



A declarative development technique

R. Harrison

Department of Electronics and Computer Science, University of Southampton, Hampshire, UK

ABSTRACT

We present a way in which structured analysis and structured design can be used together with functional languages to produce a complete declarative development technique. We also examine the use of prototyping in the software development process by considering the use of functional languages for evolutionary prototyping. Our main conclusion is that the combination of structured analysis and structured design, evolutionary prototyping and functional languages produces a powerful methodology which facilitates the development of quality software.

Keywords: rapid prototyping, specification, functional languages

INTRODUCTION

In this paper we demonstrate how structured analysis and structured design [1, 2] can be used together with functional languages to produce a complete declarative development technique. We hypothesize that the methodologies of structured analysis and structured design will complement the use of functional languages very well, due to the emphasis that such methodologies place on the flow of data, rather than flow of control.

Prototyping has frequently been claimed to facilitate the software development process; for example, see Tanik and Yeh [3], Boehm *et al.* [4], and Boehm [5]. We examine this claim by considering the use of functional languages to produce evolutionary prototypes [6]. Functional languages facilitate rapid prototyping by providing powerful tools such as pattern matching, higher-order functions and lazy evaluation [7], which allow and encourage the expression of high level, abstract concepts. We argue that a prototype written in a referentially transparent programming language can be a valuable part of the software development process, as such a prototype can be used both to validate the design (by executing test cases) and to prove properties of the design.

We start by describing a declarative development technique, and then go on to exemplify this technique with a case study. We show how we can prove

432 Software Quality Management

properties of the prototype, and consider the value of design time testing. We use the guidelines of structured analysis and structured design to estimate the code quality. The paper ends with a comparison of related work.

DESCRIPTION OF THE DEVELOPMENT TECHNIQUE

The development technique exemplified here is based on the structured analysis and structured design methods of De Marco [1] and Page Jones [2]. The methods produce graphical specifications, and have been acclaimed for their simplicity and ease of use.

The elements of structured analysis which we will utilize are data flow diagrams and the associated data dictionary. From the structured design process we adopt the use of pseudocode and structure charts. One of the advantages of structured analysis is that it results in a graphical specification which is easy to assimilate and comprehend. No mathematical skills are required either to produce the specification or to understand it: consequently such a specification can be understood by both professionals and naïve users.

The design proceeds in a top-down manner: each process is refined until a point is reached at which the implementation is straightforward. The design technique is entirely logical; that is to say, no implementation dependent features are introduced, which should result in a reusable, portable implementation. The structured analysis and structured design methodology results in an implementation which is partitioned into a number of independent functions, or, for large systems, a number of independent modules. The resulting collection of data flow diagrams, data dictionary and pseudocode constitute a detailed specification which can then be implemented. The final part of the design documentation, the structure charts, can be reverse engineered after completing and testing the implementation. Such an approach minimizes the number of changes to the documentation and also ensures its correctness.

We will use a pure functional language to implement the specification. Such languages are referentially transparent [8] and consequently functions written in them are amenable to proof and transformation techniques which can be conducted in the usual mathematical manner, as shown in Darlington [9].

The resulting prototype can be used to test the functionality of the solution. The ease with which this is achieved in the example given here emphasises the applicability of functional languages to rapid prototyping in general and evolutionary prototyping in particular. Once a program has been implemented and tested in a functional language it can, if necessary, then be implemented in an imperative language such as Modula-2 [10] or C++ [11]. This will necessitate implementing the abstract data types used in the prototype.

We will adhere to the software life cycle shown in the diagram in figure 1.

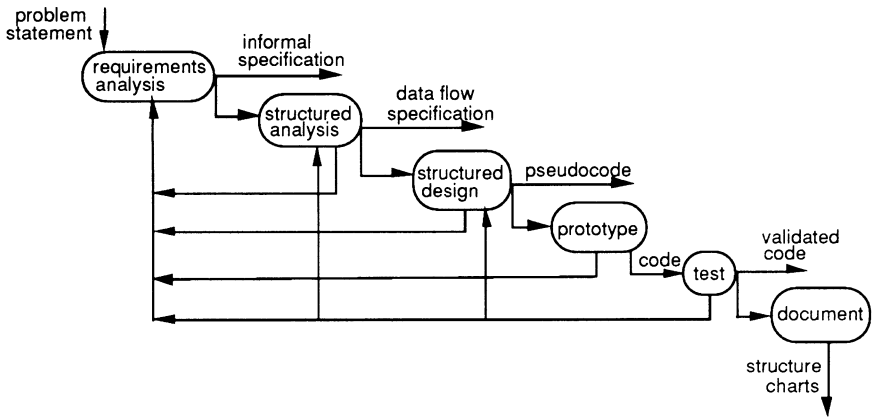


Figure 1

The following section demonstrates the stages of requirements analysis, structured analysis, structured design, prototyping and testing, using a case study.

THE CASE STUDY

We will illustrate the methodology shown in figure 1 by using a problem taken from text processing, starting with a problem statement:

Write a program which takes a text file and a phrase and returns the sentence or sentences of the text file in which the phrase occurs.

Requirements analysis

By analysing the customer's requirements, we can produce the following informal specification:

Write a program which takes a text file and a phrase and returns the sentence or sentences of the text file in which the phrase occurs. A text file is assumed to consist of sentences separated by full stops. Each sentence consists of words separated by one or more space characters (where a space character is a blank, " ", newline, "\n", or tab, "\t") and possibly some punctuation characters, and each word is a list of alphabetical characters. The punctuation characters, which are to be ignored, include the comma, colon, semi-colon, round parentheses, apostrophe and quotation mark. Tabs and newlines will also be ignored.

Structured analysis

We will use a data flow analysis technique, as it gives a clear view of the operation of the process under consideration. We start with the top level data flow diagram, which is shown in figure 2. The main process is called kwic, which stands for *keyword in context*.

434 Software Quality Management

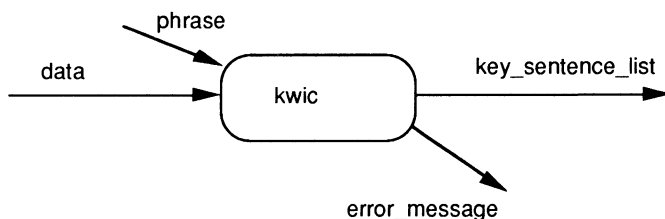


Figure 2

The words in the text file are represented by `data`, `phrase` contains the keywords being sought, and `key_sentence_list` is the result: these details are given in the data dictionary (see Appendix 1).

The `kwic` process must be put in context, by showing the source of the data and the destination of the result, which in this case are both text files. This is shown in the context diagram in figure 3.

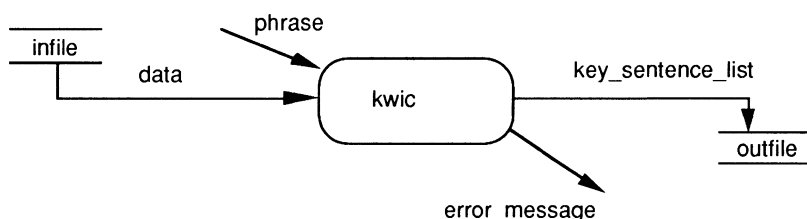


Figure 3

We must now refine the `kwic` process. The informal specification indicates that some punctuation characters, tabs and newlines are to be ignored, and so we decide that such characters will be replaced by a blank. However, this process changes the format of the text, so (with a view to producing reusable, general-purpose software) we should remove any redundant blanks which have been inserted, making sure that each word is still separated from the next by a blank. We then take the resulting list of characters and split it into phrases corresponding to sentences. Finally, we must find the phrase or phrases in which the keywords occur. This refinement process is shown in the data flow diagram of figure 4.

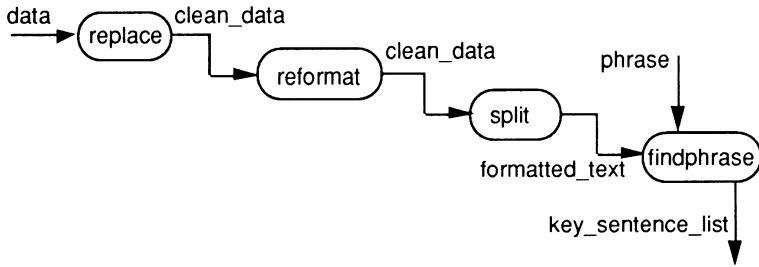


Figure 4

As before, descriptions of all of the identifiers used in the diagrams, together with their signatures (i.e. their types) are provided in the data dictionary. The data dictionary will be used during the structured design stage of the development process.

Structured design

Each process can now be designed using pseudocode. We start with the top-level process, `kwic`. This can be pseudocoded as shown below, where the first line shows the function's type:

```
kwic :: phrase → [char] → text
search a text file for a phrase
return the sentences in which the phrase occurs
or an error message if the phrase is not present
```

The functions shown in figure 4 must also be specified. The function `replace` is shown below:

```
replace :: [char] → [char]
for a list of char do
  if the char is to be ignored then
    replace it by a blank
  otherwise do nothing
```

The function `reformat` removes any redundant spaces inserted by `replace`.

```
reformat :: [char] → [char]
for a list of char do
  if 2 consecutive chars are both blanks, then
    remove the first
  otherwise do nothing
```

From the data dictionary, we see that the function `split` takes a list of characters and splits them up into phrases, where each phrase corresponds to a sentence. The function can be specified as shown below:

436 Software Quality Management

```
split :: [char] → [phrase]
for a list of char do
  if the list is empty then nothing
  if the 1st char is a full stop or a blank then ignore it
  otherwise
    find a sequence of chars ending in a full stop
    add this to the result of splitting the rest of the text
```

Finally, the function `findphrase`, which returns the sentences which contain the keywords, can be refined as:

```
findphrase :: phrase → text → text
for some text do
  search for the sentences in which the phrase is a substring
  return these sentences
```

This implies the need for a substring function:

```
substring :: phrase → phrase → bool
for 2 phrases do
  return true if the 1st is contained within the 2nd
```

A prototype

The program can now be prototyped. We use the pure, lazy functional language Miranda™ (Turner, [12]), as it is an elegant and expressive language which is supported by a robust programming environment.

From the structured analysis and structured design, we see that we can implement the function `kwic` as shown below:

```
kwic w txt = key_sentence_list, key_sentence_list ~ = []
            = error "word(s) not present", otherwise
            where
              key_sentence_list = findphrase w (split(reformat(replace txt)))
```

The function `error` is a built-in function, which takes a string and creates an error value with the associated message.

The implementation of `replace` is given below:

```
replace = map valid
        where
          valid c = ' ', if c $in punctuation ∨ c $in ignored_spaces || $ denotes an infix operator
                  = c , otherwise
                  where
                    punctuation = ",.;():\\\"'"
                    ignored_spaces = "\\n\\t"
```

This uses the function `map`, which is provided as a built-in function. It applies its functional parameter to each of the items in a list.

The function `in` takes an item and a list and returns true if the item is present. It can be defined by mapping the function `(=x)` over the list, and then using the built-in function `or` to determine whether or not one of the items in the resulting list is true:

```
in x xs = or (map (=x) xs)
```

The function `reformat` removes any redundant spaces inserted by `replace`:

```
reformat [] = []
reformat [c] = [], c = ''
               = [c], otherwise
reformat (c:d:cs) = reformat (d:cs), c = '' & d = ''
                   = (c:reformat (d:cs)), otherwise
```

The implementation of the function `split` is shown below:

```
split [] = []
split (':':cs) = split cs
split (' ':cs) = split cs
split cs = sent:(split rest)
           where
             sent = takewhile (~=':') cs           || ~ is logical negation
             rest = (dropwhile (~=':') cs)
```

The function `takewhile` is provided as a built-in function. When applied to a function and a list, `takewhile` gets items from the front of the list as long as the function returns true:

```
takewhile f [] = []
takewhile f (a:x) = a: takewhile f x, if f a
                  = [], otherwise
```

Similarly, `dropwhile`, applied to a function and a list, drops items from the front of the list as long as the function is true.

Finally, the function `findphrase` can be prototyped as:

```
findphrase p txt = [s | s ← txt; p $ substring s ]
```

where:

```
substring s l = or [s = take (#s) (drop n l) | n ← [0..#l-#s]]
```

Both of these functions have been implemented using list comprehensions. These constructs, which can be written more generally as:

$$[x \mid x \leftarrow xs ; p \ x]$$

can be understood by reading them as Zermelo-Fraenkel set abstractions. Thus, the general list comprehension shown above can be read as "the set of all x , such that x is taken from the set xs , and $(p \ x)$ is true".

The `substring` function also uses `take` and `drop`, two built-in functions which take or drop (respectively) the given number of items from a list.

Software validation

The implementation technique advocated is the evolutionary prototyping methodology outlined in Myers [13] and Beizer [14]. This means that we can validate the implementation using an incremental bottom-up methodology.

We can black-box test the functions by calling them with random input data. For example, we could test `replace` as shown below, where the system's response is shown on the second line.

438 Software Quality Management

```
replace "the;(a):it,x,,"  
>the a it x .
```

etc.

We can also perform dynamic testing by implementing dedicated validation functions, such as the function `goodchars` shown below:

```
goodchars xs = ~or [x $in punc ∨ x $in ignored_spaces | x ← xs]  
  where  
    punc = ",:();\\\""  
    ignored_spaces = "\\n\\t"
```

This uses the function `in`, which must also be tested thoroughly. Once `goodchars` has been tested we can compose it with `replace`, using the function composition operator, `(.)`, as shown below:

```
(goodchars.replace) "the;(a):it,x,,"  
>True
```

An example of black-box testing for the function `split`, which splits a list of characters into sentences, is shown below:

```
split "The early bird. Catches the worm"  
>["The early bird", "Catches the worm"]
```

The substring function can be tested in a similar manner.

We can perform integration testing for all the functions by composing them together and passing random input data to them, as the following shows:

```
(split.reformat.replace.read) "sayings"  
>["The cat sat on the mat", "The quick brown fox jumped over the lazy dog", "It never  
rains but it pours", "The early bird catches the worm", "More haste less speed"]
```

where the function `read` is a built-in function which returns the contents of a text file. The test data is given in Appendix 3.

We can also perform integration testing by using the function `findphrase`, which uses the functions which have been validated above:

```
findphrase "early bird" ((split.reformat.replace.read) "sayings")  
> ["The early bird catches the worm"]
```

We should also test the top-level function, `kwic`, as shown below:

```
kwic "rains but" (read "sayings")  
> ["It never rains but it pours"]
```

etc.

This validation procedure enables potential mistakes to be discovered early in the development cycle.

Documentation

The final part of the development process involves producing the structure charts which show the inter-function communication structure. The structure chart shown in figure 5 was reverse engineered from the code given in the previous section. An arrow from one function to another indicates that the function at the start of the arrow calls the other function. Rectangles with double bars on either side indicate functions which are provided in the standard Miranda environment.

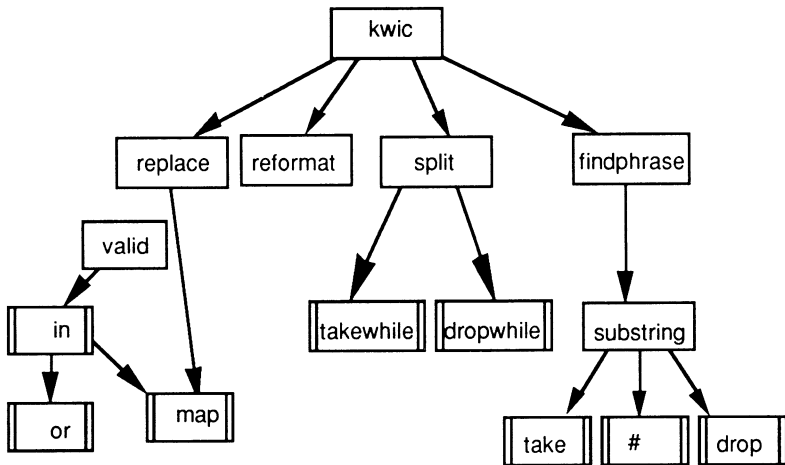


Figure 5

The chart shows the function call hierarchy only: traditionally, actual parameters would also be shown on the arcs between the functions. The omission of variable parameters reflects the reduced importance which variables have in functional programs. However, we have shown functional parameters (which are used by higher order functions) on the arcs. Note also that calls to the function `(:)` (cons) are not shown, nor are calls to the primitive logical operators `&`, `v`, `~`, or recursive function calls.

QUALITY ASSURANCE

The resulting system has the advantages of the use of both the structured analysis and structured design methodology and also the use of a pure functional language as the implementation language. Pure functional languages are referentially transparent and so the semantics of functional programs can be guaranteed. The program was implemented using bottom-up, evolutionary prototyping, together with design-time testing (isolation testing, integration testing and final system testing). This helps to increase confidence in the validity of the software. The quality of the software can be measured using the guidelines provided by the structured analysis and structured design methodology [1]. Coupling between modules is by means of data only, which is the most desirable form of coupling and indicates that the program is well-designed. The cohesion of the modules is functional, which indicates that the

440 Software Quality Management

program is highly maintainable. Taken together, these heuristics imply that the implemented system is likely to be easy to understand, modify and reuse.

PROVING PROPERTIES OF THE PROTOTYPE

As well as validating the prototype for various test data it is instructive to attempt to prove some general properties. As pure functional languages are referentially transparent we can do this either by using mathematical techniques such as induction or by reducing expressions to normal form. For example, we can verify the operational semantics of substring as shown below.

Suppose that we want to prove that the expression:

"bird" \$substring "early bird"

reduces to the term True.

Proof

This proof assumes that the standard functions or, take, drop and # (length) have been verified previously.

The definition of the function substring is given below:

substring s l = or [s = take (#s) (drop n l) | n ← [0..#l-#s]]

Instantiating the second parameter as "early bird", the length of the first parameter as 4 and the length of the second as 10, we wish to determine the normal form of the following expression:

or [s = take 4 (drop n "early bird") | n ← [0..(10-4)]]

the reduction proceeds as follows, where each reduction step is indicated by an arrow (→).

or [s = take 4 (drop n "early bird") | n ← [0..(10-4)]]

→ or [s = take 4 (drop 0 "early bird"), s = take 4 (drop 1 "early bird"),
s = take 4 (drop 2 "early bird"), s = take 4 (drop 3 "early bird"),
s = take 4 (drop 4 "early bird"), s = take 4 (drop 5 "early bird"),
s = take 4 (drop 6 "early bird")]

→ or [s = take 4 "early bird", s = take 4 "arly bird", s = take 4 "rly bird",
s = take 4 "ly bird", s = take 4 "y bird", s = take 4 " bird",
s = take 4 "bird"]

→ or [s = "earl", s = "arly", s = "rly ", s = "ly b", s = "y bi", s = "bir", s = "bird"]

→ or [False, False, False, False, False, False, True]

→ True

RELATED WORK

The benefits of prototyping have been propounded by a number of authors, including Tanik and Yeh [3], Boehm *et al.* [4], Boehm [5], and Hekmatpour and Ince [6]. The idea of utilizing functional languages for rapid prototyping is not new; for example, see Henderson [15]. More recently, the benefits of the declarative language OBJ [16] as a language for writing specifications have been expounded in Gerrard *et al.* [17], where it is argued that design time testing is a practical way of increasing confidence in the correctness of software, and that the adoption of formal methods together with design time testing can play a significant part in improving software quality. The methodology of structured analysis and structured design has numerous proponents, including De Marco [1], Page-Jones [2] and Gane and Sarson [18]. However, Coad and Yourdon [19] discuss an alternative to structured analysis (Object-Oriented Analysis), and argue that OOA has a number of advantages over structured analysis.

CONCLUSIONS

This paper has shown how evolutionary prototyping and functional languages can be used in the software development process to produce quality software. These functional prototypes can be used for verification and validation, highlighting potential problems and allowing alternative design decisions to be experimented with. We have also shown that the methodology of structured analysis and structured design can be used successfully with functional languages to produce robust, reusable software. We conclude that the combination of structured analysis and structured design, evolutionary prototyping and functional languages produces a powerful methodology which facilitates the development of quality software.

Acknowledgements

Miranda is a trademark of Research Software Ltd.

References

- [1] De Marco, T. , *Structured Analysis and System Specification*, Yourdon Press, 1978.
- [2] Page-Jones, M., *The Practical Guide to Structured Systems Design*, Yourdon Press, 1980.
- [3] Tanik, M.M. , Yeh, R.T. (eds), *Rapid Prototyping in Software Development*, *IEEE Computer*, 22 (5), May 1989.
- [4] Boehm, B.W., Gray, T.E., Seewaldt, T., 'Prototyping versus Specifying: A Multiproject Experiment', *IEEE Trans. Software Engineering*, 10 (3), pp. 290-303, 1984
- [5] Boehm, B. , 'A Spiral Model of Software Development and Enhancement', *IEEE Computer*, pp. 61-72, May 1988.
- [6] Hekmatpour, S., Ince, D., *Software Prototyping, Formal Methods and VDM*, Addison Wesley, 1988.

442 Software Quality Management

- [7] Hughes, R.J.M. 'Why Functional Programming Matters', *The Computer Journal*, 32, (2), pp. 98-107, 1989.
- [8] Stoy, J.E., *The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, 1977.
- [9] Darlington, J., 'Program Transformation', in Darlington, J., Henderson, P., Turner, D.A., (editors), *Functional Programming and its Applications: An Advanced Course*, pp. 193-209, Cambridge University Press, 1982.
- [10] Harrison, R., *Abstract Data Types in Modula-2*, John Wiley & Sons Ltd., 1989.
- [11] Gorlen, K., Orlow, S.M., Plixco, P.S., *Data Abstraction and Object-oriented Programming in C++*, John Wiley & Sons Ltd., 1990.
- [12] Turner, D.A., 'Miranda: A Non-strict Functional Language with Polymorphic Types', in *Functional Programming Languages and Computer Architecture*, Nancy, France, Springer-Verlag LNCS 201, pp. 1-16, September 1985.
- [13] Myers, G.J., *The Art of Software Testing*, John Wiley & Sons, 1979.
- [14] Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold, 1983.
- [15] Henderson, P., 'Functional Programming, Formal Specification, and Rapid Prototyping', *IEEE Transactions on Software Engineering*, SE-12 (2), pp. 241-250, 1986.
- [16] Goguen, J.A., 'Parameterized Programming', *IEEE Transactions on Software Engineering*, SE-10 (5), pp. 528-544, 1984.
- [17] Gerrard, C.P., Coleman, D., Gallimore, G.M., 'Formal Specification and Design Time Testing', *IEEE Transactions on Software Engineering*, SE-16 (1), pp. 1-11, 1990.
- [18] Gane, C., Sarson, T., *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall, 1978.
- [19] Coad, P., Yourdon, E., *Object-Oriented Analysis* (2nd edition), Yourdon Press, Prentice-Hall International, 1991.

APPENDIX 1

(The symbol :: indicates a type declaration.)

Data Dictionary

clean_data = list of char with punctuation chars replaced by blanks
clean_data :: [char]
data = a list of char, the input data
data :: [char]
error_message = a list of char, the output if an error occurs
error_message :: [char]
findphrase = function which takes a phrase and some text and returns the sentences in which the phrase occurs
findphrase :: phrase → text → text
formatted_text = a list of phrases, each of which represents a sentence
formatted_text :: [phrase]
infile = a text file of input data
infile :: [char]
key_sentence_list = the output, a list of phrases
key_sentence_list :: text

kwic = a function which takes a phrase and some unformatted text and returns a list of the sentences in the text in which the phrase occurs , or an error message, if the phrase is not present
kwic :: phrase → [char] → text
outfile = a text file of output data
outfile :: [char]
phrase = a list of char, representing a sentence
phrase :: [char]
replace = function which takes a list of char and replaces all punctuation marks, tabs and newlines by blanks
replace :: [char] → [char]
reformat = function which removes blanks inserted by replace
reformat :: [char] → [char]
split = function which takes a list of char and returns a list of phrases
split :: [char] → [phrase]
substring = a function which takes 2 phrases, and returns true if the 1st is contained within the 2nd
substring :: phrase → phrase → bool
text = a list of phrases, each of which represents a sentence
text :: [phrase]

APPENDIX 2

A collection of the functions used in the text.

phrase == [char]
||the type phrase will be used to indicate a list of chars, possibly including blanks.
text == [phrase]

|| Replace punctuation characters in a list of characters with spaces

```
replace :: [char] -> [char]
replace = map valid
  where
    valid c = ' ', if c $in punctuation ∨ c $in blanks
            = c , otherwise
    where
      punctuation = ",;():\`\"'"    ||nb don't remove '.'
      blanks = "\n\t"
```

```
in :: * -> [*] -> bool
in x xs = or (map (==x) xs)
```

||reformat removes the spurious spaces which are inserted by replace

```
reformat :: [char] -> [char]
reformat (c:d:cs) = reformat (d:cs), c = ' ' & d = ' '
                  = (c:reformat (d:cs)), otherwise
reformat [c] = [], c = ' '
              = [c], otherwise
reformat [] = []
```

||split splits text up into phrases, corresponding to sentences.
||It also removes full stops.

```
split :: [char] -> [phrase]
```

444 Software Quality Management

```
split [] = []
split (':':cs) = split cs
split (' ':cs) = split cs           ||applies if ' ' occurs between phrases
split cs = sent:(split rest)
    where
        sent = takewhile (~=':') cs
        rest = dropwhile (~=':') cs
```

||kwic takes a phrase and some unformatted text and returns a list of
||the sentences in the text which the phrase occurs in.

```
kwic :: phrase -> [char] -> text
```

```
kwic w txt = key_sentence_list, key_sentence_list ~= []
    = error "word(s) not present", otherwise
    where
        key_sentence_list = findphrase w (split(reformat(replace txt)))
```

```
findphrase:: phrase -> text -> text
```

```
findphrase p txt = [s | s <- txt; p $ substring s ]
```

```
substring :: phrase -> phrase -> bool
```

```
substring s l = or [s = take (#s) (drop n l) | n <- [0..#l-#s]]
```

APPENDIX 3

The test data used here was stored in 2 files, called empty and sayings.

"empty": an empty file

"sayings":

The cat sat on the mat.
The quick brown fox jumped over the lazy dog.
It never rains but it pours.
The early bird catches the worm.
More haste, less speed.