Chapter 9

Programming in the Large

This chapter introduces three mechanisms that extend the principles of modularity and abstraction discussed in previous chapters to facilitate large-scale program development. The first of these mechanisms is the (**%include**) directive, which allows a number of definitions to be grouped together in a script file as a single program block and then to be incorporated into another script file—thereby encouraging the reuse of existing software components. The second mechanism, the (**%export**) directive gives the programmer the option of making visible only those definitions that may be useful to other program blocks, and so hiding those definitions that are used solely within that particular block. It will be seen that this builds on Miranda's **where** facility by allowing an auxiliary function to be bound to many specified functions. Finally the (**%free**) directive enables the generalization of script files into program *templates*. Here, the full definition of certain identifiers is left unbound within the template and only completed when the template is included in another file. This has the major advantage of allowing different specializations of the template to cope with different problems.

9.1 The %include directive

A simple **%include** directive has the format:

%include "filename"

This will make available (that is, put into scope) the definitions that appear in the file *filename*, as long as all the definitions within *filename* are both *correct* and *closed*—that is, there are no unbound identifiers.[1]

[1]By convention, *filename* should have the suffix *.m*; otherwise Miranda will create an equivalent *.m* anyway. Note that, within the UNIX context, files can be included from the current directory, a named pathway or the default Miranda system directory. The reader is referred to Section 27 of the On-line Manual for further details.

This facility means that the various components of a large program can be developed separately (perhaps by different programmers) as separate scripts. These separate scripts can then be linked as and when necessary, rather than having one monolithic program. Furthermore, identifiable "libraries" of related objects (for example, a graphics-handling suite) can be developed and made available to other programmers.

%include usage

A number of points about **%include** usage are worth noting:

1. A script file can contain any number of **%include** directives; they can appear anywhere in the file, but a good policy is to place them at the start of the file.
2. **%include** directives cannot appear within **where** constructs; they are top-level objects, like type declarations.
3. Nested **%include** directives are permitted to a reasonable depth of nesting. In other words, an included file can include other **%include** directives, which themselves may include others and so on—as long as no inclusion circularity exists.
4. The definitions of an included file are restricted in scope to the file that *directly* includes them. They are not inherited by a file that includes the includer. Hence, if *file1* is included by *file2* which is, in turn, included by *file3* then the definitions of *file1* are visible to *file2* but not to *file3*. This has the advantage of allowing the whole of *file1*'s definitions to be local to *file2*, without causing name clashes within *file3*.

9.1.1 Avoiding name clashes

Miranda allows two qualifications to the **%include** directive in order to avoid potential name clashes between the *included* file and the *including* file (or indeed any other included file).

Firstly, an *alias* can be used to rename an included identifier (which can be a type name or a constructor). The general format is:

%include "filename" newname1/oldname1 newname2/oldname2 ...

Thus, the identifier *oldname1* within the file *filename* is renamed *newname1*. If a constructor is aliased and the associated `show` function for its type is included, then the definition of the `show` function will automatically be modified to work with the new alias for the constructor name.

Secondly, it is possible to drop any unwanted definition from the include file, the general format is:

> *%include "filename" -dropped1 -dropped2 ...*

This guarantees that the definitions within *filename* for *dropped1* and *dropped2* etc., are not included within the current script file and hence they can have alternative definitions. Note that, although it is permitted to alias type names and constructors (see above), they cannot be dropped.

Incorporating the Standard Environment

The Standard Environment is included into *every* script file, and it is currently *not* possible to include it explicitly, nor to rename or drop any of its components.

9.2 The %export directive

By default, a **%include** directive will incorporate *all* the top-level contents of the included file. The **%export** directive can be used in the included file to modify this default to specify which components are made visible to the including file.

A script file may have just one **%export** directive, which may take a number of optional qualifiers:

1. To export all definitions.
2. To export selected definitions.
3. To export included files.
4. To not export specified objects.

Exporting all definitions

For any script file, the default is that all of its top-level definitions will be visible to an including file. This can be made explicit by the following directive:

```
%export +
```

Exporting selected definitions

In order to restrict the scope of auxiliary definitions (that are only relevant within the exporting file), if the **%export** directive is followed by an export list of identifiers (which may be type names and constructor names) then only these will be visible within the including file.

The general format for the **%export** directive is:

%export function1 function2 type1 ...

All other definitions within the exporting file become local to that file. This facility has the advantage over the **where** construct, in that a function or identifier can be made local to *several* other functions rather than just one.

Exporting included files

In order to override the safeguard that definitions from a **%include** file are only visible within the file that has included them, it is possible to export that included file. The following extract shows the advantages of this facility to combine existing modules into a new module:

```
%include "graphicslib"
%include "matrixlib"
%export "graphicslib" "matrixlib"
|| etc
```

Not exporting specific objects

Sometimes it may be more elegant to exclude certain functions from being exported, rather than provide an export list. The following program extract first directs that all the definitions within the current file and *otherfile* be exported, and then directs that `f1` and `f2` should *not* be exported. Note that the excluded identifiers could be from the current file and/or *otherfile*.

```
%export + "otherfile" -f1 -f2
```

9.2.1 Exporting types

The following subsection, provides a brief outline of the safeguards that Miranda provides for exporting types; the reader is referred to Section 27 of the On-line Manual for further details.

Type abstraction

Exporting an algebraic type name will *automatically* export all of its constructors. There is no way to override this mechanism; it is necessary to use the **abstype** mechanism (described in Chapter 7) in the exporting file to achieve the equivalent of data hiding.

Type orphans

It is illegal to export an identifier to a place where its type, or any part of its type, is unknown; this prevents "type orphans". Thus, if a script file *tree.m* provides definitions for an algebraic type *tree ** and a function *flatten_tree :: tree * -> [char]* then *tree * must* be exported if *flatten_tree* is exported.

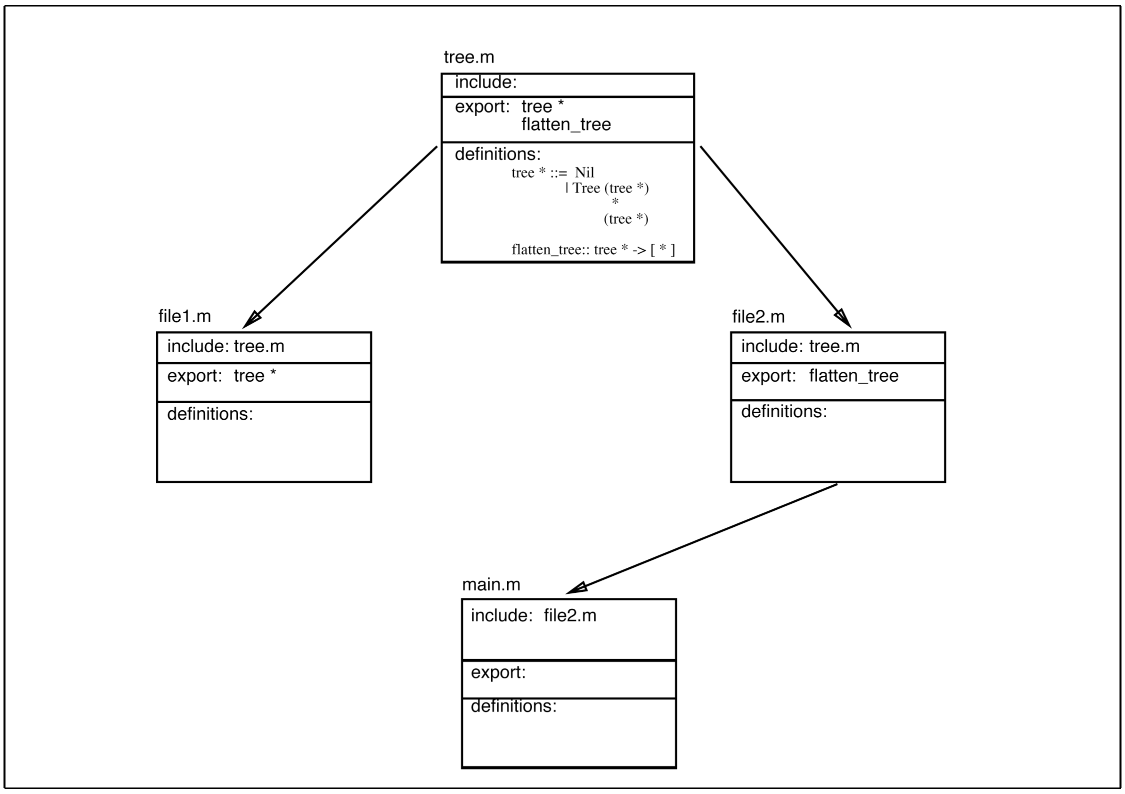

Figure 9.1 An illegal inclusion produces a type orphan.

If a function is exported without its type then the unbound type expression is known as a "type orphan". Figure 9.1 illustrates an illegal inclusion which causes Miranda to issue the following error message:

```
Miranda /f main
compiling main.m
checking types in main.m
MISSING TYPENAME
the following type is needed but has no name in this scope:
'tree' of file "tree.m", needed by: flatten;
typecheck cannot proceed - compilation abandoned
Miranda
```

Re-adoption of type orphans

If there are several levels of inclusion (recall that **%include** works transitively), then it would be possible for the above type *tree ** to be included into the main script from a different file to that which exported *flatten_tree*. Thus, the main script can "re-adopt" type orphans that would otherwise be without a parent from the most immediately included file. However, the file which exported the function *flatten_tree* must have had the type definition for *tree ** in scope: *and both the main script file and the file which exported the function must derive the type definition from the same source file.* Miranda always recognizes when the same file has been included, however indirectly. Figure 9.2 illustrates a re-adoption of the type *tree.m*, thus solving the type orphan problem seen in Figure 9.1.

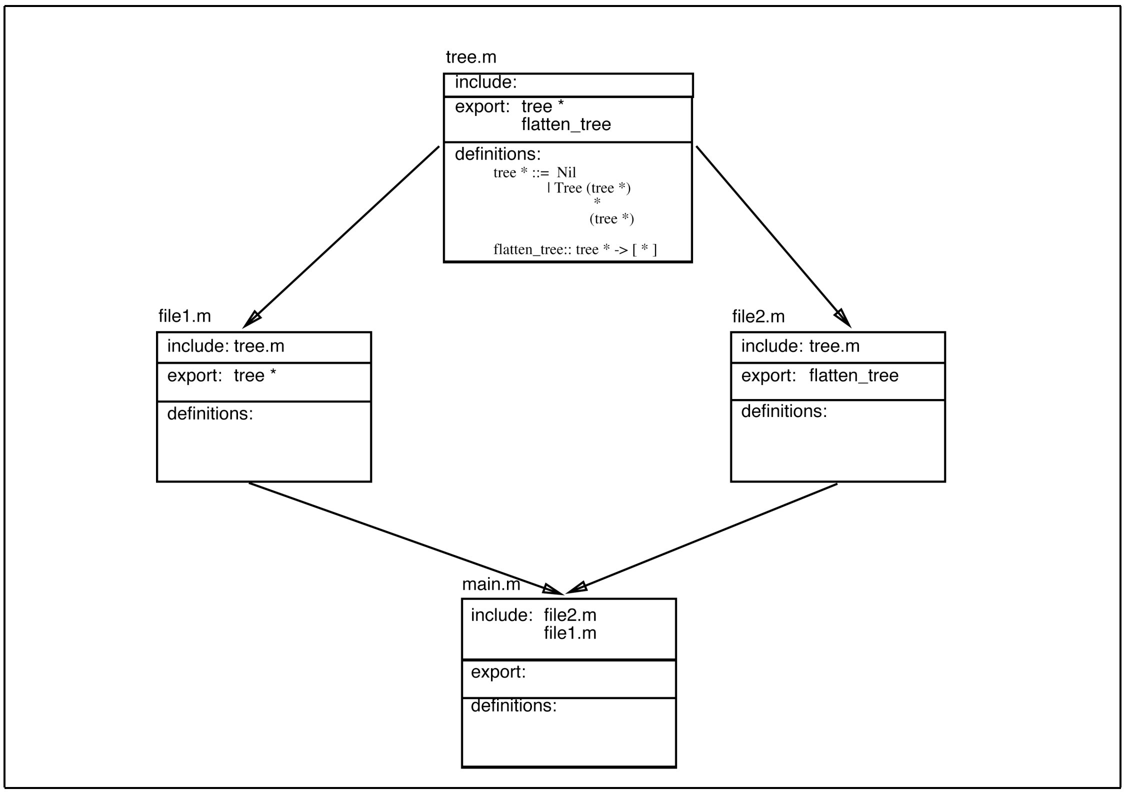

Figure 9.2 Re-adoption of a type orphan.

Type clashes

Two types with identical textual definitions but different source files will always be considered as different types and so give rise to a type clash. Furthermore, it is illegal for the same original type to be included twice into the same scope, even under different aliases.

9.2.2 Example: text justification

The example in the rest of this section discusses the design of a formatting module *justify*, to perform text justification. This module may be exported to become part of a text editor, and itself will include a number of tried and tested general-purpose functions that have already been collected into a library file.

Specification

The `justify` program will take a string of characters and a desired line width and transform it into a left and right justified text with lines of the desired line width. The last line will not be right justified. It is assumed that the selected line width is long enough to hold the longest word in the input string.

Target signature requirements

The first stage in the design of `justify` is to choose which objects will be shown to the outside world; this is simply the `justify` function itself; no other functions should be in scope outside of `justify`'s script file. This is achieved by the following directive:

```
%export justify
```

Now there will be no accidental name clashes or confusion with the other top-level functions within the including file.

Program design overview

The program follows a divide and conquer principle. The input text is considered as an unformatted paragraph with possible multiple spaces, blank lines (two consecutive newline characters) and lines of different width from the desired line width. The final output will have a fixed line width and regular spacing between each word.

The first step is to compress the "white space" characters (this has already been discussed in Section 5.2). Subsequently the text is split into lines, which are then split into words, which are output with at least one space between them.

Extra spaces must be added if a word straddles a line divide. In this situation, the word is shunted to the next line and the current line must be padded out with extra spaces to fill the gap. The program will evenly distribute extra spaces from the left; this is perhaps simplistic but the design makes a more elegant distribution quite easy to implement later.

Implementation

The script file (*justify.m*) for the justify program is now presented—it makes use of several functions to manipulate lists which are assumed to be contained within a general-purpose list manipulation library *listtools.m*. To avoid potential name clashes, the definitions within the *justify.m* file will be explicitly dropped from *listtools.m*.

The file *listtools.m:* contains functions which have not been given elsewhere in this text; these functions are now presented, followed by the main script file *justify.m*:

```
>|| Literate Script:  part of listtools.m

Contains:  a library of list manipulation functions



> isspace c = c = ' '

> occurs []     x = False
> occurs (front :  rest) front = True
> occurs (front :  rest) x = occurs rest x

> replicate n x = take n (repeat x)

note:  replicate is the equivalent
      of the built-in function ''rep''

> takeafter f []     = []
> takeafter f (front :  rest)
>         = rest, if (f front)
>         = takeafter f rest, otherwise

    etc
```

```
>|| justify.m (Page 1 of 2)

> %export justify
> %include "listtools.m"
>         -justify -compress -justifytext
>         -justifyline -splittext -splitline
>
> string == [char]
>
> justify ::  num -> string -> string
> justify width text = justifytext width (compress text)

Warning:  it is assumed that each word in the text
is not greater than the specified width

Reduce multiple white space to a single space
and delete all leading and trailing white space

> compress ::  string -> string
> compress line
>  = (notrailing .  xcompress .  noleading) line
>    where
>     xcompress [] = []
>     xcompress (front :  rest)
>       = ' ' :  (xcompress (dropwhile isspace rest)),
>         if isspace front
>       = front :  (xcompress rest), otherwise
>     notrailing = reverse .  (dropwhile isspace) .  reverse
>     noleading = dropwhile isspace

Split text and justify a line at a time, it passes
the number of gaps between words and the number of
leftover spaces

> justifytext ::  num -> string -> string
> justifytext width text
>  = text ++ "\n", if (# text) <= width
>            || last line in paragraph
>  = (justifyline line gaps leftover)
>    ++ (justifytext width restoftext), otherwise
>    where
>     (line, leftover, restoftext) = splittext width text
>     gaps = occurs line ' '
```

```
justify.m continued (Page 2)

No extra padding needed if no more words or extra spaces
at least 1 space between words, otherwise

> justifyline ::  string -> num -> num -> string
> justifyline line gaps 0 = line ++ "\n"
> justifyline line 0 leftover = line ++ "\n"
> justifyline line gaps leftover
>   = word ++ (replicate (extraspaces + 1) ' ') ++
>     justifyline rest (gaps - 1) (leftover - extraspaces)
>     where
>      (word, extraspaces, rest)
>        = splitline line gaps leftover

Split the text and calculate the number of left over spaces
necessary to pad out line

> splittext ::  num -> string -> (string,num,string)
> splittext width text
>   = (line, leftover, restoftext)
>     where
>      revline = reverse (take (width + 1) text)
>      leftover = # (takewhile ((~) .  isspace) revline)
>      line = take (width - leftover) text
>      restoftext = drop (width - leftover + 1) text

Split line and calculate extra spaces between words

> splitline ::  string -> num -> num -> (string,num,string)
> splitline line gaps leftover
>   = (word, extraspaces, restofline)
>     where
>      word = takewhile ((~) .  isspace) line
>      restofline = takeafter isspace line
>      extraspaces = leftover div gaps,
>                     if leftover mod gaps = 0
>                   = (leftover div gaps) + 1, otherwise
```

9.2.3 Constraining include files

In the above program, *justify.m* dropped definitions from the included *listtools.m* file in order to avoid potential name clashes. An alternative approach is to set up an intermediate "header file" *header.m* which includes all of *listtools.m* and only exports those definitions needed by *justify.m*. The entire contents of *header.m* would be:

```
%include "listtools.m"
%export takeafter isspace replicate occurs
```

The directives at the top of *justify.m* would now read:

```
%include "header.m"
%export justify
```

Although this approach is slightly more complex, it has two major benefits:

1. The programmer only needs to know as much about the contents of the original included file as is required for their own program.
2. If the original included file is later modified to incorporate further definitions then there is no danger of any name clashes between these additions and the current program.

9.3 The %free directive

A Miranda script file may contain one **%free** directive to be used in conjunction with an **%export** directive (and an extended **%include** directive), which gives the programmer the opportunity to write a "template" containing incomplete definitions to be completed by an including file. This can be thought of as parameterizing the exporting file.

The general format used within the exporting file is:

%free signature

where *signature* is a sequence of specifications for identifiers which will be fully defined within an including file. In other words, these identifiers are "free" or unbound within the exporting file. For example:[2]

[2]Notice that the syntax of the free type declarations is the same as that of placeholder types, and indeed the two concepts are similar, in that they allow the program designer to defer implementation decisions. For placeholder types, Miranda expects just one final type definition: for free types, Miranda will accept different files with different definitions as long as they meet the free type template.

```
|| File:  queue.m

%free {
      queue * ::  type
      qmax ::  num
      }
       || etc
```

The general format for the including file is:

%include "filename" bindings

where *bindings* is a semicolon-delimited sequence of definitions for the free identifiers in the included file. It may contain definitions for the free types using the type synonym mechanism (==), and definitions for other identifiers, using the token =.

Hence, the above *queue.m* file, could have a corresponding including file with directives such as:

```
|| File:  receiver.m

%include "queue.m" {queue * == [num]; qmax = thismax;}

     || etc

thismax = 100
```

Here the free objects in *queue.m* are bound by the *receiver.m* file. In this case, the free type `queue` obtains a binding as a number list, and the identifier `qmax` is bound to `thismax`.

The important advantage of a script which has been parameterized by a **%free** directive is that different bindings may be given for its free identifiers on different occasions. Thus a program, using the file *another_receiver*, might need `queue` to be a polytype and the value of `qmax` within *queue.m* to be the same as the value of `qmax` within *another_receiver*:

```
|| File:  another_receiver.m

%include "queue.m" {queue * == [*]; qmax = qmax;}

     || etc

qmax = 9000
```

The benefits of this approach are more fully apparent in Section 9.4, which presents an expanded version of the *grep* program, modularized into five files. The use of the **%free** directive and its associated **%include** directive will be seen in the two files *grep.m* and *main.m*. The program is then modified to deal with different meta-characters but, because of the parameterization, it is only necessary to modify the main file and the file dealing with these different meta-characters.

Rules for %free directives

The following rules hold when using **%free** directives:

1. All free types and identifiers must be exported; either explicitly in a **%export** directive or as the default behaviour of any included file.
2. The identifiers declared within a **%free** directive may denote types as well as values. When the file is included by another, bindings must be provided for *all* free identifiers. The bindings are given in braces following the pathname in the **%include** directive (before the aliases, if any), and each binding must be terminated by a semicolon.
3. The bindings for a parameterized script's free identifiers must be *explicitly* stated, even if the new name being bound is the same as the name formally defined to be "free" in the included file, as shown above with `qmax`. Another example of this explicit binding is given in Section 9.4.5, where the free type `regexplist` in the included file is bound to the type `regexplist`, which is in scope in the including file.
4. When a parameterized script exports a locally created type (other than a synonym type), each instantiation of the script by a **%include** is deemed to create a *new* type. This is relevant when deciding whether two types are the same for the purpose of re-adopting a type orphan.

9.4 Reusable software—grep revisited

The *grep* utility presented throughout the text is now extended as a comprehensive example of program construction using template files. The `sublist` and `lex` activities are separated and the primary `grep` function is parameterized on them. In addition to giving the program greater modularity, it has the advantage of making the `sublist` code *reusable.* It is now possible to change the "lexical analyser" so that it will recognize different representations of the meta-characters (for example, to deal with the UNIX Bourne Shell file-generation codes) without changing the `sublist` code.

The rest of this section continues the design shown in Chapter 3 to incorporate the other *grep* meta-characters. The inclusion of Range meta-characters leads to a slight revision in the way that regular expressions are represented and how any particular regular expression element is compared with its corresponding searched line character. However, the original search *strategy* remains essentially unaltered.

9.4.1 Program structure

The overall structure of this *grep* program is that of five interconnected files, as illustrated in Figure 9.3. These files communicate with each other as follows:

1. The file *types.m* exports definitions for common types and exports definitions for the functions `equal` and `notequal`.
2. The file *sublist.m* exports a definition for the function `sublist` but keeps the function `startswith` private.
3. The file *stdlex.m* exports a definition for the function `lex` but keeps many other functions private.
4. The file *grep.m* exports the function `xgrep`, which relies on the definitions of `sublist` and `lex`.
5. The file *main.m* is used to combine all the other files, to create the function `grep`.

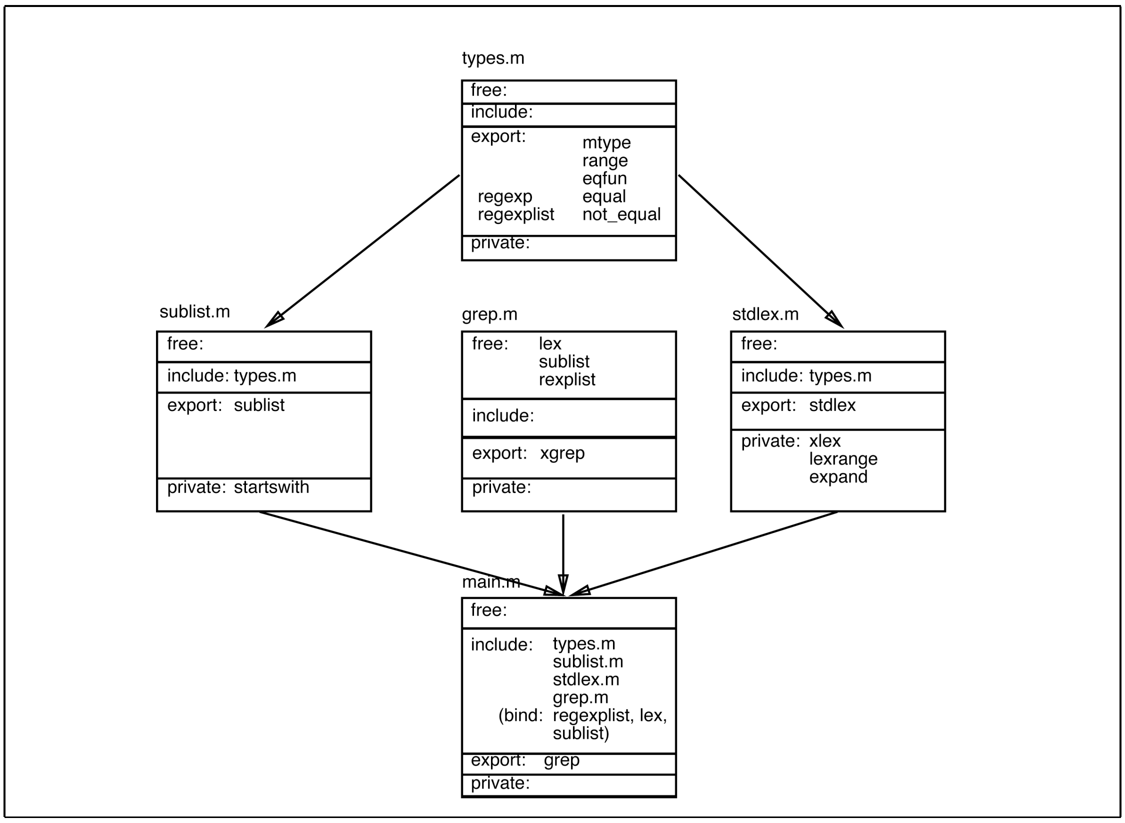

Figure 9.3 A large program divided into five files.

9.4.2 Incorporating "End of Line" and "Start of Line"

As shown in Section 3.9.2, the regular expression can be expressed as a list of pairs; the first component representing the type of match (zero or more, or one only) and the second component the actual value to be matched. This principle can be extended to cater for the meta-characters to anchor a search to the start of a line (`^`) or to the end of a line (**$**). To anchor a match from the start of the line means that the `sublist` function is only applied once (that is, it does not recurse). Matching the end of the line can only succeed if the searched line is empty when the end of line meta-character is encountered (as the last element) in the regular expression list. These considerations give rise to an extended `mtype` algebraic type, with `SOL` for start of line and `EOL` for end of line:

```
mtype ::= ZERO_MORE | ONCE | SOL | EOL
```

The existing *grep* program must be changed in three places:

1. `lex`: to recognize the new meta-characters.
2. `sublist`: to anchor the regular expression match to the start of the search line.
3. `startswith`: to succeed in a match, if the search line is empty when EOL is met in the regular expression.

The actual code is presented in Section 9.4.5, after discussion of the other meta-characters.

9.4.3 Incorporating "Any" and "Range"

The final stage of the initial design is to incorporate the meta-characters for *any* and a *range* of characters. An initial case analysis shows all the possible combinations:[3]

Number of occurrences	Object affected
One Only	a given single character
Zero or More	a given single character
One Only	any single character
Zero or More	any single character
One Only	any single character from the range [...]
Zero or More	any single character from the range [...]
One Only	any single character not in the range [...]
Zero or More	any single character not in the range [...]

[3]Notice that "Zero or More of any single character in a range" does not constrain the pattern to be a number of occurrences of the *same* character from the range; rather, for each new occurrence, a different character may be chosen from the range.

Such an inspection reveals that the matching requirements of the new meta-characters are in fact quite similar to that of matching a single character.

Ranges

One method of treating a ranged regular expression element is to expand it into a list of valid characters, then check whether the current line position matches any of the characters in the list by using the function `member` (introduced in Chapter 3). For example the range `"[b-e]"` can be expanded into the range list `"bcde"` and `"[a-d_1-3]"` can be expanded to `"abcd_123"`.

With this approach, a single actual value has the same representation as a single item range, for example `"a"` and `"[a]"` will both be represented as `"a"`.

Negative ranges

Negative ranged regular expression elements such as `"[^a-d]"` (which imply that a single character should be chosen which may be any character *except* those in the range) can be treated in one of two ways:

1. A range list could be generated with all the values that are not in [a-d], that is `['e','f',...]`, and `member` can be used as a test for equality.
2. The range list [a-d] could be expanded to `"abcd"` and the truth value returned by `member` subsequently inverted.

Both options are equally valid. The first option probably requires extra initial work to construct the range list. The latter requires either additional match types `NOT_ONCE` and `NOT_ZERO_MORE` or an additional component to the regular expression tuple that indicates what to do with the result of `member`. The design followed here is the second option (using an additional tuple element) for reasons now discussed.

Any

On inspection it can be seen that the `"."` meta-character is really a convenient shorthand for the range `"[\0 - \127]"`, which could be expanded accordingly. An alternative approach is to consider it in terms of *"not matching nothing"*, that is given `member` returns `False` for the empty list ([]) then `"."` really corresponds to the test `((~) . (member []))`.

Type requirements

The latter method using `member` implies that a consistent treatment of all the wild cards and actual values can be achieved by representing a regular expression element as the triple `(mtype, eqfun, range)` with the predefinitions:

```
mtype ::= ZERO_MORE | ONCE | SOL | EOL
range == [char]
eqfun == range -> char -> bool
```

The equality functional type `eqfun` will either be `equal` or `notequal`, based on a membership function that maps a `range` and a `char` to a Boolean value. If `range` is a `[char]` then the built-in `member` function will suffice:

```
equal::eqfun
equal = member

notequal ::  eqfun
notequal r = ((~) .  (member r))
```

9.4.4 The program libraries

Figure 9.3 illustrates the division of the *grep* program into separate files. One of these files (*main.m*) combines the other files to form the program; the others are called "library" files.

The structure presented in Figure 9.3 demonstrates that the overall program need only export the `grep` function; this function will take a string (of type `[char]`) representing the raw regular expression and another string to be searched and returns a list of strings (of type `[[char]]`) that have been matched. Hence the file *main.m* need only export the following:

```
%export grep
```

The file *grep.m* holds the definitions of `xgrep` and `xgrep_pred`; only the former is exported to *main.m*. The code in *grep.m* requires definitions for `lex` and `sublist` and for the type `regexp`; these are defined in other files and must therefore be declared *free* in *grep.m*. Bindings for these names are given in *main.m* as part of the **%include** directive for *grep.m*.

The file *sublist.m* exports the function `sublist` and must include the file *types.m*, so that it may know details of the `regexplist` type, including:

1. The underlying `mtype` enumeration to enable meta-character pattern matching.
2. The `eqfun` datatype to deconstruct the embedded membership function.

The file *stdlex.m* similarly must include *types.m* and exports the function `stdlex`. The implementation of each of the five files is presented in the following section.

9.4.5 Implementation of the grep program

The *grep* program consists of four library files and one main file which combines the exported definitions from the library files and provides appropriate bindings. The implementation of these files is now presented.

The file types.m

```
>|| types.m:  contains type definitions for grep

> %export regexp regexplist mtype range eqfun equal notequal

A regular expression is held as a list of regexp triples
with an enumeration type for meta-character pattern matching
and range types a list of characters

> regexp == (mtype, eqfun, range)
> regexplist == [regexp]

> mtype ::= ZERO_MORE | ONCE | SOL | EOL

> range == [char]

Equality functions are derived from
the built-in member function:

> eqfun ::= Eqfun (range -> char -> bool)

> equal ::  eqfun
> equal = Eqfun member

> notequal ::  eqfun
> notequal = Eqfun f
>            where
>            f r = ((~) .  (member r))
```

The file sublist.m

The new versions of `startswith` and `sublist` are both held in the file *sublist.m*. The actual code is remarkably similar to the previous version (shown in Chapter 3) because the basic search strategy has not been altered:

```
>|| sublist.m

Contains:  definitions for sublist and startswith

> %include "types"
> %export sublist

Sublist determines whether a regular expression occurs
within a given text line.  It uses startswith.

> sublist ::  regexplist -> [char] -> bool
> sublist ((SOL, x, y) :  rest) line
>     = (startswith rest line)
> sublist expr line
>    = (startswith expr line) \/ xsublist line
>       where
>        xsublist [] = False
>        xsublist ( x :  lrest)
>           = (startswith expr lrest) \/
>             (xsublist lrest)

Startswith determines whether a regular expression
occurs at the beginning of a given line of text:

> startswith ::  regexplist -> [char] -> bool
> startswith [] line = True
> startswith ((ZERO_MORE, x , y) :  regrest) []
>    = startswith regrest []
> startswith [(EOL, x , y)] line = (line = [])
> startswith rexp [] = False
> startswith ((ZERO_MORE,
>            (Eqfun ismatch), chrange) :  regrest)
>            (lfront :  lrest)
>    = startswith regrest (lfront :  lrest) \/
>      (ismatch chrange lfront &
>      startswith ((ZERO_MORE,
>                  (Eqfun ismatch), chrange) :  regrest)
>                   lrest
>      )
> startswith ((x, (Eqfun ismatch), chrange) :  regrest)
>            (lfront :  lrest)
>    = ismatch chrange lfront &
>      startswith regrest lrest
```

The file grep.m

```
>|| grep.m

Contains:  definition for xgrep

> %free { regexplist ::  type
>        sublist ::  regexplist -> [char] -> bool
>        lex ::  [char] -> regexplist
>        }

> %export xgrep

The grep function returns those lines from text
which contain a match for the given regular expression:

> xgrep ::  [char] -> [[char]] -> [[char]]
> xgrep x text = filter (sublist (lex x)) text
```

The file stdlex.m

The code for the extended lexical analyser is a straightforward matter of listing which patterns have special meanings for *grep* and converting them to the appropriate format for the matching algorithms to manipulate. In problems of this nature, the technique of case analysis (discussed in Chapter 3) is of particular importance:

```
>|| stdlex.m (Page 1 of 3)

Contains:  definitions for stdlex, xlex, lexrange & expand

> %include "types.m"
> %export stdlex

This is the "standard" lexical analyser for grep.
The top-level function stdlex first checks
for the start-of-line pattern and then applies xlex:

> stdlex ::  [char] -> regexplist
> stdlex ('^' :  rest) = (SOL, equal, []) :  xlex rest
> stdlex p = xlex p
```

```
stdlex.m continued (Page 2 of 3)

xlex does most of the conversion from the meta-patterns
to their underlying representation based on
(mtype, eqfun, range) triples:

> xlex ::  [char] -> regexplist

> xlex []
>   = []
> xlex ('\\' :  ch :  '*' :  rest)
>   = (ZERO_MORE, equal, [ch]) :  xlex rest
> xlex ('\\' :  ch :  rest)
>  = (ONCE, equal, [ch]) :  xlex rest
> xlex ('\\' :  [])
>   = [(ONCE, equal, ['\\'])]
> xlex ('.'  :  '*' :  rest)
>   = (ZERO_MORE, notequal, []) :  xlex rest
>
> xlex ('[' :  '^' :  rest)
>   = rangepart :  (xlex exprest)
>     where
>      (rangepart, exprest)
>        = lexrange (notequal, []) rest
>
> xlex ('[' :  rest)
>   = rangepart :  (xlex exprest)
>     where
>      (rangepart, exprest)
>        = lexrange (equal, []) rest
>
> xlex ('.'  :  rest)
>   = (ONCE, notequal, []) :  xlex rest
> xlex (ch :  '*' :  rest)
>   = (ZERO_MORE, equal, [ch]) :  xlex rest
> xlex ('$' :  [])
>   = [(EOL, equal, [])]
> xlex (ch :  rest)
>   = (ONCE, equal, [ch]) :  xlex rest
stdlex.m continued (Page 3)

lexrange is called by xlex when a range pattern
(in square brackets) must be deciphered.
It evaluates to the underlying representation
for the deciphered range, plus the rest of the input
which is then scanned in a recursive application of xlex:

> lexrange ::  (eqfun,range) -> [char] -> (regexp,[char])
>
> lexrange (x,y) []
>   = error "lexrange error"
> lexrange (mfunc, chrange) ('\\' :  ch :  rest)
>   = lexrange (mfunc, ch :  chrange) rest
> lexrange (x, []) (']' :  rest)
>   = error "empty range"
> lexrange (mfunc, chrange) (']' :  '*' :  rest)
>   = ((ZERO_MORE, mfunc, chrange), rest)
> lexrange (mfunc, chrange) (']' :  rest)
>   = ((ONCE, mfunc, chrange), rest)
>
> lexrange (mfunc, chrange) (start :  '-' :  stop :  rest)
>   = error "bad range",
>       if (start > stop)
>   = lexrange (mfunc,(expand chrange start stop)) rest,
>       otherwise
>
> lexrange (mfunc, chrange) (ch :  rest)
>   = lexrange (mfunc, ch :  chrange) rest


expand is used by lexrange in order to create
all the characters in a range:

>expand ::  range -> char -> char -> [char]
>
>expand chrange start stop
>   = chrange ++ (map decode [(code start)..(code stop)])
```

The file main.m

A usable instance of *grep* program can now be created by including both the *sublist.m* file and the *stdlex.m* file into an application file called *main.m*:

```
>|| main.m

Contains:  definition for grep

> %include "types"
> %include "sublist"
> %include "stdlex"
> %include "grep" {regexplist == regexplist;
>                  lex = stdlex; sublist = sublist;}

> %export grep

> grep ::  [char] -> [[char]] -> [[char]]
> grep = xgrep
```

Notice that the above binding `regexplist == regexplist` is obligatory. It is not redundant, nor is it cyclic; the `regexplist` on the left of the `==` refers to the current file and that on the right to the definition within the file that has been included.

The following session illustrates how the grep function might be used (the output line has been broken for clarity):

```
Miranda grep "[a-d]c*h"  (lines (read "/etc/passwd"))
["bsmith::1033:30::/usr/users/bsmith:/bin/csh"]
```

9.4.6 Using a different lexical analyser

The UNIX Bourne Shell[4] has a number of meta-characters for file name expansion that are similar but not identical to the *grep* meta-characters. The following table can be compared to Table 3.9 for their differences. For example, it can be seen that the wild card for a single character is different and there is no point in looking for a file name anchored at the start of a line. Otherwise the `ZERO_MORE`, `ONCE` and `RANGE` requirements are semantically the same, and so the `sublist` and `startswith` functions can work equally well for a program that meets the Bourne Shell parsing requirements as for the *grep* requirements.

[4]This is a job control language which provides an interface between the user and the UNIX Operating System.

Table 9.1 Options for the UNIX *Bourne shell*

Character	Meaning
c	any non-special character *c* matches itself
\c	turn off any special meaning of character *c*
?	any single character
[...]	any one of characters in ...
	(e.g. 1-9 covers all ASCII values between 1 and 9)
[!...]	any one of characters not in ...
	matches any string (including the empty string)
New line	is not matched by anything

The above meta-characters can be emulated simply by writing another lexical analyser (called *bournelex.m*) and changing the **%include** directive and the bindings in *main.m*.

The new file bournelex.m

```
>|| bournelex.m:  (Page 1 of 2)

Contains:  bournelex, xlex, lexrange and expand

> %include "types.m"
> %export bournelex

This is the "Bourne" lexical analyser for grep.
It uses a meta-level syntax that is similar to
the filename generation syntax of the UNIX Bourne shell.
The top-level function bournelex just applies xlex

> bournelex ::  [char] -> regexplist
> bournelex p = xlex p

The functions xlex and lexrange behave similarly
to their grep equivalents, expand is identical
```

```
bournelex.m (Page 2)

> xlex :: [char] -> regexplist
> xlex [] = []
> xlex ('\\' : ch : rest)
>   = (ONCE, equal, [ch]) : xlex rest
> xlex ('\\' : [])
>   = [(ONCE, equal, ['\\'])]
> xlex ('*' : rest)
>   = (ZERO_MORE, notequal, []) : xlex rest
> xlex ('[' : '!' : rest)
>   = rangepart : (xlex exprest)
>     where
>      (rangepart, exprest) = lexrange (notequal, []) rest
> xlex ('[' : rest)
>   = rangepart : (xlex exprest)
>     where
>      (rangepart, exprest) = lexrange (equal, []) rest
> xlex ('?' : rest)
>   = (ONCE, notequal, []) : xlex rest
> xlex (ch : rest)
>   = (ONCE, equal, [ch]) : xlex rest

> lexrange :: (eqfun,range) -> [char] -> (regexp,[char])
> lexrange (x,y) [] = error "lexrange error"
> lexrange (mfunc, chrange) ('\\' : ch : rest)
>     = lexrange (mfunc, ch : chrange) rest
> lexrange (x, []) (']' : rest)
>     = error "empty range"
> lexrange (mfunc, chrange) (']' : rest)
>     = ((ONCE, mfunc, chrange), rest)
> lexrange (mfunc, chrange) (start : '-' : stop : rest)
>     = error "bad range", if (start > stop)
>     = lexrange (mfunc, (expand chrange start stop))
>                rest, otherwise
> lexrange (mfunc, chrange) (ch : rest)
>     = lexrange (mfunc, ch : chrange) rest

> expand :: range -> char -> char -> [char]
> expand chrange start stop
>  = chrange ++ (map decode [(code start)..(code stop)])
```

The new file main.m

```
>|| main.m:  Contains definition for grep

> %include "types"
> %include "sublist"
> %include "bournelex"
> %include "grep" {regexplist == regexplist;
>                   lex = bournelex; sublist = sublist;}

> %export grep

> grep ::  [char] -> [[char]] -> [[char]]
> grep = xgrep
```

The new *grep* program now interprets the meta-level patterns in a different way:

```
Miranda grep "[a-d]c*h"  (lines (read "/etc/passwd"))
["adm:*:5:3:SGI Accounting Files Owner:/usr/adm:/bin/sh",
 "don:1h/87cnH8JbxH:16:10::/usr/users/don:/bin/csh",
 "SGIguest::998:998:SGI Guest account:/usr/people/guest:/bin/csh",
 "clack:3H/BFyMrLgGxM:1021:500::/usr/users/clack:/bin/csh",
 "macstuff:2J%Gh9xHnvBMh:1025:500::/usr/users/macstuff:/bin/csh"]
```

9.5 Summary

When "programming in the large", it is normally desirable to break the problem into manageable sub-problems, which themselves may be split further. This division should be reflected by encapsulating each sub-solution into its own script file. The communication between the files can be co-ordinated, using the features introduced into this chapter:

1. **%include**—to control the interface between files.
2. **%export**—to control the visibility of identifiers.
3. **%free**—to provide program templates.

By adopting this approach, large programs will be easier to code, test and modify, as shown in the development of the *justify* and *grep* programs. Furthermore, libraries of general purpose software components can be developed separately and then confidently reused as part of many other programs. This was demonstrated in the reuse of components from *grep* program to develop a Bourne shell filename expansion program.