal 1)) False [2,3])
==> (either (equal 1 1)) (reduce (either . (equal 1)) False [2,3])
==> either True (reduce (either . (equal 1)) False [2,3])
==> True \/ (reduce (either . (equal 1)) False [2,3])
==> True
```

The `accumulate` version is less straightforward; this is not an example where `accumulate` can be safely substituted for `reduce`. An attempt to define `member` as:

```
accumulate_member alist item
    = accumulate (either . equal item) False alist
```

will only work if `item` and the elements of `alist` are of type `bool`. This is easily verified by checking the type of the above function:

```
Miranda accumulate_member ::
[bool] -> bool -> bool
```

For example, a hand evaluation of `accumulate_member` to a list `[a,b,c]` (where the list elements are of arbitrary type) shows:

```
accumulate_member [a,b] item
==> accumulate (either . equal item) False [a,b]
==> accumulate (either . equal item) ((either . equal item) False a) [b]
==> accumulate (either . equal item) (either (equal item False) a) [b]
==> error or item must be of type bool, since equal
    expects both its arguments to be of the same type
```

In order to make the function more general, the default value `False` must become the second of the parameters to the functional argument. This is easily achieved using the `swap` combinator:

```
accumulate_member :: [*] -> * -> bool
accumulate_member alist item
      = accumulate (swap (either . (equal item))) False alist
```

A hand evaluation of `(accumulate_member [1,2,3] 1)` now shows:

```
accumulate_member [1,2,3] item
==> accumulate (swap (either . (equal 1))) False [1,2,3]
==> accumulate (swap (either . (equal 1)))
               (swap (either . (equal 1)) False 1) [2,3]
==> accumulate (swap (either . (equal 1)))
               ((either . (equal 1)) 1 False) [2,3]
==> accumulate (swap (either . (equal 1)))
               (either (equal 1 1) False) [2,3]
==> accumulate (swap (either . (equal 1))) (True \/ False) [2,3]
==> accumulate (swap (either . (equal 1))) True [2,3]
==> ...
```

Exercise 4.14

Define the function `mydropwhile` which takes a list and a predicate as arguments and returns the list without the initial sublist of members which satisfy the predicate.

This function has same behaviour as the Standard Environment `dropwhile` and can be written using the same approach as the function `takewhile`:

```
mydropwhile :: (* -> bool) -> [*] -> [*]
mydropwhile pred  [] = []
mydropwhile pred (front : rest)
    = mydropwhile pred rest, if pred front
    = (front : rest), otherwise
```

Exercise 4.15

The *set* data structure may be considered as an unordered list of unique items. Using the built-in functions `filter` and `member`, the following function will yield a list of all the items common to two sets:

```
intersection :: [*] -> [*] -> [*]
intersection aset bset = filter (member aset) bset
```

Write a function `union` to create a set of all the items in two sets.

The union of two sets can be considered as all the members of the first set that are *not* in the second set, together with that second set. The answer makes use of the design for `intersection`, but inverts the truth value of the predicate to `filter` in order to exclude common members (by means of the composition of `~`with `member`):

```
union :: [*] -> [*] -> [*]
union aset bset = aset ++ filter ((~) . (member aset)) bset
```

An alternative specification is to remove the duplicates from the result of appending the two sets.

Exercise 4.16

An equivalent version of `stringsort` using `accumulate` would require that the arguments to `(insert lessthan)` be reversed. Why is this the case?

For the same reasons as discussed with the accumulative version of member. The default value `[]` becomes the first argument to the `(insert lessthan)` and an attempt would be made to compare it with an actual value. This will only work if that value is also an empty list.

Exercise 4.17

A function `foldiftrue` which `reduces` only those elements of a list which satisfy a given predicate could be defined as:

```
foldiftrue :: (* -> bool) -> (* -> ** -> **) -> ** -> [*]
foldiftrue pred ff default [] = default
foldiftrue pred ff default (front : rest)
    = (ff front (foldiftrue pred ff default rest)), if pred front
    = foldiftrue pred ff default rest, otherwise
```

Write this function in terms of a composition of `reduce` and `filter`.

```
foldiftrue :: (* -> bool) -> (* -> ** -> **) -> ** -> [*]
foldiftrue pred ff default = (reduce ff default) . (filter pred)
```

The composed style is probably easier to read; the explicit recursion may appear algorithmically more efficient, but this really depends upon the underlying implementation (which might automatically convert function compositions to their equivalent explicit recursive form (Darlington *et al.*, 1982)).

Exercise 4.18

What is the purpose of the function `guesswhat`?

The function is a rather convoluted method of writing the built-in operator # in terms of itself.

Solutions for Chapter 5

Exercise 5.1

Write a function, using **where** definitions, to return the string that appears before a given sublist in a string. For example, `beforestring "and" "Miranda"` will return the string "Mir".

This is yet another example where it makes sense to reuse existing code, in this case the `startswith` function designed in Chapter 3:

```
beforestring :: [char] -> [char] -> [char]

beforestring any []
    = error "Beforestring: empty string to search"
beforestring bstring (front : rest)
    = [], if  startswith (bstring, front : rest)
      where
        startswith ([], any) = True
        startswith (any, []) = False
        startswith (front1 : rest1,  front2 : rest2)
            = (front1 = front2)
              & startswith (rest1, rest2)
    = front : beforestring bstring rest, otherwise
```

Note that it is not permitted to incorporate **type** declarations within a **where** clause.

Exercise 5.2

The Standard Environment function `lines` translates a list of characters containing newlines into a list of character lists, by splitting the original at the newline characters, which are deleted from the result. For example, `lines` applied to:

```
"Programming with Miranda\nby\nClack\nMyers\nand Poon"
```

evaluates to:

```
["Programming with Miranda","by","Clack","Myers","and Poon"]
```

Write this function using a **where** definition.

```
mylines :: [char]->[[char]]
mylines [] = []
mylines ('\n' : rest) = [] : mylines rest
mylines (front : rest)
  = (front : thisline) : otherlines
    where
     (thisline : otherlines)
        = mylines rest, if rest ~= []
        = [[]], otherwise
          || handle missing '\n' on last line
```

The above code uses the induction hypothesis that `mylines rest` will correctly return a list of strings as required. Thus, in order to return the correct result, all that is necessary is to cons the `front` element onto the start of the first string returned by the recursive call. There are two base cases, the first of which deals with an empty list (this is the terminating condition), and the second of which returns an appropriate result if the front character in the input string is a newline.

Exercise 5.3

Define a list comprehension which has the same behaviour as the built-in function `filter`.

```
filter :: (* -> bool) -> [*] -> [*]
filter pred anylist = [x <- anylist | pred x]
```

This can be read as the list of all elements `x`, **where** `x` is sequentially from the list `anylist` **such that** `pred x` evaluates to `True`.

Exercise 5.4

Rewrite *quicksort* using list comprehensions.

```
qsort :: (* -> * -> bool) -> [*] -> [*]
qsort order = xqsort
              where
               xqsort [] = []
               xqsort (front : rest)
                  = xqsort [x | x <- rest; order x front]
                    ++ [front] ++
                    xqsort [x | x <- rest; ~ (order x front)]
```

The algorithm is the same as shown earlier in this chapter, but the `split` function has been replaced by two list comprehensions. The first generates all values from the `rest` of the list such that, for each value `x`, the `order` predicate on `x` and the `front` evaluates to `True`; the second where it does not. Notice also that the auxiliary function does not need to carry around `order`, which is in scope.

Exercise 5.5

Use a list comprehension to write a function that generates a list of the squares of all the even numbers from a given lower limit to upper limit. Change this function to generate an infinite list of even numbers and their squares.

One approach is to generate all the possible integers between `low` and `high` and create a list of the squares of only those integers that satisfy the constraint `mod 2 - 0`:

```
gensquares low high
     = [x * x | x <- [low .. high] ; x mod 2 = 0]
```

This is readily adapted to cater for infinite lists:

```
infinite_squares
     = [(x, x * x) | x <- [2..]; x mod 2 = 0]
```

Alternatively, it is possible to utilize the list comprehension's ability to recognize integer intervals:

```
infinite_squares = [(x, x * x) | x <- [2,4..]]
```

Solutions for Chapter 6

Exercise 6.1

Write a function to calculate the distance between a pair of `coords`.

The hardest part of this solution is to remember the geometry!

```
coords ::= Coords (num,num,num)

distance :: coords -> coords -> num
distance (Coords (x1,y1,z1)) (Coords (x2,y2,z2))
   = sqrt (square (x2 - x1) + square (y2 - y1) + square (z2 - z1))
     where
       square n = n * n
```

Exercise 6.2

Given the algebraic type:

```
action ::= Stop | No_change | Start
           | Slow_down | Prepare_to_start
```

write a function to take the appropriate action at each possible change in state for `traffic_light`.

```
drive :: traffic_light -> traffic_light -> action
drive Green Amber      = Slow_down
drive Amber Red        = Stop
drive Red Red_amber    = Prepare_to_start
drive Red_amber Green  = Start
drive x x              = No_change
drive x y              = error "broken lights"
```

Exercise 6.3

A Bochvar three-state logic has constants to indicate whether an expression is true, false or meaningless. Provide an algebraic type definition for this logic together with functions to perform the equivalent three-state versions of `&`, `\/` and logical *implication*. Note that, if any part of an expression is meaningless then the entire expression should be considered meaningless.

```
bochvar ::= TRUE | FALSE | MEANINGLESS

andB :: bochvar -> bochvar -> bochvar
andB TRUE TRUE = TRUE
andB MEANINGLESS any = MEANINGLESS
andB any MEANINGLESS = MEANINGLESS
andB avalue bvalue = FALSE
```

```
orB :: bochvar -> bochvar -> bochvar
orB FALSE FALSE = FALSE
orB MEANINGLESS any = MEANINGLESS
orB any MEANINGLESS = MEANINGLESS
orB avalue bvalue = TRUE

impB :: bochvar -> bochvar -> bochvar
impB TRUE FALSE = FALSE
impB MEANINGLESS any = MEANINGLESS
impB any MEANINGLESS = MEANINGLESS
impB avalue bvalue = TRUE
```

Exercise 6.4

Explain why it is not sensible to attempt to mirror the tree data structure using nested lists.

It is necessary to know the depth of the tree before the correct level of list nesting can be determined. This is because Miranda does not allow lists to contain elements of mixed types and, for example, a double-nested list is of a different type than a triple-nested list. If the depth of the tree is known then each nested list can have the same depth; but this defeats the purpose of the tree data structure, which is designed to be of arbitrary depth.

Exercise 6.5

A number of useful tree manipulation functions follow naturally from the specification of a binary tree. Write functions to parallel the list manipulation functions `map` and `#` (in terms of the number of nodes in the tree).

The equivalent of `map` just traverses the tree, applying the parameter function to each non-empty node:

```
maptree :: (* -> **) -> tree * -> tree **

maptree ff Tnil = Tnil
maptree ff (Tree (ltree, node, rtree))
    = Tree (maptree ff ltree, ff node, maptree ff rtree)
```

The equivalent of `#` could be written by traversing the tree and adding `1` for each non-empty node, although an easier method is to reuse some existing code:

```
nodecount = (#) . tree_to_list
```

Exercise 6.6

What would have been the consequence of writing the function `list_to_tree` as:

```
list_to_tree order = reduce (insertleaf order) Tnil
```

This will fail because `reduce` expects its first argument to be a function of the form:

```
* -> ** -> **
```

whereas `(insertleaf order)` has the type:

```
tree * -> * -> tree *
```

which has the general form:

```
* -> ** -> *
```

It is always worth looking at a function's type for program design and debugging purposes.

Exercise 6.7

Write a function to remove an element from a sorted tree and return a tree that is still sorted.

The base cases are: deleting terminal nodes (for example, node 0 and node 7), which leaves the rest of the tree unaltered, and attempting to delete a node from an empty tree, which is an error.

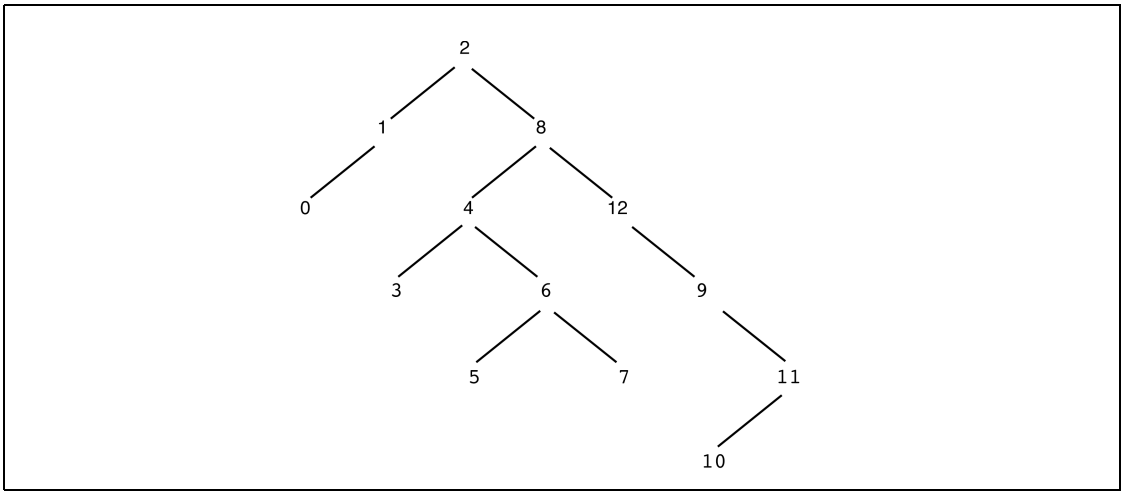

Figure A.1 A sample tree.

The general case is that of deleting non-terminal nodes. Here, only the subtree below the deleted node needs to be re-sorted. One method is to replace the deleted value with

the highest value in the left subtree below the deleted node. Thus, given the sample tree in Figure A.1, deleting node 8 will require the value 8 to be replaced with the value 7. The node which contained the replacement value must now be deleted; this may cause yet another re-sorting of the tree.

For example, to delete node 12 will require the value 11 to take the place of 12; this means that node 11 must next be deleted from its original position, thus causing the value 10 to take its place; this means that node 10 must next be deleted from its original position, which can be achieved with a simple deletion and with no need to consider further subtrees.

Following the above informal specification:

```
delnode :: (* -> * -> bool) -> * -> tree * -> tree *

delnode ff item Tnil
   = error "delnode: item not in tree"
delnode order item (Tree(Tnil, item, rtree))
   = rtree   || 'promote' right subtree
delnode order item (Tree(ltree, item, rtree))
   =  || replace deleted node with new root and
      || delete new root from left subtree
     Tree (delnode order newroot ltree, newroot, rtree)
     where
      newroot
          = findhighest ltree
      findhighest (Tree (ltree, node, Tnil))
          = node
      findhighest (Tree (ltree, node, rtree))
          = findhighest rtree
delnode order item (Tree(ltree, node, rtree))
   = || look for item in rest of tree
     Tree (delnode order item ltree, node, rtree),
        if order item node
   = Tree (ltree, node, delnode order item rtree),
        otherwise
```

Note that when the condition `item = node` is true, the method proceeds by chosing a new root from the left subtree; hence there is no need to check if the right subtree is empty.

There is a simpler approach which merely flattens the tree, then deletes the item from the resultant list, and then turns the list back into a tree using `list_to_tree`. However, this approach immediately leads to a pathologically unbalanced tree, because the list argument to `list_to_tree` is totally sorted and therefore all tree nodes will have only one branch.

Solutions for Chapter 7

Exercise 7.1

Provide function definitions for the `nat` primitives if a recursive underlying data representation is used as follows: `algnat ::= Zero | Succ algnat`.

The solution to this exercise shows that the natural numbers can be modelled using only the constructors `Succ` and `Zero`; there is no need for any underlying built-in type. The abstype body provides a sample of the arithmetic and relational operators, as well as `makeNat` and `show` interfaces.

```
abstype nat
with
   makeNat :: num -> nat
   plusNat :: nat -> nat -> nat
   timesNat :: nat -> nat -> nat
   equalNat :: nat -> nat -> bool
   shownat :: nat -> [char]
   || etc
nat_type ::= Zero | Succ nat_type
nat == nat_type

makeNat x = error "makeNat: non-negative integer expected\n",
            if (x < 0) \/ (~ (integer x))
           = xmakeNat x, otherwise
            where
             xmakeNat 0 = Zero
             xmakeNat x = Succ (xmakeNat (x - 1))

plusNat x Zero = x
plusNat x (Succ y) = plusNat (Succ x) y

timesNat x Zero = Zero
timesNat x (Succ Zero) = x
timesNat x (Succ y) = plusNat x (timesNat x y)

equalNat (Zero) (Zero) = True
equalNat (Succ x) (Succ y) = equalNat x y
equalNat (Succ x) (Zero) = False
equalNat (Zero) (Succ x) = False

shownat Zero = "Zero"
shownat (Succ x) = "(Succ " ++ (shownat x) ++ ")"
```

Note that the definition of `plusNat` is remarkably similar to the accumulative recursive definition of `plus` shown in Chapter 2.

Exercise 7.2

A *date* consists of a day, month and year. Legitimate operations on a date include: creating a date, checking whether a date is earlier or later than another date, adding a day to a date to give a new date, subtracting two dates to give a number of days, and checking if the date is within a leap year. Provide an abstract type declaration for a date.

The fact that the `date` abstract type can be declared on its own, demonstrates that a programmer does not necessarily have to worry about *implementation* at the same time as determining requirements. Coding the interface functions can be deferred or left to another programmer. Notice in this case, the declaration assumes that the date will be converted from a three number tuple, but makes no assumption concerning the type of a `day` or of a `date` itself.

```
abstype date
with
  makeDate :: (num,num,num) -> date
  lessDate :: date -> date -> bool
  greaterDate :: date -> date -> bool
  addday :: day -> date -> date
  diffDate :: date -> date -> day
  isleapyear :: date -> bool
```

Exercise 7.3

Complete the implementation for the `sequence` abstract type.

The completed implementation requires some code reuse for the left-to-right operations, and employing the built-in function `reverse` to cater for right-to-left operations:

```
seqHdL =  hd
seqHdR =  hd . reverse
seqTlL =  tl
seqTlR =  reverse . tl . reverse
seqAppend = (++)
```

Exercise 7.4

Provide a *show* function for the `sequence` abstract type.

Because `sequence` is a polymorphic abstract type, its corresponding `show` function requires a 'dummy' parameter. The `showsequence` implementation takes the first element in the underlying list, applies the dummy function `f` to it, and then concatenates the result to the recursive application on the rest of the list. The identity element is the empty list.

```
abstype sequence *
with
  ...
  seqDisplay :: (sequence *) -> [*]
  showsequence :: (* -> [char]) -> sequence * -> [char]

sequence * == [*]
  ...
seqDisplay s = s
  ...
showsequence f s = foldr ((++) . f) [] s
```

Notice that it is not possible to take the following approach:

```
showsequence f s = s
```

It is necessary to apply the dummy function to *every* component of the data structure. However, the definition for `seqDisplay` is perfectly acceptable because `seqDisplay` is not a *show* function.

Exercise 7.5

Assuming that the underlying type for an abstract `date` type is a three number tuple (day, month, year), provide functions to display the day and month in US format (month, day), UK format (day, month) and to display the month as a string such as "Jan" or "Feb".

This exercise demonstrates that "showing" an abstract type is rather arbitrary. More often than not, what is needed is to show some property of the type:

```
abstype date
with
  makeDate :: (num,num,num) -> date
  ...
  displayUK :: date -> (num,num)
  displayUS :: date -> (num,num)
  displayMonth :: date -> [char]

date = (num,num,num)

displayUK (day, month, year) = (month, day)
displayUS (day, month, year) = (day, month)
displayMonth (day, month, year)
  = ["Jan",Feb","March","April","May","June",
     "July","Aug","Sept","Oct","Nov","Dec"] ! (month - 1)
    || remember list indexing starts at 0
```

Exercise 7.6

An alternative representation of the `Atree` would be:

```
abstype other_tree *
with
   || declarations

ordering * == * -> * -> bool
other_tree * ::= Anil (ordering *)
               | ATree (ordering *) (other_tree *) * (other_tree *)
```

What would be the consequences for the abstract type implementation?

The ordering function would be contained at each node in the `other_tree` rather than just once at the highest root in the tree.

Exercise 7.7

A *queue* aggregate data structure (Standish, 1980) can be defined as either being empty or as consisting of a queue followed by an element; operations include creating a new queue, inserting an element at the end of a queue and removing the first element in a queue. The following declares a set of primitives for a polymorphic abstract type queue:

```
abstype queue *
with
  qisempty = queue * -> bool
  qtop     = queue * -> *
  qinsert  = queue * -> * -> queue *
  qcreate  = queue *
```

Provide an implementation for this abstract type.

Just as with the *array* examples, there are many possible implementations and, as far as the meaning of a program is concerned, it should not matter which is chosen. The implementation rationale is often determined by algorithmic complexity, based on assumptions about the pattern of accesses and updates; however, this subject is beyond the scope of this book. For this **abstype**, it is possible to use a simple list as the underlying type, however the following construction shows an equally valid alternative:

```
queue_type * ::= Qnil | Queue (queue *, *)
queue * == queue_type *

qtop Qnil  = error "Queue: empty queue"
qtop (Queue (Qnil, qfirst)) = qfirst
qtop (Queue (qrest, qfirst))  = qtop qrest

qisempty aq = (aq = Qnil)
qinsert queue item = Queue (queue, item)
qcreate = Qnil
```

Solutions for Chapter 8

Exercise 8.1

Adapt the *wordcount* program so that it will work for more than one input file.

```
manywordcount :: [[char]] -> [char]
manywordcount filenames
   = lay (map f filenames)
     where
      f name = name ++ ":\t" ++ show (wordcount name)
```

This simple solution invokes `wordcount` for each of the given filenames; each result is formatted so that it is transformed from a three-tuple to a string using `show` and then appears after the name of the file, a colon and a tab. The result of `map` is a list of strings, which `lay` (a function from the Standard Environment) concatenates as a single string using newline delimiters.

The above example does not give the total for all the files, nor does it return the result numerically. A different solution might only return a three-tuple giving the totals across all the files:

```
manywordcount2 :: [[char]] -> [char]
manywordcount2 filenames
   = foldr totals (0,0,0) (map f filenames)
     where
        f name = (wordcount name)
        totals (a,b,c) (t1,t2,t3) = (a+t1, b+t2, c+t3)
```

Exercise 8.2

Explain why the following code is incorrect:

```
wrongsplit infile
           = first second
             where first = hd inlist
                   second = tl inlist
                   inlist = readvals infile
```

The code is wrong because Miranda does not know the type of the contents of `infile`, and so `readvals` cannot work. Furthermore, because there is no concrete representation of a function, there is no possible way that `readvals` could create a list of values such that the first item could be treated as a function.

Exercise 8.3

Provide the functions `readbool` and `dropbool` which, respectively, will read and discard a Boolean value from the start of an input file.

These answers use the function `readword` as defined in the text of Chapter 8. They expect the input to be a string representing a Boolean; this is checked and either the appropriate action is taken or an error message is given:

```
readbool :: [char] -> bool
readbool infile
   = check (readword infile)
     where
      check "True" = True
      check "False" = False
      check other = error "readbool"

dropbool :: [char] -> [char]
dropbool infile
   = check (readword infile)
     where
      check "True" = dropword infile
      check "False" = dropword infile
      check other = error "dropbool"
```

Exercise 8.4

Explain why the following attempt at `vend` is incorrect:

```
vend = (System "clear") : wrongdialogue $+
wrongdialogue ip = [Stdout welcome] ++ quit, if sel = 4
                 = [Stdout welcome] ++ (next sel), otherwise
                   where
                   (sel : rest) = ip
                   next 1 = confirm "tea" ++ wrongdialogue rest
                   next 2 = confirm "coffee" ++ wrongdialogue rest
                   next 3 = confirm "soup" ++ wrongdialogue rest
                   next 4 = quit
```

In the above code, the guard `if sel = 4` requires the value of `sel` and therefore causes the input to be scanned **before** the welcome message is printed to the screen.

Exercise 8.5

Why is it necessary for `check` to have a **where** block, and why is `confirm` repeatedly applied within `acknowledge`?

The definition of `dialogue` is given as:

```
dialogue :: [num] -> [sys_message]
dialogue ip
  = [Stdout welcome] ++ next ++ (check rest2)
    where
    (sel : rest) = ip
    (next, rest2)
       = ([tea, coffee, soup] ! (sel - 1)) rest
    check xip = [Stdout menu3] ++ xcheck xip
                where
                 xcheck (1:rest3) = quit rest3
                 xcheck (2:rest3) = dialogue rest3
                 xcheck any       = quit any
```

The function `check` must have a **where** block because `menu3` must be output to the screen before the user's response `xip` is evaluated. In the expression `[Stdout menu3] ++ xcheck xip`, the evaluation of `xcheck` only occurs after the left operand of `++` has been output.

The definition of `acknowledge` is given as:

```
acknowledge :: [char] -> num -> [sys_message]
acknowledge d 1 = confirm (d ++ " with Milk & Sugar")
acknowledge d 2 = confirm (d ++ " with Milk")
acknowledge d 3 = confirm (d ++ " with Sugar")
acknowledge d 4 = confirm d
```

An alternative definition does not use `confirm` repeatedly:

```
wrongacknowledge :: [char] -> num -> [sys_message]
wrongacknowledge d x = confirm (d ++ (mesg x))
                       where
                       mesg 1 = " with Milk & Sugar"
                       mesg 2 = " with Milk"
                       mesg 3 = " with Sugar"
                       mesg 4 = []
```

Unfortunately, this (wrong) version has the undesirable effect that the the phrase "Thank you for choosing" is printed *before* the user has entered a number. This becomes obvious when the definition for `confirm` is given:

```
confirm :: [char] -> [sys_message]
confirm choice = [ Stdout ("\nThank you for choosing "
                 ++ choice ++ ".\n")]
```

Exercise 8.6

Use the above board, as the basis for a simple game of noughts and crosses (tic-tac-toe).

```
>|| Noughts and crosses (tic-tac-toe) game: (Page 1 of 4)

Controlling program - main:
   This makes use of Miranda's keyboard input directive $+
   Start the game by entering 'main' at the Miranda prompt.

> move==(num,num)

> main = [System "stty cbreak"] ++
>        [Stdout greeting] ++
>        [Stdout (game $+)] ++
>        [System "stty sane"]

> greeting
>  = "\nWelcome to the Noughts and Crosses game\n\n" ++
>    "Please enter moves in the form (x,y) followed by a newline\n" ++
>    "Each number should be either 0, 1 or 2; for example: (1,2)\n\n" ++
>    "Enter your first move now, and further moves after the display:\n"

> game:: [move]->[char]
> game ip = xgame ip empty_board

> xgame [] brd = []
> xgame ((x,y) : rest) brd
>    = prompt ++ m1, if ss1
>      ||
>      || don't test s2 here because it will delay printout
>      || of the user's move until the computer move is done
>      ||
>    = prompt ++ m1 ++ xgame rest brd, if (~s1)
>    = prompt ++ "Computer's move:\n" ++ nextmove, otherwise
>      where
>       prompt = clearscreen ++ "Player's move:\n"
>                ++ print_board newboard1
>       clearscreen = "\f"
>       nextmove = print_board newboard2 ++ m2 , if ss2
>                = print_board newboard2 ++ m2
>                  ++ xgame rest newboard2, if (~s2)
>                = print_board newboard2
>                  ++ xgame rest newboard2, otherwise
>       (newboard1,s1,ss1,m1) = user_move brd (x,y)
>       (newboard2,s2,ss2,m2) = comp_move newboard1
```

```
>|| Noughts and crosses game continued (Page 2 of 4)

State:

> state == (board, bool, bool, [char])

Board:

> abstype board
> with
>   empty_board   :: board    || in order to start the game
>   print_board   :: board -> [char]
>   game_over     :: board -> state
>   user_move     :: board -> (num,num) -> state
>   computer_move :: board -> state

Represent the board as a list of lists of cells,
where a cell is either Empty or contains a Nought or a Cross:

> cell ::= Empty | Nought | Cross
> board == [[cell]]

An empty board has 9 empty cells

> empty_board = [ [Empty,Empty,Empty],
>                 [Empty,Empty,Empty],
>                 [Empty,Empty,Empty]
>               ]

Print the board:

> print_board [row1,row2,row3]
>      = edge ++ printrow1 ++ edge ++ printrow2 ++ edge
>        ++ printrow3 ++ edge
>        where
>         edge = "-------------" ++ "\n"
>         printrow1 = (concat (map showcell row1)) ++ "|\n"
>         printrow2 = (concat (map showcell row2)) ++ "|\n"
>         printrow3 = (concat (map showcell row3)) ++ "|\n"
>         showcell Empty  = "|   "
>         showcell Nought = "| O "
>         showcell Cross  = "| X "
```

```
>|| Noughts and crosses game continued (Page 3 of 4)

Check for "game over".
  Find three Noughts or Crosses in a row, column or diagonal
  and then indicate who won (computer plays crosses).

> game_over brd
>     = (brd,over,over,msg)
>       where
>        over = cross_wins \/ nought_wins \/ board_full
>        cross_wins  = finished Cross
>        nought_wins = finished Nought
>        board_full  = ~(member (concat brd) Empty)
>
>        finished token = or (map (isline token) (rows ++ cols ++ diags))
>        isline x [x,x,x] = True
>        isline x [a,b,c] = False
>
>        rows = brd
>        cols = transpose brd
>        diags = [ [brd!0!0,brd!1!1,brd!2!2], [brd!0!2,brd!1!1,brd!2!0] ]
>
>        msg = "Computer won!\n", if cross_wins
>            = "Player won!\n",   if nought_wins
>            = "Board Full\n",    if board_full
>            = "",                otherwise

Add the user's move to the board.
  But first check if the new cell is on or off the board
      then  check if the new cell is already occupied.
  Then return a four-tuple indicating:
   (i)   the new board after the change has been made
   (ii)  a boolean to say whether the change was successful
   (iii) a boolean to say whether the program should terminate
   (iv)  a string containing an error message if the change failed.

> user_move brd (x,y)
>     = (newb,False,True,message2),    if over
>     = (newb,success,False,message1), otherwise
>       where
>        (newb,success,message1) = do_move brd (x,y) Nought
>        (b1,over,s2,message2) = game_over newb
```

```
>|| Noughts and crosses game continued (Page 4)

> do_move :: [[cell]]->(num,num)->cell->([[cell]],bool,[char])
> do_move brd (x,y) token
>      = (brd,False,"illegal move - off the board\n"),
>          if ~((0 <= x <= 2) & (0 <= y <= 2))
>      = (brd,False,"cell not empty\n"),
>          if (brd!y)!x ~= Empty
>      = (newb,True,""), otherwise
>        where
>         newb = simple_move brd (x,y)
>         simple_move [d,e,f] (x,0) = (newcol d x):[e,f]
>         simple_move [d,e,f] (x,1) = d:(newcol e x):[f]
>         simple_move [d,e,f] (x,2) = d:e:[newcol f x]
>         simple_move [d,e,f] (x,n)
>            = error (seq (force (system "stty sane"))
>                                  "out of bounds\n")
>         newcol [cell1,cell2,cell3] 0 = [token,cell2,cell3]
>         newcol [cell1,cell2,cell3] 1 = [cell1,token,cell3]
>         newcol [cell1,cell2,cell3] 2 = [cell1,cell2,token]
>         newcol [cell1,cell2,cell3] n
>             = error (seq (force (system "stty sane"))
>                                  "out of bounds\n")

Computer move:
    first check whether all positions are full,
    then choose where to put a cross:
    note: there is no attempt to win, just to play honestly!

> comp_move b  = (newb,False,True,message2),   if over
>              = (newb,success,False,message1),otherwise
>               where
>                (newb,success,x,message1) = computer_move b
>                (b1,over,s2,message2) = game_over newb
>
> computer_move brd
>     = (brd,False,True,"Board full\n"),
>         if ~(member (concat brd) Empty)
>     = hd [ (newb,success,False,m) | i,j <- [0..2];
>            (newb,success,m) <- [do_move brd (i,j) Cross];
>            (success = True) ], otherwise

draw successful moves from a list containing
the results of moves for all values of i and j
```