Chapter 5

Environments and Evaluation

For the solution of large-scale programming problems, it is essential that the programmer takes a *modular* approach; first the problem must be subdivided into manageable sub-problems, then each sub-problem is solved and the results combined to provide the overall solution to the problem. This itself may be a recursive process requiring several subdivisions and recombinations.

Hughes (Hughes, 1984) argues that the ways in which one can divide up the original problem depend directly on the ways in which one can combine the solutions of the sub-problems in order to arrive at the overall solution. Hughes provides an attractive analogy with carpentry: a carpenter knows how to make joints to enable pieces of wood to be combined to make a single object (a chair, perhaps)—without the various kinds of joint, the carpenter would have to carve a chair out of a solid block of wood (a much harder task!). The diversity of joints available to the carpenter determines the diversity of objects which can be built with ease (that is, without resort to solid carving). Similarly, the diversity of the methods of combination available to a programmer determines the diversity of problems which may be solved with ease.

This chapter discusses two methods of combination that have an important impact on the structure and design of programs:

1. The organization and control of environments.
2. The use of lazy evaluation.

Organizing environments (by means of Miranda's **where** mechanism and by *list comprehensions*) gives the programmer a means of grouping dependent functions and identifiers into a coherent programming block. The programmer may control the "visibility" of function names, thereby controlling the parts of the program where they can and cannot be used; this is a form of *encapsulation*, which makes the resultant code safer, easier to understand, easier to maintain and more reusable. The exploitation of lazy evaluation and infinite lists builds on the discussion in Section 3.2.4 to give programmers a new way of thinking about problems. Functions and environments may be combined in new ways; thus, laziness augments the available methods of combination in the language and provides new and elegant mechanisms for structuring programs.

5.1 Environments

When a Miranda script file is compiled, the system builds what is known as an *environment*, which comprises all the script file definitions and all the definitions from the Standard Environment (as given in Section 28 of the on-line manual). Each definition is said to *bind* a name to a value; this is often referred to as a *binding*. When an object is defined it has access to all other bindings in the environment. Any subsequent re-edit repeats this process; it removes the old definitions and constructs a new environment by adding the new definitions to those given in the Standard Environment.

When a function is defined, its body has a *new* environment consisting of the environment defined above (often referred to as the "inherited environment") together with the names of the function's formal parameters (which are only bound to actual values when the function is applied). This new environment is specific to the function body and does not affect subsequent function definitions.

Free and bound names

In order to determine the value of a name that appears in a function's body, one of two rules is applied:

1. Any appearance of a formal parameter name takes its value from the actual parameter to which the function is applied. In other words, the local environment ignores any previous binding for that name in the inherited environment. Formally, the name is said to be *bound* in that function. This rule also holds for the name of the function itself.
2. Any other name that appears in the function body takes its value from the inherited environment; formally, it is said to be *free* in that function. Any function that does not have any free values is said to be closed or to exhibit *closure*.

Scope

The region of program text in which a binding is effective is known as its *scope*. Thus, the two rules presented above are often known as the *rules of scope*.

Outside the scope of a particular binding, the name for that binding has no value and any reference to the name will result in an error (as will be demonstrated below). The only exception to this rule is where a name is reused in a function-body environment—in this case, if the name is used inside the function body it has one value, and if it is used outside the function body may have a different value. It should be clear that the redefinition of names in this way is not advisable because of the confusion that can arise.

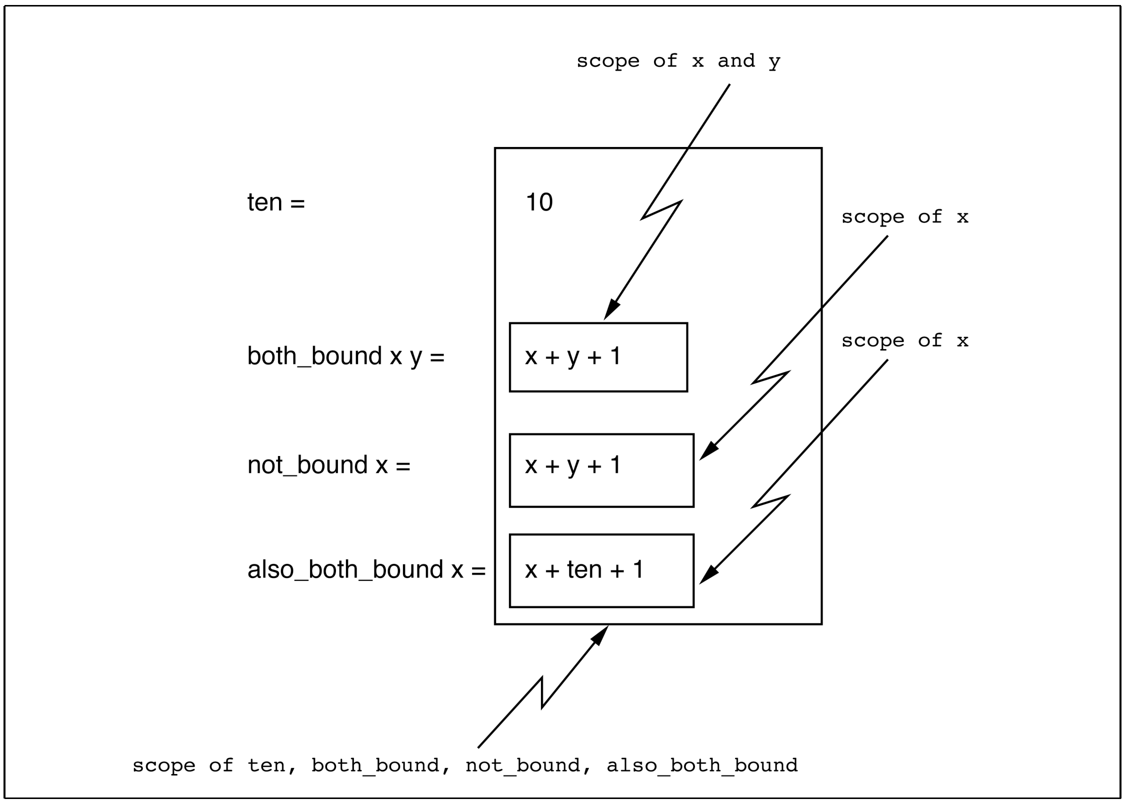


Figure 5.1 The scope of bindings.

Notice that the environment of an expression (that is, the set of names that are accessible to the expression) depends only on the *textual position* of the expression in the program.

The terms "bound" and "in scope" are often interchangeable, as are the terms "unbound" and "out of scope". For example, if a name that is free in a function does not appear in the inherited environment then Miranda will report that it is undefined, as shown in Figure 5.1. On exit from the editor, Miranda will report the error message and *all* bindings from the faulty script will be discarded, for example:

```
compiling script.m
checking types in script.m
(line   5 of "script.m") undefined name "y"
```

The error message displayed above means that Miranda cannot find a meaning for `y` in the environment of the function `not_bound`, although Miranda may have a meaning for `y` in the local environment of another function. In the above case, there was a value associated with the name `y`, but this was bound in the function `both_bound`.

What can be bound?

Reserved names[1] *cannot* be re-bound, whilst it is recommended that function names and type names *should not* be re-bound. Thus, the following example, which rebinds predefined names, is legitimate but deplorable, in that it is not immediately obvious what the function does:

```
nasty  [] char = False
nasty  (hd :  tl) char
    =  True, if hd = char
    =  nasty tl char, otherwise
```

Even though `char` is a basic type and `hd` and `tl` are functions that are normally provided by Miranda, their original meanings are suspended throughout the scope of the function `nasty`. The function happens to work because the names `char`, `hd` and `tl` are bound as formal parameters in the local environment for the function `nasty`. However, it must again be stressed that this style is dangerous and definitely *not* recommended.

5.1.1 The need for encapsulation

The next part of this chapter shows the need to restrict the scope of identifiers and functions:

1. To avoid name clashes.
2. To link together closely-related functions.

Restricting the scope of an identifier or a function implies that it can only be used in a small part of the program. This provides the ability to structure the program as several smaller sub-programs. The process of building a sub-program which contains identifiers and functions which cannot be used by the rest of the program is often known as *encapsulation*.

[1]See Appendix A.

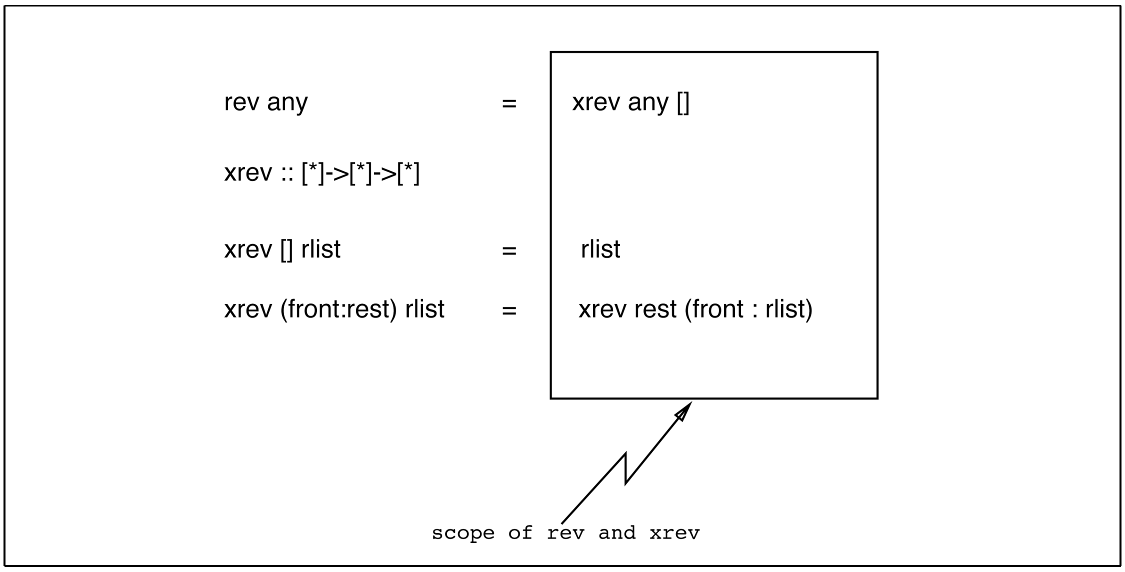
Name clashes

For small programs, any potential name clashes can easily be avoided. The development of small programs is normally only undertaken by one person who can keep track of the limited number of names. For larger programs, it is much harder to guarantee the uniqueness of names because of their increased number and the fact that more than one programmer will normally be involved in the software development process. In this context, it is certainly unreasonable to expect a programmer to invent new names for every new object in the system. Similarly, programmers should not be expected to know the names of objects that they are not directly interested in.

The next part of this chapter shows that unique names are not really a problem because names can be limited to a particular scope; that is, programmers can control their own program environment.

Linking related functions

A function's formal parameters are only in scope within the function body; this adds a little to the environment but this does not give much control. What is really needed is a means of defining a function or value to have a limited scope; that is, to have expressions that are not generally free in the script file, but are in scope only for a particular function body.



```
rev any                  =   xrev any []

xrev :: [*]->[*]->[*]

xrev [] rlist            =   rlist

xrev (front:rest) rlist  =   xrev rest (front : rlist)
```

Figure 5.2 A subsidiary function with global scope.

The simple example of `rev`, shown in Figure 5.2, makes use of an auxiliary accumulative function `xrev`. The latter is not robust (as it does not validate its parameter), nor is it reasonable to expect someone to know that it requires its second parameter to be instantiated to `[]`. The next section will discuss how to restrict the scope of (or *encapsulate*) `xrev` so that it can only be used by `rev`.

5.2 Block structure: where expressions

The requirement for encapsulation can be met by binding `xrev` in the local environment for the function body of `rev`, such that the existence of `xrev` is concealed from any other function. In Miranda this may be achieved by using the keyword **where** as shown in Figure 5.3.

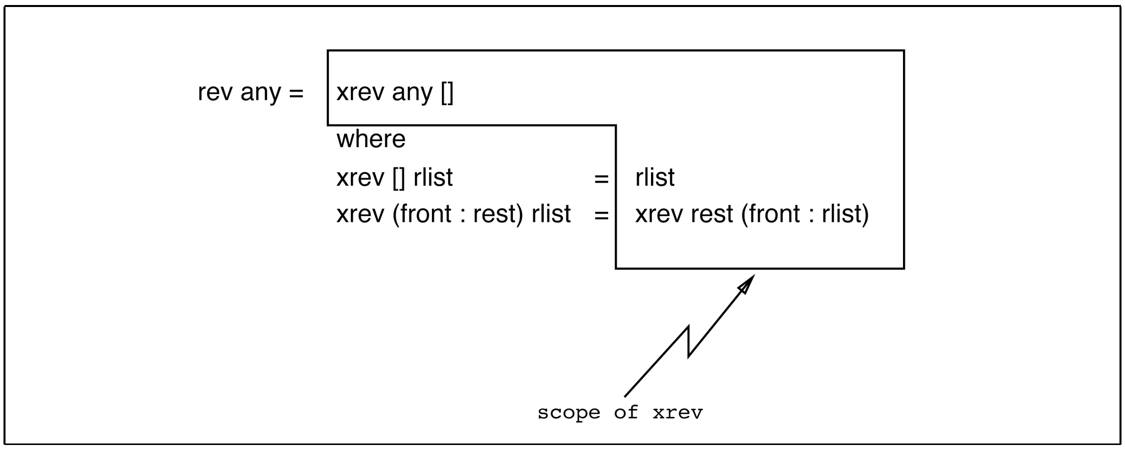


Figure 5.3 A subsidiary function with restricted scope.

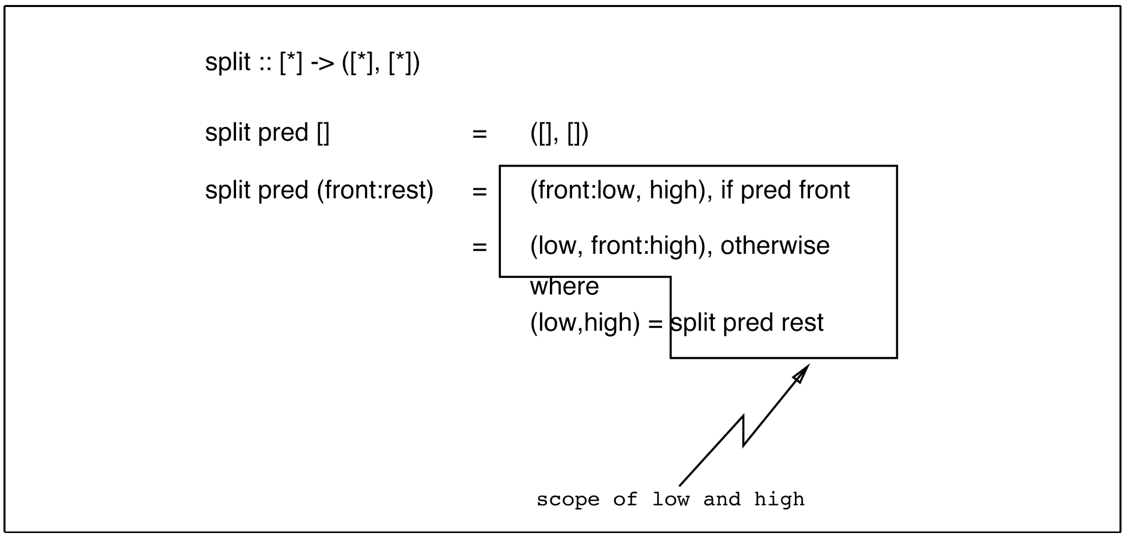
The fact that `xrev` is bound in the function `rev` is emphasized by the fact that any attempt to use it on its own will give rise to an error:

```
Miranda xrev [1,2,3] []
UNDEFINED NAME - xrev
```

Use of the keyword **where** follows the "offside rule" introduced in Chapter 2. It may appear after the function body for any particular pattern and is in scope for that entire function body, including guards, but is not in scope for the function bodies of other patterns for the same function definition. These rules are illustrated in the rest of this section.

Shortening expressions

The following example shows one of the simplest uses of **where**, which is to make code more readable by replacing long expressions with shorter ones. In Figure 5.4, the single names `low` and `high` replace the unwieldy expressions `(fst (split pred rest))` and `(snd (split pred rest))`. Once again, the new binding is not available outside the restricted scope.



```
split :: [*] -> ([*], [*])

split pred []              =    ([], [])

split pred (front:rest)    =    (front:low, high), if pred front

                           =    (low, front:high), otherwise
                                where
                                (low,high) = split pred rest
```

Figure 5.4 The scope of names in a **where** block.

Notice that the local `(low, high)` definition is available to both guards for the pattern `(front : rest)`, but is *not* available to the function body for the first pattern `[]`.

Multiple definitions within a where clause

There is no restriction on the number of definitions that can appear within a **where** expression and, as expected, if more than one definition appears then the **where** sub-environment is built in exactly the same way as the top-level environment. This is demonstrated in the following program (part of a text justifier) which throws away all leading "white space" characters in a string and which compresses all other occurrences of multiple white space characters into a single space:

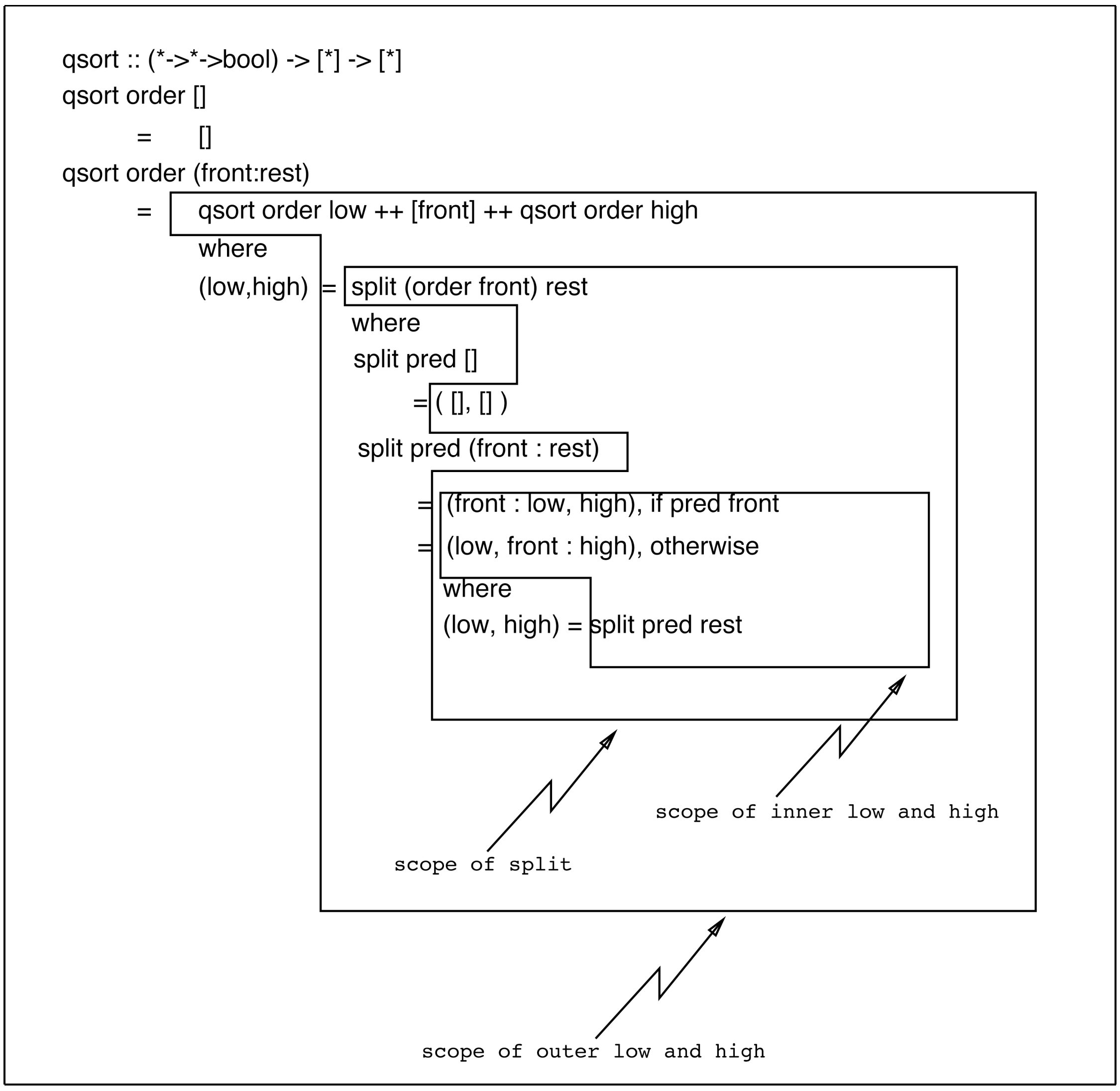
```
string == [char]
compress ::  string -> string
compress astring
    = (squeeze .  drop) astring
      where
      space = ' '
      tab = '\t'
      newline = '\n'

      isspace c
        = (c = space) \/ (c = tab) \/ (c = newline)

      || this is not the Standard Environment drop
      drop [] = []
      drop (front :  rest)
         = drop rest, if isspace front
         = (front :  rest), otherwise

      squeeze [] = []
      squeeze (front :  rest)
        = space :  squeeze (drop rest), if isspace front
        = front :  squeeze rest, otherwise
```

Nested where clauses

Local **where** clauses may be nested; perhaps the major use is to reflect the top-down design of a program, whilst restricting the scope of all of the auxiliary functions. The offside rule applies to the nested **where** block in the same way that it applies to the first **where** block. The following reworks the `int_to_string` conversion program of Chapter 2:

```
string  == [char]
int_to_string ::  num -> string
int_to_string n
     =  "-" ++ pos_to_string (abs n), if n < 0
     =  pos_to_string n, otherwise
        where
        pos_to_string n
            =  int_to_char n, if n < 10
            =  pos_to_string (n div 10)
               ++ (int_to_char (n mod 10)), otherwise
               where
               int_to_char n = decode (n + code '0')
```

Another example of the use of nested **where** clauses is given in the implementation of *quicksort* as shown in Figure 5.5. Quicksort is a recursive sorting algorithm which works by choosing one of the values in the list as a pivot; the list is then split into a sublist of all values less than the pivot, the pivot itself and a sublist of all values greater than the pivot; the quicksort algorithm is then applied to each of the two sublists just generated (choosing a new pivot for each sublist). Empty sublists are ignored and eventually every value will have been considered as a pivot—all the pivots are now in order and can be collected together to form the sorted list, as required.[2]



```
qsort :: (*->*->bool) -> [*] -> [*]
qsort order []
        =    []
qsort order (front:rest)
        =    qsort order low ++ [front] ++ qsort order high
             where
             (low,high) = split (order front) rest
                          where
                          split pred []
                                  = ( [], [] )
                          split pred (front : rest)
                                  = (front : low, high), if pred front
                                  = (low, front : high), otherwise
                                  where
                                  (low, high) = split pred rest
```

Figure 5.5 The scope of names in nested **where** blocks.

[2]For more details see (Standish, 1980).

The implementation uses the split function to decompose the unsorted list, to yield a tuple pair of those values that satisfy the ordering function and those that do not.[3] The example demonstrates how **where** expressions may be nested, with the effect that the names low and high can be used in two different contexts. In general, this is considered to be bad style because of the potential confusion over names. However, the example serves to illustrate that it is possible, and indeed this poor naming policy is unfortunately not uncommon amongst programmers.

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".

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.

Sharing common subexpressions

Using **where** can also produce more flexible code when a subexpression appears more than once in another expression. For instance, the function oklength checks to see if a given string is of at least a specified size lowerbound and not larger than upperbound:

```
oklength  ::  [*] -> num -> num -> bool
oklength  s lowerbound upperbound
       =  (# s) >= lowerbound & (# s) <= upperbound
```

However, (# s) appears twice and might have to be altered twice if the function specification changed. If a **where** definition is used then the alteration can be done in one place only, thereby making the programmer's task easier and the code safer.

[3]To sort integers in ascending order, the ordering function should be isitlessthan since it is used in a partially applied manner within split.

```
oklength  ::  [*] -> num -> num -> bool
oklength  s lowerbound upperbound
       =  slen >= lowerbound & slen <= upperbound
          where slen = # s
```

The new version is not only easier to read, but will execute more efficiently, since the common subexpression will only be evaluated once.[4] The advantage of this modular approach becomes increasingly more apparent as the subexpression occurs more often in the main expression.

Eliminating unnecessary parameters

The following curried version of the `sublist` function from the *grep* program (shown in Chapter 3) uses an auxiliary function `xsublist`, whose binding is confined to `sublist` but does not carry around the `regexp` parameter on each of its recursive applications. Although `regexp` is free in `xsublist`, it is in scope and hence accessible to `xsublist`:

```
string == [char]

mtype == string
regtype == (mtype, char)
reglist == [regtype]

sublist ::  reglist -> string -> bool
sublist  regexp line
      =  (startswith regexp line) \/ xsublist line
         where
         xsublist  []
                =  False
         xsublist  (any :  lrest)
                =  (startswith regexp lrest)
                   \/ (xsublist lrest)
```

This is quite a popular functional programming style because it is generally easier to read functions with less parameters. Unfortunately, this style can be abused. The following definition of `unclearmap` also saves on passing a parameter (`ff`), but may be considered less clear because it is not obvious from where `xmap` expects its list parameter:

[4]This beneficial feature is known as *sharing* and is a form of lazy evaluation, as will be shown in Section 5.4.

```
unclearmap ::  (* -> **) -> [*] -> [**]
unclearmap ff
=  xmap
   where  xmap [] = []
          xmap (front :  rest) = (ff front) :  xmap rest
```

A clearer version would include the list parameter:[5]

```
clearmap ::  (* -> **) -> [*] -> [**]
clearmap ff anylist
=  xmap   anylist
   where  xmap [] = []
          xmap (front :  rest) = (ff front) :  xmap rest
```

5.2.1 A larger example of where blocks

The following program (adapted from (Park and Miller, 1988)) generates a list of random numbers and illustrates the use of **where** to provide a local (restricted) environment.

```
>||Literate script (Page 1 of 2 pages)

Program to generate a list of random real numbers
for systems with 46-bit mantissas, with a function
to constrain the result to integer limits

generate N random numbers
and a new seed from an initial seed

>randomize ::  num->num->(num,([num],num))
>randomize n firstseed
>      = error "randomize:  negative number", if n < 0
>      = genlist n firstseed [], otherwise
>        where
>        c1 = 16807.0
>        upperbound = 2147483647.0
>        nextseed seed
>            = (c1 * seed) - ( upperbound
>               * (entier ((c1 * seed)/upperbound)) )
>        genlist 0 seed ranlist
>            = (seed, (ranlist,upperbound))
>        genlist n seed ranlist
>            = genlist (n-1) (nextseed seed) (seed:ranlist)
```

[5]Notice that the types of `unclearmap` and `clearmap` are identical.

```
>||Literate script continued (Page 2)

constrain real number to an acceptable integer range

>constrain ::  num->([num],num)->[num]
>constrain k (rlist,u)
>        = map (xconstruct k u) rlist
>          where
>           xconstruct k u r = 1 + entier ((r / u) * k)

the function could be used in the following manner:
(constrain 3) (snd (randomize 50 1.1))
This creates a list of 50 random numbers,
in the range 0 to 3, with an initial seed of 1.1
```

Type expressions

Note that type expressions may *not* appear inside **where** blocks. Thus, in the above example, it is unfortunately not possible to specify the type of the function `genlist`.

5.3 List comprehensions

This section presents another method of restricting the scope of identifiers; the "list comprehension" facility which is a way to specify the elements of a list through calculation (*intensionally*), rather than by explicitly stating each element (*extensionally*).[6]

Simple list comprehension definition and usage

The template for a simple list comprehension consists of an *expression* which is used to calculate the components of the resultant list, together with an *iterative construct* which defines successive values for a local *identifier* by providing these values as a list. The local identifier is only in scope in the *expression* (it is *encapsulated*); for each successive value of the local identifier, the expression is recalculated to provide a new component of the resultant list:

[expression | *identifier* `<-` *value_list]*

[6]In this respect, it is similar to the "dotdot" notation shown in Chapter 3.

It helps to pronounce the vertical bar as “such that” and the arrow as “is drawn from”. Any expression may occur to the left of the vertical bar. For example:

```
squares3 = [ x * x | x <- [1,2,3]]
```

This reads as:

> *“the list of all values $x * x$*
> *such that x is drawn from the list* `[1,2,3]`*”.*

The values for the local identifier are drawn from the *value_list* in order, from left to right. In this case, initially `x` gets the value `1` and the first value of the resultant list is calculated as $(1 * 1 = 1)$, then `x` gets the value `2` and the next element of the result list is calculated as $(2 * 2 = 4)$ and finally `x` gets the value `3` and the final element is calculated as $(3*3 = 9)$. Hence entering `squares3` at the Miranda prompt gives the list `[1,4,9]`.

The list comprehension is thus a shorthand for an anonymous function over a list. The above could have been rewritten with a named local function:

```
squares3 =  xsquares3 [1,2,3]
            where
            xsquares [] = []
            xsquares (front :  rest)
                   = (front * front) :  xsquares rest
```

The above example can be generalized to give a list of squares for any given number:

```
squaresN n = [ x * x | x <- [1..n]]
```

Inside this list comprehension `x` is a local identifier and `n` is free but in scope (because it is bound to the actual parameter for `squaresN`).

The next example is interesting in that the local identifier is only needed as a dummy to control the iteration; the number of dots is determined by the number of times the expression `'.'` is recalculated, which is controlled by the number of different values for the local identifier `j`, which is controlled by the function parameter `n`.

```
printdots n = ['.'  | j <- [1..n]]
```

The elegance of simple list comprehensions is further demonstrated in the following definition for the higher order function `map`:

```
map f itemlist = [f item | item <- itemlist]
```

The function `map` and the simple list comprehension often provide equally concise definitions of the same function and may be used interchangeably, for example:

```
squares3 = [ x ^ 2 | x <- [1,2,3]]

squares3 = map (^ 2) [1,2,3]
```

General list comprehensions

The template for a general list comprehension is:

> *[expression | qualifiers]*

A general list comprehension produces the list of all elements given by *expression* such that the *qualifiers* hold. There may be many *qualifiers*; if there are more than one, they must be separated by a semicolon (pronounced "and"). There are two forms of qualifier:

1. A generator, which provides successive values for a local identifier (as illustrated in the previous section). Each local identifier is in scope in *expression*, in the qualifier wherein the identifier is defined, and in all qualifiers to the *right* of that defining qualifier.
2. A filter, which is a Boolean expression to restrict the range of the values given to a local identifier defined in a generator to the *left* of the filter.

The following examples (the first taken from the Miranda On-line Manual)) demonstrate further aspects of list comprehension as a mechanism to create a local environment:

1. Multiple qualifiers.
2. Multiple local identifiers.
3. Recursive list comprehensions.

The function `factors` shows the use of a qualifier in conjunction with a filter:

```
factors n = [ r | r <- [1..(n div 2)]; n mod r = 0]
```

This can be read as:

> *The list of all elements r*
> *Such that*
>   *r is drawn from the list of integers*
>   *starting at 1 and ending with (n div 2)*
> *And*
>   *(n mod r) is equal to 0*

If `n` in the above example is `9` then `r` will be drawn from the list `[1,2,3,4]`, but only those values for which `(9 mod r) = 0` will be allowed. Thus, `r` will take the successive values `1` and `3` and the result will be the list `[1,3]`.

Note that the rules of scope for the local identifier `r` mean that the qualifiers could not have been written in reverse order. The following examples compare two versions with `n` set to the value `17`:

```
Miranda [r |  r <- [1..(17 div 2)]; 17 mod r = 0]
[1]

Miranda [r | 17 mod r = 0; r <- [1..(17 div 2)]]
[
UNDEFINED NAME - r
```

Multiple generators represent nested loops, with the leftmost generator being the outermost loop:

```
cartesianA n m = [ (x,y) | x <- [1..n]; y <- [1..m] ]
```

This is read as:

The list of all two-tuples (x, y)
Such that
  x is drawn from the list [1..n]
And
  y is drawn from the list [1..m]

In the above example, the leftmost generator for `x` performs the outermost loop and so the resultant list for `cartesianA 2 3` is `[ (1,1), (1,2), (1,3), (2,1), (2,2), (2,3) ]`. By contrast, with the definition:

```
cartesianB n m = [ (x,y) | y <- [1..m]; x <- [1..n] ]
```

the resultant list for `cartesianB 2 3` is `[(1,1), (2,1), (1,2), (2,2), (1,3), (2,3)]`.

If two local identifiers are drawn from the same list then a shorthand notation is available; the single generator `x,y <- [1..3]` is equivalent to the two generators `x <- [1..3]; y <- [1..3]`. Hence:

```
squarecoords n = [ (x,y) | x,y <- [1..n] ]
```

This can be read as:

The list of all two-tuples (x, y)
Such that
  x is drawn from [1..n]
And
  y is drawn from [1..n]

Hence `squarecoords 2` evaluates to: `[(1,1),(1,2),(2,1),(2,2)]`.
The final example (also from the Miranda On-line Manual) generates all the possible permutations for a given list, demonstrates a list comprehension with more than one local identifier, more than one local calculation and also involves a recursive application of `permutations`. Notice that the local identifiers in a list comprehension come into scope from left to right, so that a generator can make use of identifiers already defined to its left, but not vice versa.

```
permutations  [] = [[]]
permutations  anylist
           =  [ front :  rest | front <- anylist;
              rest <- permutations(anylist--[front]) ]
```

The list comprehension in this example can be read as:

The list of all lists (front : rest)
Such that
  front is drawn from the list anylist
And
  rest is drawn from the result of the recursive call to permutations,
  when applied to the argument anylist -- *[front])*

Exercise 5.3
Define a list comprehension which has the same behaviour as the built-in function `filter`.
Exercise 5.4
Rewrite *quicksort* using list comprehensions.

5.4 Lazy evaluation

This section explores a style of programming that is not generally available in other languages: programming by means of infinite lists of data, together with the use of *generator* functions and *selector* functions.

5.4.1 Lazy evaluation of lists

As has been shown in previous chapters, Miranda is a *lazy* functional language; this means that parameters to functions are evaluated only when necessary and that shared subexpressions are never evaluated more than once. Furthermore, the items in a list are only evaluated if their values are necessary to produce a result. In other words, Miranda will not always fully evaluate a function's environment. For example, in the following Miranda session:

```
Miranda const 3 (4 / 0)
3
```

the `(4 / 0)` is not needed and so is not evaluated.

Similarly:

```
Miranda hd [3, (4/0)]
3
```

Evaluation of a list has two distinct stages: a function such as `hd` may cause the first item of a list to be evaluated, but does not need to evaluate the rest of the list; by contrast, a function such as `length` may need to know the number of items in a list without needing to evaluate the items in the list. Thus:

```
Miranda length [3, (4/0), 5]
3
```

Lazy evaluation implies that both evaluation stages of a list are only undertaken as necessary. Since the tail of a list may never be evaluated, it is possible to exploit this fact to define *potentially infinite lists* (perhaps using a recursive function) and then only use that part of the list that is required. For example:

```
natsfrom n = n :  natsfrom (n + 1)
```

This will generate the list `(n :  (n + 1) :  (n + 2) :  ...)`. If an expression such as `natsfrom 1` were typed at the Miranda prompt then Miranda would attempt to print out the entire list and will continue to do so until the user forcibly interrupts the process or Miranda suffers some internal error.[7]

Clearly `natsfrom` on its own is useless, but when combined with other functions it becomes very powerful. This is shown in the following higher order function `make_index_pairs` which takes a list of items and produces a list of pairs, where each item has been paired with an index number. The function `natsfrom` is used to generate an infinite list of indices (because `make_index_pairs` must work on input lists of any length) and `zip2` ensures that just enough of the infinite list is selected in order to give an index to each item:

```
make_index_pairs ::  [*]->[(num,*)]
make_index_pairs names = zip2 (natsfrom 1) names

|| where zip2 is defined in the Standard Environment
|| and is similar to map_two defined in Chapter 4
```

```
Miranda make_index_pairs ["chris","colin","ellen"]
[(1,"chris"),(2,"colin"),(3,"ellen")]
```

It is worthwhile comparing this definition to a recursive version where a subsidiary function `makepairs` must explicitly control the index number generation:

[7]Remember that Miranda has infinite-precision integers, which means that it will generate integers until the system memory is no longer big enough to represent the integer.

```
make_index_pairs ::  [*]->[(num,*)]
make_index_pairs names
      = makepairs 1 names
        where
         makepairs n [] []
         makepairs n (front2 :  rest2)
            = (n, front2) :  (makepairs (n + 1) rest2)
```

Note that the two versions exhibit exactly the same behaviour and have exactly the same type.

5.4.2 Infinite "dotdot" lists

Miranda recognizes the utility of infinite lists by providing a special form of the "dotdot" notation (introduced in Section 3.2.4) whereby potentially infinite lists are specified by omitting the final number from the "dotdot" expression. Thus `[1..]` represents the potentially infinite list of ascending integers starting at `1`, and `[2,4..]` represents the potentially infinite list of ascending even numbers starting at `2`.

The function `make_index_pairs` can now be defined using this notation:

```
make_index_pairs names = zip2 [1..]  names
```

This can be compared favourably with the previous higher order definition, which required an additional definition for `natsfrom`. It is also less cluttered than the following definition using finite "dotdot" notation, which requires identification of the `names` parameter and then explicit calculation of its length:

```
make_index_pairs names = zip2 [1..#names] names
```

Finally, the infinite dotdot version could be made more elegant by defining the name `make_index_pairs` to be an alias for the partial application of `zip2`:

```
make_index_pairs = zip2 [1..]
```

Infinite list comprehensions

Not surprisingly, infinite "dotdot" lists are often used in conjunction with list comprehensions, for example:

```
oddsquares = [ x * x | x <- [1,3..]]
```

which returns a list of all the squares of odd numbers.

The first `n` items could be selected using `take`:

```
Miranda take 3 oddsquares
[1,9,27]
```

An equivalent recursive program is:

```
oddsquares =  xodds 1
              where
              xodds n = (n * n) :  xodds (n + 2)
```

which would be applied in exactly the same manner.

The next example introduces a variation on the basic mechanism, where a generator may be expressed in terms of itself (this is sometimes called a *recurrence relation*). A recursive generator uses the syntax *n <- exp1, exp2 ..* and is similar to the dotdot expression *[exp1,exp2 ..]*; except that there are no square brackets, and *exp2* is defined recursively in terms of *n*:

```
powers_of_two = [n | n <- 1, 2 * n ..]
```

The above list comprehension can be read as:

The list of all items n
Such that
  n is drawn from the list of values
  starting with 1 and calculating each new value
  by multiplying the current value of n by 2.

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.

5.4.3 Generators and selectors

In general, lazy evaluation makes it possible to modularize a program as a *generator* function, which constructs a large number of possible answers (perhaps as a potentially infinite list), and a *selector* function, which chooses the appropriate answer. The selector function is applied to the output of the generator function; this is often achieved using function composition. This section takes a further look at this programming style, in conjunction with list comprehensions, infinite lists and also finite but very large lists. As with higher order functions (Section 4.2), there are no rigid rules which style or approach to use. Once again the best advice is to choose the definition which most closely mirrors the natural specification and then optimize if necessary.

Grep as a generator

In Section 4.3.3, the `grep` function was extended so that it would inspect a list of lists of characters (representing many lines of text) and return those lines which contain a given pattern. Here, the code for `grep` is repeated:

```
string == [char]

grep ::  string -> [string] -> [string]
grep regexp = filter (xgrep regexp)

xgrep ::  string -> string -> bool
xgrep regexp line = sublist (lex regexp) line
```

Assume that the identifier `manual` is bound to a list of lists of characters; the `grep` function can now be utilized to provide a list of all lines in `manual` in which the word `"evaluation"` appears. For example:[8]

```
Miranda grep "evaluation" manual
["Lazy evaluation is elegant.",
 "evaluation of arguments is also an",
 "It also optimizes evaluation."]
```

The above application of `grep` can be viewed as a generator of data; it inspects all of the text in `manual` and returns the appropriate data. Parts of that data can now be selected by a separate function. For example, to select the *first* line in `manual` which mentions `"evaluation"`:

```
Miranda hd (grep "evaluation" manual)
"Lazy evaluation is elegant."
```

Here, the function `grep` generated data and the function `hd` selected a part of that data. Lazy evaluation ensures that `grep` only inspects the text in `manual` as far as necessary to generate *one* line of output data (which is all that `hd` needs).

Interestingly, the same function can be used as both a generator and a selector. For example, to select all lines in `manual` which mention both the word `"evaluation"` *and* the word `"also"`:

[8]Miranda would actually print the output all on one line, but it is split across several lines here to make it clearer.

```
Miranda grep "also" (grep "evaluation" manual)
["evaluation of arguments is also an",
 "It also optimizes evaluation."]
```

However, in this case, because `grep` in its role as a selector must inspect all of its argument, then so must `grep` in its role as a generator.

Pipelining generators and selectors

Generator functions and selector functions are often combined using function composition to produce a "pipeline". There may be more than one selector function in the pipeline, as illustrated by the following session which calculates the length of the first line in `manual` which mentions both of the words `"evaluation"` and `"also"`:

```
Miranda (# . hd . (grep "also")) (grep "evaluation" manual)
34
```

In this case it is also possible to include the generator in the function composition:

```
Miranda (# . hd . (grep "also") . (grep "evaluation")) manual
34
```

In these examples, `hd` only requires the first part of the data from `(grep "also")`, which therefore only needs sufficient data from `(grep "evaluation")` in order to determine the first line which contains the word `"also"`. Thus, lazy evaluation ensures that only a small part of the text in `manual` is inspected.

The lazy evaluation of generators also applies where the generator is a list comprehension. In the following example, only the first square is calculated by the list comprehension because that is all that the selector `takewhile (< 20)` requires:

```
Miranda takewhile (< 20) [ x * x | x <- [2,4..2000] ]
[4,16]
```

The above expression can be rewritten with the selector inside the list comprehension:

```
Miranda [ x * x | x <- [2,4..2000] ; x * x < 20]
[4,16]
```

However, in the above expression all of the squares from 4 to 4,000,000 are calculated! This is because *selectors inside a list comprehension cannot affect the evaluation of generators inside the same list comprehension.*

5.4.4 Pitfalls for the unwary user of laziness

The ability to specify potentially infinite computation and potentially infinite lists is an extremely powerful tool which must be used with care. There are three major potential pitfalls when employing a selector approach to restricting a lazily generated list:

1. Looking beyond the solution.
2. Never reaching the solution.
3. Losing laziness, and thereby doing too much computation.

Looking beyond the solution

A potential mistake with list comprehensions is to write a function which produces a list containing the required items but continues to look for more items and never terminates because no more exist! An example of this kind of error is now considered:

```
neverending_cubes_lessthan_fifty
         = [x ^ 3 | x <- [1..]; x ^ 3 < 50]
```

Whilst it is quite clear that there are only three cubes which are less than fifty, the function has been defined to consider *all* possible cubes and test *every* one of them to see if it is over fifty. Unfortunately, the function will produce the result `[1,8,27` and will then “hang” whilst it continually loops and tests all the other generated cubes.

As explained in the previous section, this error occurs because a filter inside a list comprehension cannot be used to modify the number of times a generator (in the same list comprehension) loops—it can only determine which of the values produced by the generator will be valid for the local identifier.

However, the error of looking beyond the solution is not restricted to the use of list comprehensions. Here is an example which abuses function composition:

```
also_neverending_cubes_lessthan_fifty
          = ((filter (<50)) .  (map (^ 3))) [1..]
```

This function attempts to generate the infinite list of numbers and then to cube all of these numbers and finally to test each cube to check if it is less than fifty; in other words, it loops forever.

In order to correct this error, it is necessary to make use of the fact that each successive cube will be larger than the previous one, and so as soon as one cube fails the test then all subsequent cubes will also fail the test:

```
cubes_lessthan_fifty
       = ((takewhile (< 50)) .  (map (^ 3))) [1..]
```

The function `takewhile` terminates as soon as it finds an element that is less than fifty. The values of the remaining cubes are never needed; they will never be generated and so there will be no infinite computation.

Never reaching the solution

The above discussion showed the dangers of finding the result and going beyond it; however, sometimes the result may never be reached. This mainly arises with list comprehensions that have more than one generator ranging over an infinite list. For example, the function `wrong_triangles` is meant to produce the infinite list of three-tuples (triples) which represent the lengths of the sides of right-angled triangles. However, it fails to produce a single element of that list!

```
wrong_triangles
    = [(a,b,c) | a,b,c <- [1..]; a ^ 2 + b ^ 2 = c ^ 2]
```

To understand the behaviour of the above function, recall the following:

1. The construction `a,b,c <- [1..]` is an abbreviation for the three terms `a <- [1..]; b <- [1..]; c <- [1..]`.
2. Multiple generators represent nested loops, with the leftmost generator controlling the outermost loop. Thus, in the above example, the values generated for `(a,b,c)` will be, in order: `(1,1,1), (1,1,2), (1,1,3), (1,1,4)` and so on until all of the (infinite) values of `c` have been considered.
3. Because the innermost loop for `c` is infinite, the values for `(a,b,c)` where `a` or `b` are greater than `1` *are never considered.*
4. There is no solution to the problem for `a = 1` and `b = 1`, so the function will loop forever.

In recognition of the above problem, Miranda provides a "diagonalizing" list comprehension which combines multiple generators in a different order. If there are two generators `a,b <- [1..]`, the diagonalizing list comprehension tries values in the following sequence: (1,1), (1,2), (2,1), (1,3), (2,2), (3,1), (1,4), (2,3), (3,2), (4,1), (1,5), (2,4), (3,3), and so on. A diagonalizing list comprehension uses the same syntax as a normal list comprehension, *except* that the vertical bar is replaced by two forward-slashes:

```
triangles = [(a,b,c) // a,b,c <- [1..];
                        a^ 2 + b^ 2 = c^ 2]
```

The above definition of `triangles` will still loop forever, but it will test combinations of `a`, `b` and `c` in a "reasonable" order so that solutions are actually generated:

```
Miranda triangles
[(4,3,5),(3,4,5),(8,6,10),(6,8,10),(12,5,13) ...
```

The result is still an infinite list, but one that can generate results that can be used by a selector function.

Losing laziness

It is sometimes tempting to assume that, no matter how a program is written, somehow the Miranda system will always work out the most lazy (that is, least computationally expensive) solution. This is not the case! Unfortunately, it is quite easy to write a selector in such a way (perhaps using pattern matching) that it forces evaluation; hence it does unnecessary computation and so takes an unreasonably long time to execute.

For example, consider the following program which illustrates the use of a generator and several selectors in order to produce the thirteenth prime number:

```
primes12 =  selectors generator
            where
             generator = [2..1000]
             selectors = ((!12) .  (foldr sieve []))
             sieve x [] = [x]
             sieve x any
                = x :  (filter ((0 =).(mod x)) any)
```

In the above example, the dotdot expression is the generator; it generates a list of numbers between 2 and 1,000. The `primes12` function applies the selectors to the result of this generator. The first selector is `foldr sieve []`, which distributes the function `sieve` across the list, and so performs a standard "sieve of Eratosthenes" to produce a list of prime numbers (assuming that the thirteenth prime is less than 2000). The final selector is `(!12)`, which is a prefix sectional form of the list indexing operator (recall that (!0) selects the first item in a list). The function works as follows:

```
Miranda primes12
41
```

Initially, it appears that if `generator` is replaced by an infinite dotdot list then `selectors` can be generalized so that it can pick an arbitrary *nth* prime number:

```
nth_prime n =  selectors generator
               where
                generator = [2..]
                selectors = ((!n) .  (foldr sieve []))
                sieve x [] = [x]
                sieve x any
                   = x :  (filter ((0 =).(mod x)) any)
```

Lazy evaluation should ensure that the generator will only do as much work as is required by the last (the leftmost) selector (which is `(!n)`). Now, `nth_prime 12` should produce the same result as the previous version because `generator` will only generate the numbers from 2 to 41. However, this is evidently not the case as the following test reveals:

```
Miranda nth_prime 12
<<not enough heap space -- task abandoned>>
```

The above behaviour happens because one of the selectors *requires all of its input data to be evaluated before it will return a result.*

The offending selector is actually `(foldr sieve [])`. This is somewhat surprising since the following behaves correctly:

```
Miranda ((!12) . foldr (:) [])  [1..]
13
```

Thus, it is not `foldr` which requires the entire input to be evaluated. The culprit is actually the function `sieve`—this is because `sieve` has been defined with pattern matching on its second argument. The consequence of this is that the second argument must be evaluated as far as is necessary to determine whether it is an empty list. However this second argument is the recursive application of `foldr`, which entails another application of `sieve` and therefore triggers an evaluation of the next part of the list, and so on. This behaviour is illustrated in the hand evaluation given below:

```
(!0) (foldr sieve [] [2,3,4])
==> (!0) (sieve 2 (foldr sieve [] [3,4]))
==> (!0) (sieve 2 (sieve 3 (foldr sieve [] [4])))
==> (!0) (sieve 2 (sieve 3 (sieve 4 (foldr sieve [] []))))
==> (!0) (sieve 2 (sieve 3 (sieve 4 [])))
==> (!0) (sieve 2 (sieve 3 ([4])))
==> (!0) (sieve 2 ([3,4]))
==> (!0) ([2,3])
==> 2
```

Compare this to the following:

```
(!0) (foldr (:) [] [1,2,3])
==> (!0) ((:) 1 (foldr (:) [] [2,3]))
==> (!0) (1 : (foldr (:) [] [2,3]))
==> 1
```

In the first hand evaluation, `foldr sieve [] [2,3,4]` cannot produce the initial list item until the entire input list has been scanned. By contrast, the expression `foldr (:)  [] [1,2,3]` can produce the initial item almost immediately because `(:)` does not require pattern matching; lazy evaluation ensures that the calculation for the rest of the list is never done.

The generalized program can be made lazy if the function `sieve` does not use pattern matching (which in this case was entirely unnecessary):

```
nth_prime n
      = selectors generator
        where
         generator = [2..]
         selectors = ((!n) .  (foldr sieve []))
         sieve x any = x :  (filter ((0 =).(mod x)) any)
```

```
Miranda nth_prime 12 [2..]
41
```

5.5 Summary

One of the very first features of Miranda to be covered in this book was the ability to allocate names to expressions. These names are useful because the values of their expressions may be recalled by using the names in subsequent expressions. There are strict rules which govern the binding of names to values and which dictate how these names are made accessible to subsequent expressions. Firstly, every expression has access to those names that exist in its *environment*; secondly, the environment of an expression depends on the textual position of the expression in the program; thirdly, the various environments in the program may be explicitly augmented using **where** definitions and *list comprehensions*.

The main advantage of using these two techniques is that closely related functions and definitions may be self-contained and so not rely on values bound outside the expression (that is, they exhibit closure). They are consequently safer and easier to reuse.

Infinite lists are available to the Miranda programmer as a direct result of the *lazy evaluation* mechanism; the presence of infinite lists, list comprehensions, function composition and higher order functions leads to a style of programming which combines data generators with data selectors. This style is powerful and expressive, but care must be taken to avoid infinite computations and to avoid evaluating too much data.

In particular, it is possible to express the solution to a problem as a combination of a generator of data and a selector which operates on that data (and possibly several subsequent selectors). This encourages an approach to software design whereby a specific problem is solved as the result of a specialization of a general problem, and where the solution is divided into separate components, which are coupled via potentially infinite lists of data, which are evaluated lazily. This approach has the considerable advantage that a selector component can be replaced with a different component in order to solve a different problem. If several selectors are combined using functional composition, this leads to a "pipeline" style of programming, which leads to greater reuse of code and faster code production.