GOFER

Mark P. Jones

October 23, 1995

```
     __________   __________   __________   __________   ________
    /  _______/  /  ____   /  /  _______/  /  _______/  /  ____  \
   /  / _____   /  /   /  /  /  /______   /  /______   /  /___/  /
  /  / /_   /  /  /   /  /  /  _______/  /  _______/  /  __   __/
 /  /___/  /  /  /___/  /  /  /         /  /______   /  /  \  \
/_________/  /_________/  /__/         /_________/  /__/    \__\
```

Functional programming environment, Version 2.20

An Introduction to Gofer

draft version only

please report any errors, suggestions for improvements, extensions (or deletions!) to `jones-mark@cs.yale.edu`

This version includes a number of small corrections made since the original release.

This LaTeX version of the manual was prepared by Jeroen Fokker (`jeroen@cs.ruu.nl`).

Contents

Chapter 1

Introduction

Gofer is a functional programming environment (in other words, an interpreter) that I have implemented for my own personal use as part of my research into 'qualified types'. Nevertheless, the system is sufficiently complete for me to believe that Gofer may be of interest and use to others interested in the field of functional programming.

These notes give a brief introduction to the Gofer system and include some examples of Gofer programs. They are not the notes that I originally intended to write, being somewhat longer and perhaps more tutorial in nature. Nevertheless, you will not be able to learn functional programming from this document alone. A number of useful references are given in the reading list at the end of this document. In particular, the book by Bird and Wadler [1] is particularly good as a general introduction to the use, techniques and theory of functional programming. Although their notation is a little different from the language used by Gofer, it is a relatively straightforward task to translate between the two, and some suggestions for this are given in a appendix D. More importantly, the underlying semantics of Gofer do correspond to those expected by the authors of [1].

Whereas the work involved in investigating and implementing the ideas on which Gofer is based were motivated largely by my own program of work, the writing of these notes has rather more to do with the hope that Gofer will be useful to others. I would therefore be very grateful for any feedback on any aspect of the these notes (or of the Gofer system itself). Please let me know if you discover any errors, or if you find particular sections of these notes rather hard to follow. Suggestions for improvements or extensions are more than welcome.

Chapter 2

Background and acknowledgements

The language supported by Gofer is both syntactically and semantically similar to that of the functional programming language Haskell [5]. My principal task in the implementation of Gofer has therefore been to decide which features I should omit and then to implement what remains. Features common to both include:

- Non-strict semantics (lazy evaluation).
- Higher-order functions.
- Extended polymorphic type system with support for user-defined overloading.
- User-defined algebraic datatypes.
- Pattern matching.
- List comprehensions.
- Facilities for I/O, whilst retaining referential transparency within a program.

For the benefit of readers familiar with Haskell, the following features of Haskell are not supported in the standard version of Gofer:

- Modules.
- Arrays.
- Defaults for unresolved overloading.
- Derived instances of standard classes.
- Contexts in datatype definitions.
- Full range of numeric types and classes.

But Gofer is not just a partial implementation of Haskell; it also includes a number of experimental features which extend the type system in several ways:

- An alternative approach to type classes which avoids the need for construction of dictionaries during the evaluation of an expression.
- Type classes may take multiple parameters.
- Instances of type classes may be defined at arbitrary non-overlapping types.
- Contexts may include arbitrary type expressions.

These extensions stem from my own research [8, 9, 10, 11, 12] and were among the principal motivations for the development of Gofer. Full details of the differences between Gofer and Haskell 1.1 are given in appendix C.

Gofer would not have been implemented without my original introduction to functional programming using Orwell [6], and I am particularly grateful to Quentin Miller for answering so many of my questions about functional programming and about the Orwell system in particular. I should also like to mention the influence of the Haskell B. compiler from Lennart Augustsson and Thomas Johnsson and based on their earlier LML compiler [7].

Right from the beginning, I wanted to be able to use Gofer on a range of machines - and in particular, on the humble PC that I use at home. With this in mind, Gofer was actually developed on that same PC using Borland's Turbo C 1.5 and a public domain version of the yacc parser generator that I picked up some time ago. Gofer was also written with some degree of portability in mind and has subsequently been compiled to run on Sun workstations. I hope it will also be possible to port it to other platforms. It is my intention that Gofer be distributed complete with source code and I hope that this will be of interest to some users.

Many of the ideas used in the back-end of the Gofer system (i.e. the compiler and abstract machine) originate from the chapters of Simon Peyton Jones textbook [2]; I very much doubt whether Gofer would have been completed without frequent reference to that book. The lambda-lifter used in Gofer is based on Thomas Johnsson's algorithm described in [3].

On the theoretical side, I'm grateful to Phil Wadler for the encouragement that he has given me with my work on qualified types. Many of the basic ideas that I have used were inspired by his original paper motivating the use of type classes [4].

Chapter 3

Starting Gofer

The Gofer interpreter is usually entered by giving the command `gofer`, after which a display something like the following will normally be produced:

```
Gofer Version 2.20

Reading script file "/gofer/prelude":
Parsing..................................................
Dependency analysis......................................
Type checking............................................
Compiling................................................

Gofer session for:
/gofer/prelude
Type :? for help
?
```

The file name `/gofer/prelude` mentioned in the output above is the name of a file of standard definitions which are loaded into Gofer each time that the interpreter is started. By default, Gofer reads these definitions from a file called `prelude` in the current working directory. Alternatively you can set the environment variable `GOFER` to the name of the standard prelude file, which will then be used, whatever the current working directory might be.

Most commands in Gofer take the form of a colon followed by one or more characters which distinguish one command from another. There are two commands which are particularly worth remembering:

- `:q` exits the Gofer interpreter. On most systems, you can also exit from Gofer by typing the end of file character (`^Z` on an MS-DOS machine, usually `^D` on a Unix based machine).
- `:?` prints a list of all the commands, which can be useful if you forget the name of the command that you want to use.

The complete range of commands supported by the Gofer interpreter is described in appendix F.

Note that the interrupt key (`^C` on most systems) can be used at any time whilst using Gofer to abandon the process of reading in a file of function definitions or the evaluation of an expression. When the interrupt key is detected, Gofer prints the string `{Interrupted!}` and prints the '`? `' prompt so that further commands can be entered.

Chapter 4

Using Gofer

Using Gofer is rather like using a high-level programmable calculator; Once the interpreter is loaded, the system prints a prompt `?` and waits for you to enter an expression, and then press the enter (return) key. Once the input is complete, Gofer evaluates the expression and prints its value on the terminal, before returning to the original prompt and waiting for the next expression. For example:

```
? (2+3)*8
40
(5 reductions, 9 cells)
? sum [1..10]
55
(91 reductions, 130 cells)
?
```

In the first example, the user entered the expression `(2+3)*8`, which was evaluated by Gofer and the result `40` printed on the terminal. At the end of any calculation, Gofer displays the number of reductions (a measure of the amount of work) and cells (a measure of the amount of memory) that were used during the calculation. These figures can be useful for comparing the performance of different ways of carrying out the same calculation.

In the second example, the user typed the expression `sum [1..10]`. The notation `[1..10]` represents the list of integers between 1 and 10 inclusive, and `sum` is a standard function which can be used to determine the sum of a list of integers. Thus the result obtained by Gofer is:

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55$$

We could have typed this sum into Gofer directly:

```
? 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
55
(10 reductions, 23 cells)
?
```

and this calculation is certainly more efficient as it uses only $\frac{1}{9}$th of the number of reductions and $\frac{1}{5}$th of the number of cells as the original calculation. On the other hand, the original expression is much shorter and you are much less likely to make a mistake typing in the expression `sum [1..200]` than you would be if you tried to enter the sum of the integers from 1 to 200 directly.

You will learn more about the kind of expressions that can be entered into Gofer in the rest of this document.

Chapter 5

Standard and user-defined functions

The function `sum` used in the examples above, and indeed the addition and multiplication functions (+) and ($*$), are all standard functions which are included as part of a large collection of functions called the 'standard prelude' which are loaded into the Gofer system each time that you start the interpreter. Quite a number of useful calculations can be carried out using these functions alone, but for more general use you can also define your own functions and store the definitions in a file so that these functions can be loaded and used by by the Gofer system. For example, suppose that you create a file `fact` containing the following definition:

```
fact n = product [1..n]
```

The `product` function is another standard function which can be used to calculate the product of a list of integers, and so the line above defines a function `fact` which calculates the factorial of its argument. In standard mathematical notation, $\mathit{fact}\ n = n!$ which is usually defined informally by an equation of the form:

$$n! = 1 * 2 * \cdots * (n-1) * n$$

Once you become familiar with the notation used by Gofer, you will see that the Gofer definition of the factorial function is really very similar to this informal mathematical definition.

In order to use this definition from the Gofer interpreter, we must first load the definitions of the file into the interpreter. The simplest way to do this uses the `:l` command:

```
? :l fact
Reading script file "fact":
Parsing................................................
Dependency analysis....................................
Type checking..........................................
Compiling..............................................

Gofer session for:
/gofer/prelude
fact
?
```

Notice the list of filenames displayed after `Gofer session for:`; this tells you which files of definitions are currently being used by Gofer, the first of which is the file containing the definitions for the standard prelude. Since the file containing the definition of the factorial function has now been loaded, we can make use of this function in expressions entered to the interpreter:

```
? fact 6
720
(57 reductions, 85 cells)
```

For another example, recall the standard mathematical formula which tells us that the number of ways of choosing r objects from a collection of n objects is given by $n!/(r! * (n-r)!)$. In Gofer, this function can be defined by:

```
comb n r = fact n /(fact r * fact (n-r))
```

In order to use this function, we can either edit the file `fact` which contains the definition of the factorial function, adding the definition of `comb` on a new line, or we can include the definition as part of an expression entered whilst using Gofer:

```
? comb 5 2 where comb n r = fact n /(fact r * fact (n-r))
10
(110 reductions, 161 cells)
?
```

The ability to define a function as part of an expression like this is often quite useful. However, if the function `comb` were likely to be wanted on a number of occasions, it would be more sensible to add its definition to the contents of the file `fact`, instead of having to repeat the definition each time it is used.

You will learn more about the functions defined in the standard prelude and find out how to define your own functions in the following sections.

Chapter 6

Function names: identifiers and operators

As the examples of the previous section show, there are two kinds of name that can be used for a function; identifiers such as `sum` and operator symbols such as `+` and `*`. Choosing the appropriate kind of name for a particular function can often help to make expressions involving that function easier to read. If for example the addition function was represented by the name `plus` rather than the operator symbol `+` then the sum of the integers from 1 to 5 would have to be written as:

```
plus (plus (plus (plus 1 2) 3) 4) 5
```

In this particular case, another way of writing the same sum is:

```
plus 1 (plus 2 (plus 3 (plus 4 5)))
```

Not only does the use of the identifier `plus` make these expressions larger and more difficult to read than the equivalent expressions using `+`; it also makes it very much harder to see that these two expressions do actually have the same value.

Gofer distinguishes between the two types of name according to the way that they are written:

- An identifier begins with a letter of the alphabet optionally followed by a sequence of characters, each of which is either a letter, a digit, an apostrophe (`'`) or an underbar (`_`). Identifiers representing functions or variables must begin with a lower case letter (identifiers beginning with an upper case letter are used to denote a special kind of function called a 'constructor function' described in section 11.1). The following identifiers are examples of Gofer variable and function names:

  ```
  sum    f    f''    integerSum    african_queen    do'until'zero
  ```

  The following identifiers are reserved words in Gofer and cannot be used as the name of a function or variable:

  ```
  case      of         where     let        in        if
  then      else       data      type       infix     infixl
  infixr    primitive  class     instance
  ```

- An operator symbol is written using one or more of the following symbol characters:

  ```
  :  !  #  $  %  &  *  +  .  /  <  =  >  ?  @  \  ^  |  -
  ```

  In addition, the tilde character (`~`) is also permitted, although only in the first position of an operator name[1]. Operator names beginning with a colon are used for constructor functions in the same way as identifiers beginning with a capital letter as mentioned above. In addition, the following operator symbols have special uses in Gofer:

[1] Haskell also makes the same restriction for the minus/dash character (`-`).

```
::    =    ..    @    \    |    <-    ->    ~    =>
```

All other operator symbols can be used as variables or function names, including each of the following examples:

```
+    ++   &&   ||    <=   ==   /=   //  .
==>  $    @@    -*-  \/   /\   ...  ?
```

Note that each of the symbols in the first line is used in the standard prelude. If you are interested in using Gofer to develop programs for use with a Haskell compiler, you might also want to avoid using the operator symbols `:=`, `!`, `:+` and `:%` which are used to support features in Haskell not currently provided by the Gofer standard prelude.

Gofer provides two simple mechanisms which make it possible to use an identifier as an operator symbol, or an operator symbol as an identifier:

- Any identifier will be treated as an operator symbol if it is enclosed in backquotes (`) – for example, the expressions using the `plus` function above are a little easier to read using this technique:

  ```
  (((1 `plus` 2) `plus` 3) `plus` 4) `plus` 5
  ```

  In general, an expression of the form ``x `op` y`` is equivalent to the corresponding expression `op x y`, whilst an expression such as `f x y z` can also be written as ``(x `f` y) z``.[2]

- Any operator symbol can be treated as an identifier by enclosing it in parentheses. For example, the addition function denoted by the operator symbol `+` is often written as `(+)`. Any expression of the form `x + y` can also be written in the form `(+) x y`.

There are two more technical problems which have to be dealt with when working with operator symbols:

- Precedence: Given operator symbols $(+)$ and $(*)$, should $2*3+4$ be treated as either $(2*3)+4$ or $2*(3+4)$?

  This problem is solved by assigning each operator a precedence value (an integer in the range 0 to 9). In a situation such as the above, we simply compare the precedence values of the operators involved, and carry out the calculation associated with the highest precedence operator first. The standard precedence values for $(+)$ and $(*)$ are 6 and 7 respectively so that the expression above will actually be treated as $(2*3)+4$.

- Grouping: The above rule is only useful when the operator symbols involved have distinct precedences. For example, should the expression $1-2-3$ be treated as either $(1-2)-3$ giving a result of -4, or as $1-(2-3)$ giving a result of 2?

  This problem is solved by giving each operator a 'grouping' (sometimes called its associativity). An operator symbol $(\oplus)$ is said to:

  - group to the left if $x \oplus y \oplus z$ is treated as $(x \oplus y) \oplus z$
  - group to the right if $x \oplus y \oplus z$ is treated as $x \oplus (y \oplus z)$

  A third possibility is that an expression of the form $x \oplus y \oplus z$ is to be treated as ambiguous and will be flagged as a syntax error. In this case we say that the operator $(\oplus)$ is non-associative.

  The standard approach in Gofer is to treat $(-)$ as grouping to the left so that $1-2-3$ will actually be treated as $(1-2)-3$.

By default, every operator symbol in Gofer is treated as non-associative with precedence 9. These values can be changed by a declaration of one of the following forms:

`infixl digit ops`	to declare operators which group to the left
`infixr digit ops`	to declare operators which group to the right
`infix digit ops`	to declare non-associative operators

[2] For those using Gofer on a PC, you may find that your keyboard does not have a backquote key! In this case you should still be able to enter a backquote by holding down the key marked ALT, pressing the keys '9' and then '6' on the numeric keypad and then releasing the ALT key.

In each of these declarations ops represents a list of one or more operator symbols separated by commas and digit is an integer between 0 and 9 which gives the precedence value for each of the listed operator symbols. The precedence digit may be omitted in which case a value of 9 is assumed. There are a number of restrictions on the use of these declarations:

- Operator declarations can only appear in files of function definitions which are loaded into Gofer; they cannot be entered directly whilst using the Gofer interpreter.
- At most one operator declaration is permitted for any particular operator symbol (even if repeated declarations all specify the same precedence and grouping as the original declaration).
- Any file containing a declaration for an operator precedence and grouping must also contain a (top-level) declaration for that operator.

In theory, it is possible to use an operator declaration at any point in a file of definitions. In practice, it is sensible to ensure that each operator is declared before the symbol is used. One way to guarantee this is to place all operator declarations at the beginning of the file [this condition is enforced in Haskell]. Note that until an operator declaration for a particular symbol is encountered, any occurrence of that symbol will be treated as a non-associative operator with precedence 9.

The following operator declarations are taken from the standard prelude:

```
-- Operator precedence table

infixl 9 !!
infixr 9 .
infixr 8 ^
infixl 7 *
infix  7 /, `div`, `rem`, `mod`
infixl 6 +, -
infix  5 \\
infixr 5 ++, :
infix  4 ==, /=, <, <=, >=, >
infix  4 `elem`, `notElem`
infixr 3 &&
infixr 2 ||
```

and their use is illustrated by the following examples:

Expression:	Equivalent to:	Reasons:
`1 + 2 - 3`	`(1 + 2) - 3`	`(+)` and `(-)` have the same precedence and group to the left.
`x : ys ++ zs`	`x : (ys ++ zs)`	`(:)` and `(++)` have the same precedence and group to the right.
`x == y || z`	`(x == y) || z`	`(==)` has higher precedence than `(||)`.
`3 * 4 + 5`	`(3 * 4) + 5`	`(*)` has higher precedence than `(+)`.
``y `elem` z:zs``	``y `elem` (z:zs)``	`(:)` has higher precedence than `elem`.
`12 / 6 / 3`	syntax error	ambiguous use of `(/)`; could mean either `(12/6)/3` or `12/(6/3)`.

Note that function application always binds more tightly than any infix operator symbol. For example, the expression `f x + g y` is equivalent to `(f x) + (g y)`. Another example which often causes problems is the expression `f x + 1`, which is treated as `(f x) + 1` and not as `f (x+1)` as is sometimes expected.

Chapter 7

Built-in types

An important part of Gofer is the type system which is used to detect errors in expressions and function definitions. Starting with primitive expressions such as numeric constants, Gofer assigns a type to each expression that describes the kind of value represented by the expression.

In general we write *object* :: *type* to indicate that a particular expression has the indicated type. For example:

`42   :: Int`	indicating that 42 is an integer (`Int` is the name for the type of integer values).
`fact :: Int -> Int`	indicating that `fact` is a function which takes an integer argument and returns an integer value (its factorial).

The most important property of the type system is that it is possible to determine the type of an expression without having to evaluate it. For example, the information given above is sufficient to determine that `fact 42 :: Int` without needing to calculate 42! first.

Gofer has a wide range of built-in types, described in the following sections. In addition, Gofer also includes facilities for defining new types as well as types acting as abbreviations for complicated type expressions as described in section 11.

7.1 Functions

If `t1` and `t2` are types then `t1 -> t2` is the type of a function which, given an argument of type `t1` produces a result of type `t2`. A function of type `t1 -> t2` is said to have argument type `t1` and result type `t2`.

In mathematics, the result of applying a function f to an argument x is traditionally written as $f(x)$. In many situations, these parentheses are unnecessary and may be omitted when using Gofer. For example, if `f :: t1 -> t2` and `x :: t1` then `f x` is the result of applying `f` to `x` and has type `t2`.

If t_1, t_2,..., t_n are type expressions then:

$$t_1 \to t_2 \to \ldots \to t_n$$

can be used as an abbreviation for the type:

$$t_1 \to (t_2 \to (\ldots \to t_n) \ldots)$$

In a similar way, an expression of the form $f\ x_1\ x_2 \ldots x_n$ is simply an abbreviation for the expression $(\ldots((f\ x_1)\ x_2)\ldots x_n)$.

These two conventions allow us to deal with functions taking more than one argument rather elegantly. For example, the type of the addition function `(+)` is:

```
Int -> Int -> Int
```

In other words, `(+)` is a function which takes an integer argument and returns a value of type `(Int -> Int)`. For example, `(+)  5` is the function which takes an integer value n and returns the value of the integer n plus 5. Hence `(+) 5 4`, which is equivalent to `5 + 4`, evaluates to the integer 9 as expected.

7.2 Booleans

Represented by the type `Bool`, there are two boolean values written as `True` and `False`. The standard prelude includes several useful functions for manipulating boolean values:

- `(&&), (||) :: Bool -> Bool -> Bool`

  The value of the expression `b && d` is `True` if and only if both `b` and `d` are `True`. If `b` is `False` then `d` is not evaluated.

  The value of the expression `b || d` is `True` if either of `b` or `d` is `True`. If `b` is `True` then `d` is not evaluated.

- `not  :: Bool -> Bool`

  The value of the expression `not b` is the opposite boolean value to that of `b`; `not True = False`, `not False = True`.

Gofer includes a special form of 'conditional expression' which enables an expression to select between two alternatives according to the value of a boolean expression:

```
if b then t else f
```

is an expression which is equivalent to `t` if `b` evaluates to `True`, or to `f` if `b` evaluates to `False`. Note that an expression of this form is only acceptable if `b` is an expression of type `Bool` and if the types of `t` and `f` are the same, in which case the whole expression also has that type.

7.3 Integers

Represented by the type `Int`, the integer type includes both positive and negative integers such as -273, 0 and 383. Like many computer systems, the range of integer values that can be used is restricted and calculations using large positive or negative numbers may lead to (undetected) overflow.

A wide range of operators and functions are defined in the standard prelude for use with integers:

`(+)`	addition
`(*)`	multiplication
`(-)`	subtraction
`(^)`	raise to power
`negate`	unary negation
`(/)`	integer division
`div`	integer division
`rem`	remainder
`mod`	modulo
`odd`	returns `True` if argument is odd, `False` otherwise
`even`	returns `True` if argument is even, `False` otherwise
`gcd`	returns the greatest common divisor of its two arguments
`lcm`	returns the least common multiple of its two arguments
`abs`	returns the absolute value of its argument
`signum`	returns -1, 0 or 1 indicating that its argument is negative, zero or positive respectively

An expression of the form `-x` is treated as the expression `negate x`. 'Remainder' is related to integer division by the law:

```
(x `div` y)*y + (x `rem` y) == x
```

'Modulo' is like remainder except that the modulo has the same sign as the divisor. The less familiar operators are illustrated by the following identities:

```
3^4 == 81,          7 `div` 3 == 2,      even 23 == False
7 `rem` 3 == 1,    -7 `rem` 3 == -1,     7 `rem` -3 == 1
7 `mod` 3 == 1,    -7 `mod` 3 == 2,      7 `mod` -3 == -2
gcd 32 12 == 4,    abs (-2) == 2,        signum 12 == 1
```

7.4 Floating point numbers

Represented by the type `Float`, elements of this type can be used to represent fractional values as well as very large or very small quantities. Such values are however usually only accurate to a fixed number of digits and rounding errors may occur in some calculations making significant use of floating point quantities. A numeric value in an input expression will only be treated as a floating point number if it either includes a decimal point such as 3.14159, or if the number is too large to be stored as a value of type Int. Scientific notation may also be used to enter floating point quantities; for example `1.0e3` is equivalent to 1000.0, whilst `5.0e-2` is equivalent to 0.05. Floating point numbers are not included in all implementations of Gofer.

7.5 Characters

Represented by the type `Char`, elements of this type represent individual characters such as those entered at a keyboard. Character values are written as single characters enclosed by apostrophe characters: e.g. `'a'`, `'0'`, `'Z'`. Some special characters must be entered using an 'escape code'; each of these begins with a backslash character '\', followed by one or more characters to select the required character. Some of the most useful escape codes are:

`\\`	backslash
`\'`	apostrophe
`\"`	double quote
`\n`	newline
`\b` or `\BS`	backspace
`\DEL`	delete
`\t` or `\HT`	tab
`\a` or `\BEL`	alarm (bell)
`\f` or `\FF`	formfeed

Additional escape characters include:

`\^`*c*	control character, where *c* is replaced by one of the characters: `@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_` For example, `'\^A'` represents control-A.
`\`*number*	representing the character with ASCII value specified by the given decimal *number*.
`\o`*number*	representing the character with ASCII value specified by the given octal *number*.
`\x`*number*	representing the character with ASCII value specified by the given hexadecimal *number*.
`\`*name*	named ASCII control character, where *name* is replaced by one of the standard ASCII names e.g. `'\DC3'`.

In contrast with some common languages (such as C, for example) character values are quite distinct from integers; however the standard prelude does include functions:

```
ord :: Char -> Int
chr :: Int -> Char
```

which enable you to map a character to its corresponding ASCII value, or from an ASCII value to the corresponding character:

```
? ord 'a'
97
(2 reductions, 6 cells)
? chr 65
'A'
(2 reductions, 7 cells)
?
```

7.6 Lists

If `t` is a type then `[t]` is the type whose elements are lists of values of type `t`. There are several ways of writing list expressions:

- The simplest list of any type is the empty list, written `[]`.
- Non-empty lists can be constructed either by explicitly listing the members of the list (for example: `[1,3,10]`) or by adding a single element onto the front of another list using the `(:)` operator (pronounced 'cons'). These notations are equivalent:

  ```
  [1,3,10]  =  1 : [3,10]  =  1 : 3 : [10]  =  1 : 3 : 10 : []
  ```

  (the `(:)` operator groups to the right so `1:3:10:[]` is equivalent to `(1:(3:(10:[])))` – a list whose first element is 1, second element is 3 and last element is 10).

The standard prelude includes a wide range of functions for calculations involving lists. For example:

- `length xs` returns the number of elements in the list `xs`.
- `xs ++ ys` returns the list of elements in `xs` followed by the elements in `ys`
- `concat xss` returns the list of elements in each of the lists in `xss`
- `map f xs` returns the list of values obtained by applying the function `f` to each of the values in the list `xs` in turn.

Here are some examples using these functions:

```
? length [1,3,10]
3
(15 reductions, 28 cells)

? [1,3,10] ++ [2,6,5,7]
[1, 3, 10, 2, 6, 5, 7]
(19 reductions, 77 cells)

? concat [[1], [2,3], [], [4,5,6]]
[1, 2, 3, 4, 5, 6]
(29 reductions, 93 cells)

? map ord ['H', 'e', 'l', 'l', 'o']
[72, 101, 108, 108, 111]
(22 reductions, 73 cells)

?
```

Note that all of the elements in a list must be of the same type, so that an expression such as `['a', 2, False]` is not permitted.

At this point it might be useful to mention an informal convention that is used by a number of functional programmers when choosing names for variables representing elements of lists, lists themselves, lists of lists and so on. If for example, a typical element of a list is called `x`, then it is often useful to use the name `xs` for a list of such values, suggesting that a list contains a number of '`x`'s. Similarly, a list of lists might be called `xss`. Once you have understood this convention it is much easier to remember the relationship between the variables in the expression `(x:xs)` than it would be if different names had been used such as `(a:b)`.

7.7 Strings

A string is treated as a list of characters and the type `String` is simply an abbreviation for the type `[Char]`. Strings are written as sequences of characters enclosed between speech marks. All of the escape codes that can be used for characters may also be used in a string:

```
? "hello, world"
```

```
hello, world
(0 reductions, 13 cells)

? "hello\nworld"
hello
world
(0 reductions, 12 cells)
?
```

In addition, strings may contain the escape sequence `\&` which can be used to separate otherwise ambiguous pairs of characters within a string, e.g.:

`\123h`	represents the string `['\123', 'h']`
`\12\&3h`	represents the string `['\12', '3', 'h']`

A string expression may be spread over a number of lines using a gap – a non-empty sequence of space, tab and new line characters enclosed by backslash characters:

```
? "hell\   \o"
hello
(0 reductions, 6 cells)
?
```

Notice that strings are printed differently from other values, which gives the programmer complete control over the format of the output produced by a program. The only values that Gofer can in fact display on the terminal are strings. If the type of an expression entered into Gofer is equivalent to String then the expression is printed directly by evaluating and printing each character in the list in sequence. Otherwise, the expression to be evaluated, `e`, is replaced by the expression `show' e` where `show'` is a built-in function (defined as part of the standard prelude) which converts any value to a printable representation. The only way of printing a string value in the same way as any other value is by explicitly using the `show'` function:

```
? show' "hello"
"hello"
(7 reductions, 24 cells)
?
```

The careful reader may have been puzzled by the fact the number of reductions used in the first three examples above was zero. This is in fact quite correct since these expressions are constants and no further evaluation can be carried out. For constant expressions of any other type there will always be at least one reduction needed to print the value since the constant must first be translated to a printable representation using the `show'` function.

Because strings are represented as lists of characters, all of the standard prelude functions for manipulating lists can also be used with strings:

```
? length "Hello"
5
(22 reductions, 36 cells)

? "Hello, " ++ "world"
Hello, world
(8 reductions, 37 cells)

? concat ["super","cali","fragi","listic"]
supercalifragilistic
(29 reductions, 101 cells)

? map ord "Hello"
[72, 101, 108, 108, 111]
(22 reductions, 69 cells)
```

7.8 Tuples and the unit type

If $t_1, t_2, \ldots, t_n$ are types and $n \geq 2$, then there is a type of n-tuples written $(t_1, t_2, \ldots, t_n)$ whose elements are also written in the form $(x_1, x_2, \ldots, x_n)$ where the expressions x_1, x_2, ..., x_n have types t_1, t_2, ..., t_n respectively. For example,

```
(1, [2], 3)   :: (Int, [Int], Int)
('a', False)  :: (Char, Bool)
((1,2),(3,4)) :: ((Int, Int), (Int, Int))
```

Note that, unlike lists, the elements in a tuple may have different types, although the number of elements in the tuple is fixed.

The unit type is written `()` and has a single element which is also written as `()`. The unit type is of particular interest in theoretical treatments of the type system of Gofer, although you may occasionally find a use for it in practical programs.

Chapter 8

Errors

8.1 Errors detected on input

After an expression has been entered, but before any attempt is made to evaluate it, Gofer carries out a number of checks to make sure that the expression that you typed does not contain any errors. Here are some examples of the kind of problem that might occur:

- Syntax errors. The most common situation in which this happens is when you make a typing mistake, either leaving out some characters, or perhaps pressing the wrong keys instead. In the following example, the user has missed out a '`[`' character:

```
? sum 1..100]
ERROR: Syntax error in input (unexpected `..')
```

- Undefined variables. This happens when you enter an expression using a variable or function name that is not defined in any of the files of definitions loaded into Gofer. This can often mean that you have misspelt the name of a function, or that the files defining a function have not yet been loaded. For example:

```
? sum [1..n]
ERROR: Undefined variable "n"
```

- Type errors. Certain expressions are sensible only when the functions used in those expressions are applied to values of the appropriate type. For example, whilst the factorial function can be used to calculate the factorial of an integer, it is clearly meaningless to try to determine the factorial of a character value. This kind of problem can be detected using the types of the components of an expression. In the expression `fact 'A'`, we can see that the argument `'A'` has type `Char` which does not match the argument type Int of the factorial function. This error will be detected by Gofer if you try to evaluate the expression:

```
? fact 'A'
ERROR: Type error in application
*** expression     : fact 'A'
*** term           : 'A'
*** type           : Char
*** does not match : Int
```

8.2 Errors during evaluation

If no errors are detected in an input expression, Gofer then begins to evaluate that expression. Despite all of the checks that are carried out before the evaluation begins, it is still possible for an error to occur during the evaluation of an expression. A typical example of this is an attempt to divide a number by zero. In this case, Gofer prints the part of the expression being evaluated that caused the error, surrounded by braces '`{`' and '`}`':

```
? 3/0
{primDivInt 3 0}
(4 reductions, 30 cells)
?
```

The function `primDivInt` which appears here is a primitive function used to divide one integer (its first argument) by another (the second). If an error occurs in just one part of an expression, only the part causing the problem will be displayed:

```
? 4 + (5/0)
{primDivInt 5 0}
(5 reductions, 32 cells)
?
```

A standard function called `error` is defined in the standard prelude which is often useful for ensuring that appropriate error messages are produced when an error occurs:

```
? error "Problem has occurred"
{error "Problem has occurred"}
(23 reductions, 99 cells)
?
```

Chapter 9

More about value declarations

9.1 Simple pattern matching

Although the Gofer standard prelude includes many useful functions, you will usually need to define a collection of new functions for specific problems and calculations. The declaration of a function `f` usually takes the form of a number of equations of the form:

$$\texttt{f}\ pat_1\ pat_2\ \ldots\ pat_n\ \texttt{=}\ rhs$$

(or an equivalent expression, if `f` is written as by an operator symbol). Each of the expressions pat_1, pat_2, …, pat_n represents an argument to the function `f` and is called a 'pattern'. The number of such arguments is called the arity of `f`. If `f` is defined by more than one equation then they must be entered together and each one must give the same arity for `f`.

When a function is defined by more than one equation, it will usually be necessary to evaluate one or more of the arguments to the function to determine which equation applies. This process is called 'pattern-matching'. In all of the previous examples we have used only the simplest kind of pattern – a variable. As an example, consider the factorial function defined in section 5:

```
fact n = product [1..n]
```

If we then wish to evaluate the expression `fact 6` we first match the expression `6` against the pattern `n` and then evaluate the expression obtained from `product [1..n]` by replacing the variable `n` with the expression `6`. The process of matching the arguments of a function against the patterns in its definition and obtaining another expression to be evaluated is called a 'reduction'. Using Gofer, it is easy to verify that the evaluation of `fact 6` takes one more reduction than that of `product [1..6]`:

```
? fact 6
720
(57 reductions, 85 cells)
? product [1..6]
720
(56 reductions, 85 cells)
?
```

Many kinds of constants such as the boolean values True and False can also be used in patterns, as in the following definition of the function `not` taken from the standard prelude:

```
not True  = False
not False = True
```

In order to determine the value of an expression of the form `not b`, we must first evaluate the expression `b`. If the result is `True` then we use the first equation and the value of `not b` will be `False`. If the value of `b` is `False`, then the second equation is used and the value of `not b` will be `True`. Other constants, including integers, characters and strings may also be used in patterns. For example, if we define a function `hello` by:

```
hello "Mark"  =  "Howdy"
hello name    =  "Hello " ++ name ++ ", nice to meet you!"
```

then:

```
? hello "Mark"
Howdy
(1 reduction, 12 cells)
? hello "Fred"
Hello Fred, nice to meet you!
(13 reductions, 66 cells)
?
```

Note that the order in which the equations are written is very important because Gofer always uses the first applicable equation. If instead we had defined the function with the equations:

```
hello name    =  "Hello " ++ name ++ ", nice to meet you!"
hello "Mark"  =  "Howdy"
```

then the results obtained using this function would have been a little different:

```
? hello "Mark"
Hello Mark, nice to meet you!
(13 reductions, 66 cells)
? hello "Fred"
Hello Fred, nice to meet you!
(13 reductions, 66 cells)
?
```

There are a number of other useful kinds of pattern, some of which are illustrated by the following examples:

Wildcard:	`_`	matches any value at all; it is like a variable pattern, except that there is no way of referring to the matched value.
Tuples:	`(x,y)`	matches a pair whose first and second elements are called `x` and `y` respectively.
Lists:	`[x]`	matches a list with precisely one element called `x`.
	`[_,2,_]`	matches a list with precisely three elements, the second of which is the integer 2.
	`[]`	matches the empty list.
	`(x:xs)`	matches a non-empty list with head `x` and tail `xs`.
As patterns:	`p@(x,y)`	matches a pair whose first and second components are called `x` and `y`. The complete pair can also be referred to directly as `p`.
(n+k) patterns:	`(m+1)`	matches an integer value greater than or equal to 1. The value referred to by the variable `m` is one less than the value matched.

A further kind of pattern (called an irrefutable pattern) is introduced in section 9.11.

Note that no variable name can be used more than once on the left hand side of each equation in a function definition. The following example:

```
areTheyTheSame x x = True
areTheyTheSame _ _ = False
```

will not be accepted by the Gofer system, but should instead be defined using the notation of guards introduced in the next section:

```
areTheyTheSame x y
        | x==y      = True
        | otherwise = False
```

9.2 Guarded equations

Each of the equations in a function definition may contain 'guards' which require certain conditions on the values of the function's arguments to be met. As an example, here is a function which uses the standard prelude function `even :: Int -> Bool` to determine whether its argument is an even integer or not, and returns the string `"even"` or `"odd"` as appropriate:

```
oddity n | even n    = "even"
         | otherwise = "odd"
```

In general, an equation using guards takes the form:

```
f x1 x2 ... xn | condition1  =  e1
               | condition2  =  e2
               .
               .
               | conditionm  =  em
```

This equation is used by evaluating each of the conditions in turn until one of them evaluates to `True`, in which case the value of the function is given by the corresponding expression e on the right hand side of the '`=`' sign. In Gofer, the variable `otherwise` is defined to be equal to `True`, so that writing `otherwise` as the condition in a guard means that the corresponding expression will always be used if no previous guard has been satisfied.

As an aside: in the notation of [1], the above examples would be written as:

```
oddity n        =  "even",   if even n
                =  "odd",    otherwise
f x1 x2 ... xn  = e1,     if condition1
                = e2,     if condition2
                  .
                  .
                = em,     if conditionm
```

Translation between the two notations is relatively straightforward.

9.3 Local definitions

Function definitions may include local definitions for variables which can be used both in guards and on the right hand side of an equation. Consider the following function which calculates the number of distinct real roots for a quadratic equation of the form $a * x^2 + b * x + c = 0$:

```
numberOfRoots a b c | discr>0   =  2
                    | discr==0  =  1
                    | discr<0   =  0
                      where discr = b*b - 4*a*c
```

The operator `(==)` is used to test whether two values are equal or not. You should take care not to confuse this with the single `=` sign used in function definitions.

Local definitions can also be introduced at an arbitrary point in an expression using an expression of the form:

`let` *decls* `in` *expr*

For example:

```
? let x = 1 + 4 in x*x + 3*x + 1
41
(8 reductions, 15 cells)
? let p x = x*x + 3*x + 1  in  p (1 + 4)
41
(7 reductions, 15 cells)
?
```

9.4 Recursion with integers

Recursion is a particularly important and powerful technique in functional programming which is useful for defining functions involving a wide range of datatypes. In this section, we describe one particular application of recursion to give an alternative definition for the factorial function from section 5.

Suppose that we wish to calculate the factorial of a given integer n. We can split the problem up into two special cases:

- If n is zero then the value of $n!$ is 1.
- Otherwise, $n! = 1 * 2 * \ldots * (n-1) * n = (n-1)! * n$ and so we can calculate the value of $n!$ by calculating the value of $(n-1)!$ and then multiplying it by n.

This process can be expressed directly in Gofer using a conditional expression:

```
fact1 n  =  if n==0 then 1 else n * fact1 (n-1)
```

This definition may seem rather circular; in order to calculate the value of $n!$, we must first calculate $(n-1)!$, and unless n is 1, this requires the calculation of $(n-2)!$ etcetera ...However, if we start with some positive value for the variable n, then we will eventually reach the case where the value of 0! is required – and this does not require any further calculation. The following diagram illustrates how 6! is evaluated using `fact1`:

```
fact1 6  ==>  6 * fact1 5
         ==>  6 * (5 * fact1 4)
         ==>  6 * (5 * (4 * fact1 3))
         ==>  6 * (5 * (4 * (3 * fact1 2)))
         ==>  6 * (5 * (4 * (3 * (2 * fact1 1))))
         ==>  6 * (5 * (4 * (3 * (2 * (1 * fact1 0)))))
         ==>  6 * (5 * (4 * (3 * (2 * (1 * 1)))))
         ==>  6 * (5 * (4 * (3 * (2 * 1))))
         ==>  6 * (5 * (4 * (3 * 2)))
         ==>  6 * (5 * (4 * 6))
         ==>  6 * (5 * 24)
         ==>  6 * 120
         ==>  720
```

Incidentally, there are several other ways of writing the recursive definition of `fact1` above in Gofer. For example, using guards:

```
fact2 n
  | n==0        =  1
  | otherwise   =  n * fact2 (n-1)
```

or using pattern matching with an integer constant:

```
fact3 0         =  1
fact3 n         =  n * fact3 (n-1)
```

Which of these you use is largely a matter of personal taste.

Yet another style of definition uses the `(n+k)` patterns mentioned in section 9.1:

```
fact4 0         =  1
fact4 (n+1)     =  (n+1) * fact4 n
```

which is equivalent to:

```
fact5 n | n==0  =  1
        | n>=1  =  n * fact5 (n-1)
```

Although each of the above definitions gives the same result as the original `fact` function for each non-negative integer, the functions can still be distinguished by the values obtained when they are applied to negative integers:

- `fact (-1)` evaluates to the integer 1.
- `fact1 (-1)` causes Gofer to enter an infinite loop, which is only eventually terminated when Gofer runs out of 'stack space'.
- `fact4 (-1)` causes an evaluation error and prints the message `{fact4 (-1)}` on the screen.

To most people, this suggests that the definition of `fact4` is perhaps preferable to that of either `fact` or `fact1` as it neither gives the wrong answer without allowing this to be detected nor causes a potentially non-terminating computation.

9.5 Recursion with lists

The same kind of technique that can be used to define recursive functions with integers can also be used to define recursive functions on lists. As an example, suppose that we wish to define a function to calculate the length of a list. As the standard prelude already includes such a function called `length`, we will call the function developed here `len` to avoid any conflict. Now suppose that we wish to find the length of a given list. There are two cases to consider:

- If the list is empty then it has length 0
- Otherwise, it is non-empty and can be written in the form `(x:xs)` for some element `x` and some list `xs`. Thus the original list is one element longer than `xs`, and so has length `1+len xs`.

Writing these two cases out leads directly to the following definition:

```
len []      =  0
len (x:xs)  =  1 + len xs
```

The following diagram illustrates the way that this function can be used to determine the length of the list `[1,2,3,4]` (remember that this is just an abbreviation for `1:2:3:4:[]`):

```
len [1,2,3,4]  ==>  1 + len [2,3,4]
               ==>  1 + (1 + len [3,4])
               ==>  1 + (1 + (1 + len [4]))
               ==>  1 + (1 + (1 + (1 + len [])))
               ==>  1 + (1 + (1 + (1 + 0)))
               ==>  1 + (1 + (1 + 1))
               ==>  1 + (1 + 2)
               ==>  1 + 3
               ==>  4
```

As further examples, you might like to look at the following definitions which use similar ideas to define the functions product and map introduced in earlier sections:

```
product []     = 1
product (x:xs) = x * product xs
map f []      =  []
map f (x:xs)  =  f x : map f xs
```

9.6 Lazy evaluation

Gofer evaluates expressions using a technique sometimes described as 'lazy evaluation' which means that:

- No expression is evaluated until its value is needed.
- No shared expression is evaluated more than once; if the expression is ever evaluated then the result is shared between all those places in which it is used.

The first of these ideas is illustrated by the following function:

```
ignoreArgument x = "I didn't need to evaluate x"
```

Since the result of the function `ignoreArgument` doesn't depend on the value of its argument `x`, that argument will not be evaluated:

```
? ignoreArgument (1/0)
I didn't need to evaluate x
(1 reduction, 31 cells)
?
```

In some situations, it is useful to be able to force Gofer to evaluate the argument to a function before the function is applied. This can be achieved using the function `strict` defined in the standard prelude; An expression of the form `strict f x` is evaluated by first evaluating the argument `x` and then applying the function `f` to the result:

```
? strict ignoreArgument (1/0)
{primDivInt 1 0}
(4 reductions, 29 cells)
?
```

The second basic idea behind lazy evaluation is that no shared expression should be evaluated more than once. For example, the following two expressions can be used to calculate $3 * 3 * 3 * 3$:

```
? square * square where square = 3 * 3
81
(3 reductions, 9 cells)
? (3 * 3) * (3 * 3)
81
(4 reductions, 11 cells)
?
```

Notice that the first expression requires one less reduction than the second. Excluding the single reduction step needed to convert each integer into a string, the sequences of reductions that will be used in each case are as follows:

```
square * square where square = 3 * 3
   -- calculate the value of square by reducing 3 * 3 ==> 9
   -- and replace each occurrence of square with this result
   ==> 9 * 9
   ==> 81

(3 * 3) * (3 * 3)   -- evaluate first (3 * 3)
   ==> 9 * (3 * 3)  -- evaluate second (3 * 3)
   ==> 9 * 9
   ==> 81
```

Lazy evaluation is a very powerful feature of programming in a language like Gofer, and means that only the minimum amount of calculation is used to determine the result of an expression. The following example is often used to illustrate this point.

Consider the task of finding the smallest element of a list of integers. The standard prelude includes a function `minimum` which can be used for this very purpose:

```
? minimum [100,99..1]
1
(809 reductions, 1322 cells)
?
```

(The expression `[100,99..1]` denotes the list of integers from 1 to 100 arranged in decreasing order, as described in section 10.1).

A rather different approach involves sorting the elements of the list into increasing order (using the function `sort` defined in the standard prelude) and then take the element at the head of the resulting list (using the standard function `head`). Of course, sorting the list in its entirety is likely to require significantly more work than the previous approach:

```
? sort [100,99..1]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, ... etc ..., 99, 100]
(10712 reductions, 21519 cells)
?
```

However, thanks to lazy-evaluation, calculating just the first element of the sorted list actually requires less work in this particular case than the first solution using `minimum`:

```
? head (sort [100,99..1])
1
(713 reductions, 1227 cells)
?
```

Incidentally, it is probably worth pointing out that this example depends rather heavily on the particular algorithm used to `sort` a list of elements. The results are rather different if we compare the same two approaches used to calculate the maximum value in the list:

```
? maximum [100,99..1]
100
(812 reductions, 1225 cells)
? last (sort [100,99..1])
100
(10612 reductions, 20732 cells)
?
```

This difference is caused by the fact that each element in the list produced by `sort` is only known once the values of all of the preceding elements are also known. Thus the complete list must be sorted in order to obtain the last element.

9.7 Infinite data structures

One particular benefit of lazy evaluation is that it makes it possible for functions in Gofer to manipulate 'infinite' data structures. Obviously we cannot hope either to construct or store an infinite object in its entirety – the advantage of lazy evaluation is that it allows us to construct infinite objects piece by piece as necessary (and to reuse the storage space used by parts of the object when they are no longer required).

As a simple example, consider the following function which can be used to produce infinite lists of integer values:

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

If we evaluate the expression `countFrom 1`, Gofer just prints the list of integer values beginning with 1 until it is interrupted. Once each element in the list has been printed, the storage used to hold that element can be reused to hold later elements in the list. Evaluating this expression is equivalent to using an 'infinite' loop to print the list of integers in an imperative programming language:

```
? countFrom 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,^C{Interrupted!}
(53 reductions, 160 cells)
?
```

For practical applications, we are usually only interested in using a finite portion of an infinite data structure (just as loops in an imperative programming language are usually terminated after finitely many iterations). For example, using `countFrom` together with the function `take` defined in the standard prelude, we can repeat the calculation from section 4 to find the sum of the integers 1 to 10:

```
? sum (take 10 (countFrom 1))
55
(62 reductions, 119 cells)
?
```

The expression `take n xs` evaluates to a list containing the first `n` elements of the list `xs` (or to `xs` itself if the list contains fewer than n elements). Thus `countFrom 1` generates the infinite list of integers, `take 10` ensures that only the first ten elements are calculated, and `sum` calculates the sum of those integers as before.

A particular advantage of using infinite data structures is that it enables us to describe an object without being tied to one particular application of that object. Consider the following definition for the infinite list of powers of two [1, 2, 4, 8, ...]:

```
powersOfTwo = 1 : map double powersOfTwo
              where double n = 2*n
```

This list be used in a variety of ways; using the operator `(!!)` defined in the standard prelude (`xs!!n` evaluates to the `n`th element of the list `xs`), we can define a function to find the nth power of 2 for any given integer n:

```
twoToThe n = powersOfTwo !! n
```

Alternatively, we can use the list `powersOfTwo` to define a function mapping lists of bits (represented by integers 0 and 1) to the corresponding decimal number: simply reverse the order of the digits, multiply each by the corresponding power of two and calculate the sum. Using functions from the standard prelude, this translates directly into the definition:

```
binToDec ds = sum (zipWith (*) (reverse ds) powersOfTwo)
```

For example:

```
? twoToThe 12
4096
(15 reductions, 21 cells)
? binToDec [1,0,1,1,0]
22
(40 reductions, 85 cells)
?
```

9.8 Polymorphism

Given the definition of `product` in section 9.5, it is easy to see that product takes a single argument which is a list of integers and returns a single integer value – the product of the elements of the list. In other words, `product` has type `[Int] -> Int`. On the other hand, it is not immediately clear what the type of the function `map` should be. Clearly the first argument of `map` must be a function and both the second argument and the result are lists, so that the type of `map` must be of the form:

$$\underbrace{(a \to b)}_{\text{type of argument } \texttt{f}} \quad \to \quad \underbrace{[c]}_{\text{type of argument } \texttt{xs}} \quad \to \quad \underbrace{[d]}_{\text{type of result } \texttt{map f xs}}$$

But what can be said about the types a, b, c and d? One possibility would be to choose $a = b = c = d =$ `Int` which would be acceptable for expressions such as `map fact [1,2,3,4]`, but this would not be suitable in an expression such as `map chr [65,75,32]` because the `chr` function does not have type `Int -> Int`.

Notice however that the argument type of `f` must be the same as the type of elements in the second argument (i.e. $a = c$) since `f` is applied to each element in that list. Similarly, the result type of `f` must be the same as the type of elements in the result list (i.e. $b = d$) since each element in this list is obtained as a result of applying the function `f` to some value. It is therefore reasonable to treat the `map` function as having any type of the form:

$$(a \to b) \to [a] \to [b]$$

The letters a and b used in this type expression represent arbitrary types and are called type variables. An object whose type includes one or more type variables can be thought of as having many different types and is often described as having a 'polymorphic type' (literally: its type has 'many shapes').

The ability to define and use polymorphic functions in Gofer turns out to be very useful. Here are the types of some of the other polymorphic functions which have been used in previous examples which illustrate this point:

```
length :: [a] -> Int
(++)   :: [a] -> [a] -> [a]
concat :: [[a]] -> [a]
```

Thus we can use precisely the same `length` function to determine both the length of a list of integers as well as finding the length of a string:

```
? length [1..10]
10
(98 reductions, 138 cells)
? length "Hello"
5
(22 reductions, 36 cells)
?
```

9.9 Higher-order functions

In Gofer, function values are treated in much the same way as any other kind of value; in particular, they can be used both as arguments to, and results of other functions.

Functions which manipulate other functions in this way are often described as 'higher-order functions'. Consider the following example, taken from the standard prelude:

```
(.)       :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x  = f (g x)
```

As indicated by the type declaration, we think of the `(.)` operator as a function taking two function arguments and returning another function value as its result. If `f` and `g` are functions of the appropriate types, then `(f . g)` is a function called the composition of `f` with `g`. Applying `(f . g)` to a value is equivalent to applying `g` to that value, and then applying `f` to the result (as described, far more eloquently, by the second line of the declaration above!).

Many problems can often be described very elegantly as a composition of other functions. Consider the problem of calculating the total number of characters used in a list of strings. A simple recursive function provides one solution:

```
countChars []     = 0
countChars (w:ws) = length w + countChars ws

? countChars ["super","cali","fragi","listic"]
20
(96 reductions, 152 cells)
?
```

An alternative approach is to notice that we can calculate the total number of characters by first combining all of the words in the argument list into a single word (using concat) and then finding the length of that word:

```
? (length . concat) ["super","cali","fragi","listic"]
20
(113 reductions, 211 cells)
?
```

Another solution is to first find the length of each word in the list (using the `map` function to apply `length` to each word) and then calculate the sum of these individual lengths:

```
? (sum . map length) ["super","cali","fragi","listic"]
20
(105 reductions, 172 cells)
?
```

9.10 Variable declarations

A variable declaration is a special form of function definition, almost always consisting of a single equation of the form:

var = *rhs*

(i.e. a function declaration of arity 0). Whereas the values defined by function declarations of arity > 0 are guaranteed to be functions, the values defined by variable declarations may or may not be functions:

```
odd = not . even   -- if an integer is not even then it must be odd
val = sum [1..100]
```

Note that variables defined like this at the top level of a file of definitions will be evaluated using lazy evaluation. The first time we refer to the variable `val` defined above (either directly or indirectly), Gofer evaluates the sum of the integers from 1 to 100 and overwrites the definition of `val` with this number. This calculation can then be avoided for each subsequent use of `val` (unless the file containing the definition of `val` is reloaded).

```
? val
5050
(809 reductions, 1120 cells)

? val
5050
(1 reduction, 7 cells)

?
```

Because of this behaviour, we should probably try to avoid using variable declarations where the resulting value will require a lot of storage space. If we load a file of definitions including the line:

```
longList = [1..10000]
```

and then evaluate the expression `length longList` (eventually obtaining the expected result of 10000), then Gofer will evaluate the definition of `longList` and replace it with the complete list of integers from 1 upto 10000. Unlike other memory used during a calculation, it will not be possible to reuse this space for other calculations without reloading the file defining `longList`, or loading other files instead.

9.11 Pattern bindings and irrefutable patterns

Another useful way of defining variables uses 'pattern bindings' which are equations of the form:

pat = *rhs*

where the expression on the left hand side is a pattern as described in section 9.1. As a simple example of pattern bindings, here is one possible definition for the function `head` which returns the first element in a list of values:

```
head xs  =  x  where  (x:ys) = xs
```

The definition `head (x:_) = x` used in the standard prelude is slightly more efficient, but otherwise equivalent.

Note that pattern bindings are treated quite differently from function bindings (of which the variable declarations described in the last section are a special case). There are two situations in which an ambiguity may occur; i.e. if the left hand side of an equation is a simple variable or an `(n+k)` pattern of the kind described in section 9.1. In both cases, these are treated as function bindings, the former being a variable declaration whilst the latter will be treated as a definition for the operator symbol `(+)`.

Pattern bindings are often useful for defining functions which we might think of as 'returning more than one value' – although they are actually packaged up in a single value such as a tuple. As an example, consider the function `span` defined in the standard prelude.

```
span :: (a -> Bool) -> [a] -> ([a],[a])
```

If `xs` is a list of values and `p` is a predicate, then `span p xs` returns the pair of lists `(ys,zs)` such that `ys++zs == xs`, all of the elements in `ys` satisfy the predicate `p` and the first element of `zs` does not satisfy `p`. A suitable definition, using a pattern binding to obtain the two lists resulting from the recursive call to `span` is as follows:

```
span p []              = ([],[])
span p xs@(x:xs')
           | p x       = let (ys,zs) = span p xs' in (x:ys,zs)
           | otherwise = ([],xs)
```

For consistency with the lazy evaluation strategy used in Gofer, the right hand side of a pattern binding is not evaluated until the value of one of the variables bound by that pattern is required. The definition:

```
(0:xs) = [1,2,3]
```

will not cause any errors when it is loaded into Gofer, but will cause an error if we attempt to evaluate the variable `xs`:

```
? xs
{v120 [1, 2, 3]}
(11 reductions, 46 cells)
?
```

The variable name `v120` appearing in this expression is the name of a function called a 'conformality check' which is defined automatically by Gofer to ensure that the value on the right hand side of the pattern binding conforms with the pattern on the left.

Compare this with the behaviour of pattern matching in function definitions such as:

```
? example [1] where example (0:xs) = "Hello"
{v126 [1]}
(4 reductions, 22 cells)
?
```

where the equivalent of the conformality check is carried out immediately even if none of the values of the variables in the pattern are actually required. The reason for this difference is that the arguments supplied to a function must be evaluated to determine which equation in the definition of the function should be used. The error produced by the example above was caused by the fact that the argument [1] does not match the pattern used in the equation defining `example` (represented by an internal Gofer function called `v126`).

A different kind of behaviour can be obtained using a pattern of the form `~pat`, known as an irrefutable (or lazy) pattern. This pattern can initially be matched against any value, delaying the check that this value does indeed match `pat` until the value of one of the variables appearing in it is required. The basic idea (together with the method used to implement irrefutable patterns in Gofer) is illustrated by the identity:

`f ~`*pat* `=` *rhs* is equivalent to `f v =` *rhs* `where` *pat* `= v`

The following examples, based very closely on those given in the Haskell report [5], illustrate the use of irrefutable patterns. The variable `undefined` used in these examples is included in the standard prelude and causes a run-time error each time it is evaluated (technically speaking, it represents the $\perp$ element of the relevant semantic domain, and is the only value having all possible types):

```
(\ (x,y) -> 0) undefined = {undefined}
(\~(x,y) -> 0) undefined = 0

(\ [x] -> 0) [] = {v113 []}
(\~[x] -> 0) [] = 0

(\~[x, (a,b)] -> x) [(0,1),undefined] = {undefined}
```

```
(\~[x,~(a,b)] -> x) [(0,1),undefined] = (0,1)

(\ (x:xs) -> x:x:xs) undefined = {undefined}
(\~(x:xs) -> x:x:xs) undefined = {undefined}:{undefined}:{undefined}
```

Irrefutable patterns are not used very frequently, although they are particularly convenient in some situations (see section 12 for some examples). Be careful not to use irrefutable patterns where they are not appropriate. An attempt to define a map function `map'` using:

```
map' f ~(x:xs) = f x : map' f xs
map' f []      = []
```

turns out to be equivalent to the definition:

```
map' f ys  =  f x : map f xs where (x:xs) = ys
```

and will not behave as you might have intended:

```
? map' ord "abc"
[97, 98, 99, {v124 []}, {v124 []}, {v^C{Interrupted!}
(35 reductions, 159 cells)
?
```

9.12 Type declarations

The type system used in Gofer is sufficiently powerful to enable Gofer to determine the type of any function without the need to declare the types of its arguments and the return value as in some programming languages. Despite this, Gofer allows the use of type declarations of the form:

$var_1, \ldots var_n$ `::` *type*

which enable the programmer to declare the intended types of the variables $var_1, \ldots var_n$ defined in either function or pattern bindings. There are a number of benefits of including type declarations of this kind in a program:

- Documentation: The type of a function often provides useful information about the way in which a function is to be used – including the number and order of its arguments.
- Restriction: In some situations, the type of a function inferred by Gofer is more general than is required. As an example, consider the following function, intended to act as the identity on integer values:

  ```
  idInt x  =  x
  ```

  Without an explicit type declaration, Gofer treats `idInt` as a polymorphic function of type `a -> a` and the expression `idInt 'A'` does not cause a type error. This problem can be solved by using an explicit type declaration to restrict the type of `idInt` to a particular instance of the polymorphic type `a -> a`:

  ```
  idInt :: Int -> Int
  ```

  Note that a declaration such as:

  ```
  idInt :: Int -> a
  ```

  is not a valid type for the function `idInt` (the value of the expression `idInt 42` is an integer and cannot be treated as having an arbitrary type, depending on the value of the type variable `a`), and hence will not be accepted by Gofer.
- Consistency check: As illustrated above, declared types are always checked against the definition of a value to make sure that they are compatible. Thus Gofer can be used to check that the programmer's intentions (as described by the types assigned to variables in type declarations) are consistent with the definitions of those values.
- Overloading: Explicit type declarations can be used to solve a number of problems associated with overloaded functions and values. See section 14 for further details.

Chapter 10

Increasing your power of expression

This section describes a number of useful extensions to the basic range of expressions used in the previous sections. None of these add any extra computational power to Gofer – anything that can be done with these constructs could also be done with the constructs already described. They are however included in Gofer because they allow many expressions and function definitions to be written more clearly and concisely than the equivalent expressions without these notations.

10.1 Arithmetic sequences

A number of useful lists can be generated using the notation of arithmetic sequences (so named because of their similarity to arithmetic progressions in mathematics). The following list summarises the four forms of sequence expression that can be used in Gofer, together with their translation using the standard functions `enumFrom`, `enumFromTo`, `enumFromThen` and `enumFromThenTo`:

`[ n .. ]`	`enumFrom n`	Produces the (potentially infinite) list of values starting with the value of `n` and increasing in single steps.
`[ n .. m ]`	`enumFromTo n m`	Produces the list of elements from `n` upto and including `m` in single steps. If `m` is less than `n` then the list is empty.
`[ n, m .. ]`	`enumFromThen n m`	Produces the (potentially infinite) list of values whose first two elements are given by the values `n` and `m`. If `m` is greater than `n` then the following elements of the list are increasing in steps of the same size. A similar result is obtained if `m` is less than `n` in which case the elements of `[n,m..]` will be decreasing. If `n` and `m` are equal then `[n,m..]` is an infinite list in which each element is equal to `n`.
`[ n, n' .. m ]`	`enumFromThenTo n n' m`	Produces the list of elements from `[n,n'..]` upto the limit value `m`. If `m` is less than `n` and `[n,n'..]` is increasing, or `m` is greater than `n` and `[n,n'..]` is decreasing the resulting list will be empty.

Examples:

```
[1..]       =  [1, 2, 3, 4, 5, 6, 7, 8, 9, etc...
[-3..3]     =  [-3, -2, -1, 0, 1, 2, 3]
[1..1]      =  [1]
[9..0]      =  []
[1,3..]     =  [1, 3, 5, 7, 9, 11, 13, etc...
[0,0..]     =  [0, 0, 0, 0, 0, 0, 0, etc...
[5,4..]     =  [5, 4, 3, 2, 1, 0, -1, etc...
[1,3..12]   =  [1, 3, 5, 7, 9, 11]
[0,0..10]   =  [0, 0, 0, 0, 0, 0, 0, etc...
[5,4..1]    =  [5, 4, 3, 2, 1]
```

In the standard prelude, the functions `enumFrom`, `enumFromTo`, `enumFromThen` and `enumFromThenTo` are overloaded and may also be used to enumerate lists of characters or floating point values:

```
? ['0'..'9'] ++ ['A'..'Z']
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
(397 reductions, 542 cells)

? [1.2, 1.35 .. 2.00]
[1.2, 1.35, 1.5, 1.65, 1.8, 1.95]
(56 reductions, 133 cells)
```

Arithmetic sequences such as those described above play the same role in functional programming languages as the iterative 'for' constructs in traditional imperative languages. A good example of this is the example in section 4 used to calculate the sum of the integers from 1 upto 10 – `sum [1..10]`. An equivalent program in an imperative language might look something like (especially if you think of C!):

```
int i;
int total=0;
for (i=1; i<=10; i++)
    total = total + i;
return total;
```

The advantages of the functional notation in this case are clear:
- It is more compact.
- It separates the task of generating the sequence of integers `[1..10]` from the task of finding their sum.
- It does not require the declaration or use of auxiliary variables such as `i` and `total` in the above.

10.2 List comprehensions

List comprehensions provide another very powerful and compact notation for describing certain kinds of list expression. The basic form of a list comprehension is:

`[` *expr* `|` *qualifiers* `]`

There are three kinds of qualifier that can be used in Gofer:

- Generators: A qualifier of the form *pat* `<-` *exp* is used to extract each element that matches the pattern pat from the list exp in the order that they elements appear in that list. A simple example of this is the expression `[x*x | x<-[1..10]]` which denotes the list of the squares of the integers between 1 and 10 inclusive and evaluates to [1, 4, 9, 16, 25, 36, 49, 64, 81, 100] as expected.

  Formally, we can define the meaning of a list comprehension with a single generator by the equation:

  ```
  [ e | pat <- exp ]  =  loop exp
                         where loop []       = []
                               loop (pat:xs) = e : loop xs
                               loop (_:xs)   = loop xs
  ```

  If `pat` is an irrefutable pattern (for example, a variable) then this is equivalent to:

```
[ e | pat <- exp ]  =  map f exp
                       where f pat = e
```

The full definition is needed for those cases where the pattern pat may not match all of the elements in the list exp. This is the case in expressions such as `[ y | (3,y)<-[(1,2),(3,4),(5,6)] ]` which evaluates to the singleton list `[4]`.

- Filters: A boolean valued expression may also be used as a qualifier in which case it is often called a filter. We can define the meaning of a list comprehension with a single filter by the equation:

  ```
  [ e | condition ]  =  if condition then [e] else []
  ```

  Whilst this form of list comprehension is occasionally useful as it stands, it is more common to use filters in conjunction with generators as described below.

- Local definitions: A qualifier of the form *pat* = *expr* can be used to introduce a local definition within a list comprehension. Its meaning can be defined formally using the equation:

  ```
  [ e | pat = exp ]  =  [ let pat=exp in e ]
  ```

  As in the case of filters, local definitions are more commonly used within lists of more than one qualifier as described below. Particular care should be taken to distinguish a filter of the form *pat*==*expr* from a local definition of the form *pat*=*expr*.

  (I originally suggested this form of qualifier in a message sent to the Haskell mailing list, only to discover that a similar (and more comprehensive) suggestion had been made by Kevin Hammond almost a year earlier. There was a certain amount of controversy surrounding the choice of an appropriate syntax and semantics for the construct and consequently, this feature is not currently part of the Haskell standard. The syntax and semantics above is implemented by Gofer in the hope that it will give functional programmers an opportunity to experiment with this facility in their own programs.)

The real power of this notation is that it is possible to use several qualifiers, separated by commas on the right of the vertical bar '`|`' symbol in a list comprehension. Formally, if `qs1` and `qs2` are two such lists of qualifiers, then we can define the meaning of multiple qualifiers using:

```
[ e | qs1, qs2 ]  =  concat [ [ e | qs2 ] | qs1 ]
```

The following examples illustrate how this definition works in practice:

- Variables generated by later qualifiers vary more quickly than those generated by earlier qualifiers:

  ```
  ? [ (x,y) | x<-[1..3], y<-[1..2] ]
  [(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)]
  (107 reductions, 246 cells)
  ?
  ```

- Later qualifiers may use the values generated by earlier ones:

  ```
  ? [ (x,y) | x<-[1..3], y<-[1..x]]
  [(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]
  (107 reductions, 246 cells)

  ? [ x | x<-[1..10], even x ]
  [2, 4, 6, 8, 10]
  (108 reductions, 171 cells)
  ?
  ```

- Variables defined in later qualifiers hide those introduced by earlier ones. The following expressions are valid list comprehensions, but this style of definition in which names are reused can result in programs which are difficult to understand, and is not recommended:

```
? [ x | x<-[[1,2],[3,4]], x<-x ]
[1, 2, 3, 4]
(18 reductions, 53 cells)

? [ x | x<-[1,2], x<-[3,4] ]
[3, 4, 3, 4]
(18 reductions, 53 cells)
?
```

- Changing the order of qualifiers has a direct effect on efficiency. The following two examples produce the same result, but the first uses more reductions and cells because it repeats the evaluation of `even x` for each possible value of `y`.

```
? [ (x,y) | x<-[1..3], y<-[1..2], even x ]
[(2,1), (2,2)]
(110 reductions, 186 cells)
? [ (x,y) | x<-[1..3], even x, y<-[1..2] ]
[(2,1), (2,2)]
(62 reductions, 118 cells)
?
```

The following example illustrates a similar kind of behaviour with local definitions; in the first case the expression `fact x` is evaluated twice for each possible value of `x`, whilst the second expression uses a local definition to ensure that the evaluation is not repeated:

```
? [ fact x + y | x<-[1..3], y<-[1..2] ]
[2, 3, 3, 4, 7, 8]
(246 reductions, 398 cells)

? [ factx + y | x<-[1..3], factx = fact x, y<-[1..2] ]
[2, 3, 3, 4, 7, 8]
(173 reductions, 294 cells)
?
```

10.3 Lambda expressions

In addition to named function definitions, Gofer also allows the definition and use of unnamed functions using a 'lambda expression' of the form:

\ *atomicPatterns* -> *expr*

This is a slight generalisation of the form of lambda expression used in most theoretical treatments of functional programming and dating back to the pioneering work of logicians including Alonzo Church and Haskell Curry, from whom the programming language takes its name. The '\' character used at the beginning of a Gofer lambda expression has been chosen for its resemblance to the greek letter λ (lambda) that might be used if the standard character set were a little larger.

This expression denotes a function taking a number of parameters (one for each pattern) and producing the result specified by the expression to the right of the -> symbol. For example, `(\x->x*x)` represents the function which takes a single integer argument `x` and produces the square of that number as its result. Another example is the lambda expression `(\x y->x+y)` which takes two integer arguments and outputs their sum; this expression is in fact equivalent to the `(+)` operator:

```
? (\x y->x+y) 2 3
5
(3 reductions, 7 cells)
?
```

A lambda expression of the form illustrated above is equivalent to the following expression using a local definition:

```
(let newName <atomic patterns> = <expr> in newName)
```

where `newName` is a new variable name, chosen to avoid conflicts with other variables that are already in use. This name will be printed if you enter an expression involving a lambda expression without supplying the full number of parameters for that function:

```
? (\x y -> x+y) 42
v117 42
(2 reductions, 14 cells)
?
```

Lambda expressions are particularly useful for certain styles of functional programming; an example of this is the continuation-based approach to I/O described in section 12.

10.4 Case expressions

A case expression can be used to evaluate an expression and, depending on the result, return one of a number of possible values. As such, case statements are a straightforward generalisation of conditional expressions. Indeed, an expression of the form `if e then t else f` is in fact equivalent to the case expression:

```
case e of
  True  -> t
  False -> f
```

In general, a case expression takes the form `case exp of alts` where `exp` is the expression to be evaluated and `alts` is a list of alternatives, each of which is of the form:

```
pat -> rhs
```

for a simple alternative

```
pat | condition1 -> rhs1
    | condition2 -> rhs2
          .
          .
    | conditionn -> rhsn
```

using guard expressions as described in section 9.2 for function definitions.

In Gofer, a case expression of the form `case e of alts` is implemented by choosing a new function name `newName` as in the previous section and using the alternatives in alts to construct an appropriate definition for this function (essentially by replacing each '`->`' symbol with a '`=`' symbol). The complete case expression is then treated as being equivalent to the expression `newName e`. A simple example of this is the `scanl` function whose definition in the standard prelude:

```
scanl f q xs = q : (case xs of
                     []   -> []
                     x:xs -> scanl f (f q x) xs)
```

is equivalent to:

```
scanl f q xs = q : scanl' xs
               where scanl' []     = []
                     scanl' (x:xs) = scanl f (f q x) xs
```

This latter form is precisely the definition used in [1] (but using the name `scan` where Gofer uses `scanl`).

Evaluating a case expression in which none of the alternatives match the value of the discriminant results in an error such as the following:

```
? case [1,2] of [] -> "empty list"
{v117 [1, 2]}
(6 reductions, 31 cells)
?
```

The function name `v117` which appears here is the name of the function which is used internally by Gofer to implement the case expression whilst the expression `[1, 2]` gives the discriminant value which could not be matched.

By combining case expressions with the lambda expressions introduced in the previous section, any function declaration can be translated into a single equation of the form `functionName = expr`. For example, the standard function `map` whose definition is usually written as:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

can also be defined by the equation:

```
map = \f xs -> case xs of
                 []     -> []
                 (y:ys) -> f y : map f ys
```

This kind of translation is used in the implementation of many functional programming languages, including Gofer. See Simon Peyton Jones book [2] for more details of this.

10.5 Operator sections

As we have seen, most functions in Gofer taking more than one argument are treated as a function of a single argument, whose result is a function which can then be applied to the remaining arguments. Thus `(+) 1` denotes the function which takes an integer argument `n` and returns the integer value `1+n`. Functions of this kind involving operator symbols are sufficiently common that Gofer provides a special syntax for them. Using e to denote an atomic expression and the symbol $\oplus$ to represent an arbitrary infix operator, there are functions $(e\oplus)$ and $(\oplus e)$, known as 'sections of the operator $(\oplus)$' defined by:

```
(e *) x  = e * x
(* e) x  = x * e
```

or, using lambda expressions as introduced in section 10.3:

```
(e *)    =  \x -> e * x
(* e)    =  \x -> x * e
```

For example:

`(1+)`	is the successor function which returns the value of its argument plus 1,
`(1.0/)`	is the reciprocal function,
`(/2)`	is the halving function,
`(:[])`	is the function which maps any value to the singleton list containing that element.

In Gofer, the expressions `(e *)` and `(* e)` are actually treated as abbreviations for `(*) e` and `flip (*) e` respectively, where `flip` is the function defined by:

```
flip        :: (a -> b -> c) -> b -> a -> c
flip  f x y  =  f y x
```

There is an important special case which occurs with an expression of the form $(-e)$; this is interpreted as `negate e` and not as the section which subtracts the value of `e` from its argument. The latter function can be written as the section `(+ (- e))` or as `subtract e` where `subtract` is the function defined in the standard prelude using:

```
subtract = flip (-)
```

10.6 Explicitly typed expressions

As described in section 9.12, it is often useful to be able to declare the type of a variable defined in a function or pattern binding. For much the same reasons, Gofer allows expressions of the form:

```
expr :: type
```

so that the type of an expression can be specified explicitly. Note that the `:t` command can be used to find the type of a particular expression that is inferred by Gofer:

```
? :t  \x -> [x]
\x -> [x] :: a -> [a]

? :t  sum . map length
sum . map length :: [[a]] -> Int
```

The types inferred in each case can be modified by including explicit types in these expressions:

```
? :t  (\x -> [x]) :: Char -> String
\x -> [x] :: Char -> String

? :t  sum . map (length :: String -> Int)
sum . map length :: [String] -> Int
```

Note that an error occurs if the type declared in an explicitly typed expression is not compatible with the type inferred by Gofer:

```
? :t (\x -> [x]) :: Int -> a
ERROR: Declared type too general
*** Expression    : \x -> [x]
*** Declared type : Int -> a
*** Inferred type : Int -> [Int]
```

Explicitly typed expressions are most commonly used together with overloaded functions and values as described in section 14.

Chapter 11

User-defined datatypes and type synonyms

11.1 Datatype definitions

In addition to the wide range of built-in datatypes described in section 7, Gofer also allows the definition of new datatypes using declarations of the form:

`data` *Datatype* $a_1 \ldots a_n$ `=` $constr_1$ `|` ... `|` $constr_m$

where *Datatype* is the name of a new type constructor of arity $n \geq 0$, $a_1, \ldots, a_n$ are distinct type variables representing the arguments of *DatatypeName* and $constr_1, \ldots, constr_m$ ($m \geq 1$) describe the way in which elements of the new datatype are constructed. Each *constr* can take one of two forms:

- *Name* $type_1 \ldots type_r$ where *Name* is a previously unused constructor function name (i.e. an identifier beginning with a capital letter). This declaration introduces *Name* as a new constructor function of type:

  $type_1 \rightarrow \ldots \rightarrow type_r \rightarrow Datatype\ a_1 \ldots a_n$

- $type_1 \oplus type_2$ where $\oplus$ is a previously unused constructor function operator (i.e. an operator symbol beginning with a colon). This declaration introduces $(\oplus)$ as a new constructor function of type:

  $type_1 \rightarrow type_2 \rightarrow Datatype\ a_1 \ldots a_n$

Only the type variables $a_1, \ldots, a_n$ may appear in the type expressions in each *constr* in the definition of *Datatype*.

As a simple example, the following definition introduces a new type `Day` with elements `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri` and `Sat`:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

Simple functions manipulating elements of type `Day` can be defined using pattern matching:

```
what_shall_I_do Sun = "relax"
what_shall_I_do Sat = "go shopping"
what_shall_I_do _   = "looks like I'll have to go to work"
```

Another example uses a pair of constructors to provide a representation for temperatures which may be given using either of the centigrade or fahrenheit scales:

```
data Temp = Centigrade Float | Fahrenheit Float

freezing                  :: Temp -> Bool
freezing (Centigrade temp) = temp <= 0.0
freezing (Fahrenheit temp) = temp <= 32.0
```

The following example uses a type variable on the left hand side of the datatype definition to implement a `Set` type constructor for representing sets using a list of values:

```
data Set a = Set [a]
```

For example, `Set [1,2,3]` is an element of type `Set Int`, representing the set of integers $\{1, 2, 3\}$ whilst `Set ['a']` represents a singleton set of type `Set Char`. As this example shows, it is possible to use the same name simultaneously as both a type constructor and as a constructor function.

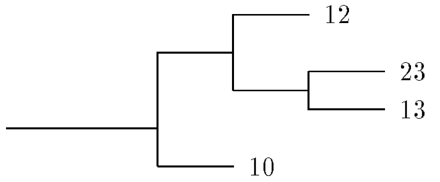
Datatype definitions may also be recursive, using the name of the datatype being defined on the right hand side of the datatype definition (mutually recursive datatype definitions are also permitted). The following example is taken from the Haskell report [5] and defines a type representing binary trees with values of a particular type at their leaves:

```
data Tree a = Lf a | Tree a :^: Tree a
```

For example,

```
(Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10
```

has type `Tree Int` and represents the binary tree:



As an example of a function defined on trees, here are two definitions using recursion and pattern matching on tree valued expressions which calculate the list of elements at the leaves of a tree traversing the branches of the tree from left to right. The first definition uses a simple definition, whilst the second uses an 'accumulating parameter' giving a more efficient algorithm:

```
leaves, leaves'  :: Tree a -> [a]

leaves (Lf l)  = [l]
leaves (l:^:r) = leaves l ++ leaves r

leaves' t = leavesAcc t []
            where leavesAcc (Lf l)  = (l:)
                  leavesAcc (l:^:r) = leavesAcc l . leavesAcc r
```

Using the binary tree above as an example:

```
? leaves ((Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10)
[12, 23, 13, 10]
(24 reductions, 73 cells)
? leaves' ((Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10)
[12, 23, 13, 10]
(20 reductions, 58 cells)
?
```

11.2 Type synonyms

Type synonyms are used to provide convenient abbreviations for type expressions. A type synonym is introduced by a declaration of the form:

`type` *Name* $a_1 \ldots a_n$ = *expansion*

where *Name* is the name of a new type constructor of arity $n \geq 0$, $a_1, \ldots, a_n$ are distinct type variables representing the arguments of *Name* and *expansion* is a type expression. Note that the only type variables permitted in the expansion type are those on the left hand side of the synonym definition. Using this declaration any type expression of the form:

$Name\ type_1 \ldots type_n$

is treated as an abbreviation of the type expression obtained from expansion by replacing each of the type variables $a_1, \ldots, a_n$ with the corresponding type $type_1, \ldots, type_n$.

The most frequently used type synonym is almost certainly the `String` type which is a synonym for `[Char]`:

```
type String = [Char]
```

(This definition is actually built in to the Gofer system, but the effect is the same as if this declaration were included in the standard prelude.)

Note that the types of expressions inferred by Gofer will not usually contain any type synonyms unless an explicit type signature is given, either using an explicitly typed expression (section 10.6) or a type declaration (section 9.12):

```
? :t ['c']
['c'] :: [Char]
? :t ['c'] :: String
['c'] :: String
?
```

Unlike the datatype declarations described in the previous section, recursive (and mutually recursive) synonym declarations are not permitted. This rules out examples such as:

```
type BadSynonym = [BadSynonym]
```

and ensures that the process of expanding all of the type synonyms used in any particular type expression will always terminate. The same property does not hold for the illegal definition above, in which any attempt to expand the type BadSynonym would lead to the non-terminating sequence:

```
BadSynonym ==> [BadSynonym] ==> [[BadSynonym]] ==> ....
```

Chapter 12

Dialogues: input and output

The Gofer system implements a subset of the facilities for programs involving I/O described in the Haskell report [5]. In particular, this makes it possible for Gofer programs to be run interactively, and to make limited use of text files for both reading and writing. A significant factor in the design of the Haskell I/O facilities is that it allows the use of such programs without loss of referential transparency.

12.1 Basic description

Programs using the I/O facilities in Gofer are modelled by functions of type Dialogue, defined by the type synonym:

```
type Dialogue    =  [Response] -> [Request]
```

In other words, a Gofer program produces a list of output values, each of which may be thought of as a request for some particular input or output action, and obtains the corresponding list of operating system responses as its input. Note that the input list of responses will be evaluated lazily; i.e. we can ensure that we do not attempt to obtain the response to a given request until that request has been completed.

The current range of requests supported by Gofer is described by the following datatype definition, taken from the standard prelude:

```
data Request  =  -- file system requests:
                ReadFile      String
              | WriteFile     String String
              | AppendFile    String String
                 -- channel system requests:
              | ReadChan      String
              | AppendChan    String String
                 -- environment requests:
              | Echo          Bool
```

Each response is an element of the type defined by the following datatype definition, using an auxiliary datatype `IOError` to describe a variety of error conditions that may occur:

```
data Response = Success
              | Str String
              | Failure IOError

data IOError  = WriteError   String
              | ReadError    String
              | SearchError  String
              | FormatError  String
              | OtherError   String
```

The following list describes the kind of I/O behaviour specified by each form of Request and indicates the possible Response values that may be obtained in each case:

- `ReadFile string`: Read contents of file named by `string`. Possible responses to this request are:
  - `Str contents` if the request is successful, where `contents` is a string (evaluated lazily) containing the contents of the file specified by the `ReadFile` request.
  - `Failure (SearchError name)` occurs if file `name` cannot be accessed.
  - `Failure (ReadError name)` occurs if some other error occurs whilst opening the file `name`.
- `WriteFile name string`: Write the given `string` to the file `name`. If the file does not already exist, it is created before attempting to write the value to file. If the file already exists then it will be truncated to zero length before the write begins. No response is obtained until the string argument has been fully evaluated and its contents written to file. Possible responses are:
  - `Success` if the write to file was completed successfully.
  - `Failure (WriteError msg)` if an error was detected whilst trying to perform the output. If the problem occurred whilst attempting to open the specified file, then `msg` contains the filename, otherwise it contains a printable representation of the evaluation error which occurred.
- `AppendFile name string`: Similar to the `WriteFile` request except that the value of the given `string` is appended onto the file `name` if that file already exists. The responses that may be obtained from this request are the same as those for `WriteFile`.
- `ReadChan name`: Read from the input stream `name`. Note that it is an error to attempt to read from the same channel more than once in the same program. Possible responses are:
  - `Str contents` if the request is successful, where `contents` is a string (evaluated lazily) containing the list of characters entered on the input stream.
  - `Failure (SearchError name)` if the named channel cannot be found. The only input channel known to Gofer is the standard input channel `stdin`. For convenience, the standard prelude defines the variable stdin bound to this string.
  - `Failure (ReadError name)` if a `ReadChan` request for the named channel has already been given by a previous request.
- `AppendChan name string`: Output `string` on channel `name`. No response is obtained until the string has been fully evaluated and written to the named channel. Possible responses are:
  - `Success` if the append to channel was completed successfully.
  - `Failure (SearchError name)` if the named channel cannot be found. The only output channels known to Gofer are `stdout`, `stderr` and `stdecho` (which is actually just another name for `stdout` in Gofer). For convenience, the standard prelude defines variables `stdout`, `stderr` and `stdecho` bound to the corresponding string values.
  - `Failure (WriteError msg)` if an error is detected whilst trying to perform the output. The string `msg` contains a printable representation of the evaluation error which occurred.
- `Echo status`: Set the echo status on the standard input channel stdin to the given boolean value. If the echo status is `True`, then user input will be echoed onto the screen as it is typed and the usual line editing facilities (such a backspace or delete) provided by the host system can be used to edit the input lines as they are entered. If the echo status is False, then individual characters may be read from the standard input channel without any echo or line editing features.

  Note that at most one `Echo` request can be used in a program, and must precede any `ReadChan` request for `stdin`. If not set by an explicit `Echo` request, the echo status defaults to `True`. Possible responses are:
  - `Success` if the request was completed successfully.
  - `Failure (OtherError msg)` if the request could not be completed either because a `readChannel` request for `stdin` has already been processed, or because a previous `Echo` request has already been given. The corresponding values of `msg` are `stdin already in use` and `repeated Echo request` respectively.

A simple example of a program using these facilities to output a short message on the standard output stream is:

```
helloWorld      :: Dialogue
helloWorld resps = [AppendChan stdout "hello, world"]
```

Any expression entered into Gofer of type `Dialogue` will be treated as a Gofer program using I/O and will be executed accordingly:

```
? helloWorld
hello, world
(1 reduction, 28 cells)
?
```

Notice that without the explicit type declaration, the type that would be inferred for `helloWorld` would be `a -> [Request]`, and hence `helloWorld` would not be executed as a `Dialogue` program. This point can be illustrated using lambda expressions:

```
? \resps -> [AppendChan stdout "hello, world"]
v128
(1 reduction, 7 cells)
? (\resps -> [AppendChan stdout "hello, world"]) :: Dialogue
hello, world

(1 reduction, 28 cells)
?
```

In many cases the structure of an expression is enough to fully determine its type as `Dialogue` (or equivalently as `[Response] -> [Request]`), in which case no explicit types are required to ensure that the expression is treated as a Gofer program using I/O:

```
? \~[Success] -> [AppendChan stdout "hello, world"]
hello, world
(1 reduction, 29 cells)
?
```

Note the use of the irrefutable pattern `~[Success]` for the lambda expression in the last example; without this, the usual rules of pattern matching as described in section 9 would force Gofer to try to match the pattern `[Success]` against the list of responses, before the corresponding request had been produced:

```
? \ [Success] -> [AppendChan stdout "hello, world"]

Aborting Dialogue:
      {error "Attempt to read response before request complete"}
(50 reductions, 229 cells)
?
```

The next example takes a single string as a parameter and displays the contents of the corresponding file:

```
showFile               :: String -> Dialogue
showFile name ~(read:_) = [ReadFile name, AppendChan stdout result]
 where result = case read of Str contents -> contents
                             Failure _    -> "Can't open " ++ name
```

With a few modifications, we can implement a similar program which prompts for, and reads, a filename from the standard input and then reads and displays the contents of that file as before. This program is based on a similar example in the Haskell report [5]:

```
main ~(Success : ~(Str userInput : ~(r3 : _)))
  = [ AppendChan stdout "Please type a filename: ",
      ReadChan stdin,
      ReadFile name,
      AppendChan stdout (case r3 of Str contents -> contents
                                    Failure _    -> "Can't open "
                                                    ++ name)
    ] where (name : _) = lines userInput
```

12.2 Continuation style I/O

As an alternative to the 'stream-based' approach to programs using the I/O facilities in Gofer, the standard prelude defines a family of functions which enables such programs to be written in a 'continuation' style. The basic idea is to define a function corresponding to each different kind of request, whose parameters include the values required to make the request together with two continuations. The continuations are functions describing 'what to do next', one of which is used if the request is successful, the other if the request fails.

As an example, the `ReadFile` request is represented by the function `readFile` whose definition is equivalent to:

```
readFile name fail succ ~(r:rs) = ReadFile name : rest rs
 where rest = case r of Str s           -> succ s
                        Failure ioerror -> fail ioerror
```

The first thing to happen when a dialogue expression of the form `readFile name fail succ` is evaluated is that the corresponding request `ReadFile name` is added to the list of I/O requests. A new dialogue value `rest` is chosen, depending on the response to the ReadFile request, and the program continues by passing the remaining part of the response list to `rest`. The functions `succ` and `fail` (called the success and failure continuations respectively) describe the way in which the new dialogue `rest` is obtained.

The following example (edited a little to fit within the margins of this document) shows how the readFile function described above can be used to print the contents of a file called `test` on the display:

```
? readFile "test" (\ioerror resps -> [])
                  (\s resps->[AppendChan stdout s])
This is a test message

(4 reductions, 52 cells)
?
```

The success continuation `(\s resps->[AppendChan stdout s])` used here receives the contents of the file `test` in the the parameter `s` and uses an `AppendChan` request to output that string on the display. As this example shows, the stream based approach of the previous section can be combined with the continuation based style of I/O without any difficulty. The failure continuation `(\ioerror resps -> [])` ignores the error condition `ioerror` which caused the request to fail and gives a dialogue which terminates immediately without any action. For example, assuming that the file `Test` cannot be found:

```
? readFile "Test" (\ioerror resps -> [])
                  (\s resps->[AppendChan stdout s])

(4 reductions, 24 cells)
?
```

In practice, it is usually a good idea to produce some kind of diagnostic message when an error occurs:

```
? readFile "Test"
     (\ioerror resps -> [AppendChan stdout (show' ioerror)])
     (\s resps       -> [AppendChan stdout s])
SearchError "Test"
(11 reductions, 59 cells)
?
```

In each of the examples above, the failure continuation has type `FailCont` as defined by the following type synonym in the standard prelude:

```
type FailCont  =  IOError -> Dialogue
```

Similarly, the success continuation, which takes a string representing an input string and produces a new Dialogue has type `StrCont`:

```
type StrCont  =  String -> Dialogue
```

A third kind of continuation is needed for those requests which return a response of the form `Success` if successful (e.g. output requests). In this case the continuation is simply another dialogue:

```
type SuccCont =  Dialogue
```

The following list gives the type of each of the six functions corresponding to the six different kinds of I/O request described in the previous section. Full definitions for each of these functions are given in appendix B:

```
readFile   :: String -> FailCont -> StrCont -> Dialogue
writeFile  :: String -> String -> FailCont -> SuccCont -> Dialogue
appendFile :: String -> String -> FailCont -> SuccCont -> Dialogue
readChan   :: String -> FailCont -> StrCont  -> Dialogue
appendChan :: String -> String -> FailCont -> SuccCont -> Dialogue
echo       :: Bool -> FailCont -> SuccCont -> Dialogue
```

As an illustration of the use of these functions, we show how each of the example programs from the previous section can be rewritten using the continuation based style of I/O, starting with the program `helloWorld`:

```
helloWorld :: Dialogue
helloWorld  = appendChan stdout "hello, world" abort done
```

In this case, the explicit type declaration is not actually required since the type of the expression is completely determined by the type of `appendChan`. The failure continuation `abort` is equivalent to the function `(\ioerror resps -> [])` described above and terminates the program if an error occurs without any further action. In a similar way, `done` is the trivial dialogue which terminates immediately without any action. Both of these values are defined in the standard prelude:

```
done          :: Dialogue
done resps     = []
abort         :: FailCont
abort ioerror = done
```

Using the same approach, the `showFile` and `main` programs from the previous section are written as:

```
showFile :: String -> Dialogue
showFile name
 = readFile name (\ioerror -> appendChan stdout
                                ("Can't open " ++ name) abort done)
                 (\contents-> appendChan stdout contents abort done)

main :: Dialogue
main  = appendChan stdout "Please type a filename: " abort
        (readChan stdin abort
        (\userInput -> let (name : _) = lines userInput in
         readFile name
          (\ioerror  -> appendChan stdout ("Can't open " ++ name)
                             abort done)
          (\contents -> appendChan stdout contents abort done)))
```

12.3 Interactive programs

One of the principal motivations for including facilities for I/O in Gofer programs was to provide a way of using interactive programs as described in [1]. An interactive program is represented by a function of type `String -> String` mapping an input string of characters entered at the keyboard into an output string to be displayed on the screen.

There are two functions defined in the standard prelude which can be used to 'execute' functions of this kind as interactive programs:

- `interact f` executes `f::String->String` as an interactive program with echo on. This means that characters are read from the keyboard a line at a time. The usual editing characters such as backspace can be used to correct mistakes which are noticed before the return key is pressed at the end of each line. The input stream can be terminated by typing an end of file character at the beginning of a line:

```
? interact (map toUpper)
This text was entered using the interact function
THIS TEXT WAS ENTERED USING THE INTERACT FUNCTION
^Z
(874 reductions, 1037 cells)
?
```

- `run f` behaves like `interact f` except that echo is turned off. In this case, the only way of terminating the input stream without reaching the end of the string produced by `f` is to use the interrupt key:

```
? run (map toUpper)
ALTHOUGH THIS IS ENTERED IN LOWER CASE, IT STILL
APPEARS IN UPPER CASE !
{Interrupted!}

(1227 reductions, 1463 cells)
?
```

(Of these two functions, only `interact` is also included in the standard prelude for Haskell, although `run` may also be added to a Haskell system using the definition below.)

The definitions of `interact` and `run` provide further examples of Gofer programs using simple I/O facilities:

```
interact        :: (String -> String) -> Dialogue
interact f       = readChan stdin abort
                          (\s -> appendChan stdout (f s) abort done)

run             :: (String -> String) -> Dialogue
run f            = echo False abort (interact f)
```

(Exercise for the interested reader: construct alternative definitions for these functions using the stream based approach from section 12.1.)

Chapter 13

Layout

13.1 Comments

Comments provide an informal but useful way of annotating a program with a description of its purpose, structure and development. Following the definition of Haskell, two styles of comment are supported by Gofer:

- A one line comment begins with the two characters `--` and is terminated at the end of the same line. Note that an operator symbol cannot begin with `--` as this will be treated as the beginning of a comment. It is however possible to use the two characters `--` at any other position within an operator symbol. Thus a line such as:

  ```
  (xs ++ ys) -- xs
  ```

  includes a comment and will actually be treated as if the line had been written:

  ```
  (xs ++ ys)
  ```

  Whereas the line:

  ```
  xs >--> ys >--> zs
  ```

  does not contain any comments (although it will cause an error unless `>-->` has been defined using an appropriate `infixl` or `infixr` declaration).

- A nested comment begins with the characters `{-`, ends with the characters `-}` and may span any number of lines. The initial `{-` string cannot overlap with the terminating `-}` string so that the shortest possible nested comment is `{--}`, and not `{-}`. An unterminated nested comment will be treated as an error.

  As the name suggests, comments of this kind may be nested so that

  ```
  {- {- ... -} ... {- ... -} -}
  ```

  is treated as a single comment. This makes nested comments particularly convenient for enclosing parts of a program which may already contain other nested comments.

Both kinds of comment may be used in expressions entered directly into the Gofer system, or more usually, in files of definitions loaded into Gofer. The two styles of comment may be mixed within the same expression or program, remembering that the string `--` has no special significance within a nested comment and that the strings `{-` and `-}` have no special significance in a single line comment. Thus:

```
[ 2, -- {-                [ 2, {-
  3, -- -}                  -- -} 3,
  4 ]                       4 ]
```

are both equivalent to the list expression `[2,3,4]`.

13.2 The layout rule

In a tradition dating back at least a quarter of a century to Landin's ISWIM family of languages, most Gofer programs use indentation to indicate the structure of a program. For example, in a definition such as:

```
f x y = g (x + w)
        where g u = u + v
                    where v = u * u
              w   = 2 + y
```

it is clear from the layout that the definition of w is intended to be local to f rather than to g. Another example where layout plays an important role is in distinguishing the two definitions:

```
example x y z = a + b       example x y z = a + b
      where a = f x y             where a   = f x
            b = g z                     y b = g z
```

There are three situations in Gofer where indentation is typically used to determine the structure of a program:

- At the top-level of a file of definitions.
- In a group of local declarations following either of the keywords `let` or `where`.
- In a group of alternatives in a case expression, following the keyword `of`.

In each case, Gofer actually expects to find a list of items enclosed between braces '`{`' and '`}`' with individual items separated from one another by semicolons '`;`'. However, if the leading brace is not found then Gofer uses the layout rule described below to arrange for '`{`', '`}`' and '`;`' tokens to be inserted into the input stream automatically according to the indentation of each line.

In this way, the first example above will in fact be treated as if the user had entered:

```
f x y = g (x + w)
        where {g u = u + v
                     where {v = u * u
              }; w   = 2 + y
}
```

or, equivalently, just:

```
f x y = g (x + w) where {g u = u + v where {v = u * u}; w = 2 + y}
```

where the additional punctuation using the '`{`', '`}`' and '`;`' characters makes the intended grouping clear, regardless of indentation.

The layout rule used in Gofer is the same as that of Haskell, and can be described as follows:

- An opening brace '`{`' is inserted in front of the first token at the beginning of a file or following one of the keywords `where`, `let` or `of`, unless that token is itself an opening brace.
- A '`;`' token is inserted in front of the first token in any subsequent line with exactly the same indentation as the token in front of which the opening brace was inserted.
- The layout rule ends and a '`}`' token is inserted in front of the first token in a subsequent line whose indentation is strictly less than that of the token in front of which the opening brace was inserted.
- A closing brace '`}`' will also be inserted at any point where an otherwise unexpected token is encountered. This part of the rule makes it possible to use expressions such as:

  ```
  let a = fact 12 in a+a
  ```

  without needing to use the layout characters explicitly as in:

  ```
  let {a = fact 12} in a+a.
  ```

- Lines containing only whitespace (blanks and tabs) and comments do not affect the use of the layout rule.
- For the purposes of determining the indentation of each line in a file, tab stops are assumed to be placed every 8 characters, with the leftmost tab stop in column 9. Each tab character inserts one or more spaces as necessary to move to the next tab stop.
- The indentation of the end of file token is zero.

The following (rather contrived) program, is based on an example in the Haskell report [5], and provides an extended example of the use of the layout rule. A file containing the following definitions:

```
data Stack a = Empty
             | MkStack a (Stack a)

push    :: a -> Stack a -> Stack a
push x s = MkStack x s

size  :: Stack a -> Int
size s = length (stkToList s) where
           stkToList Empty         = []
           stkToList (MkStack x s) = x:xs where xs = stkToList s

pop :: Stack a -> (a, Stack a)
pop (MkStack x s) = (x, case s of r -> i r where i x = x)

top :: Stack a -> a
top (MkStack x s) = x
```

will be treated by Gofer as if it has been written:

```
{data Stack a = Empty
              | MkStack a (Stack a)

;push    :: a -> Stack a -> Stack a
;push x s = MkStack x s

;size  :: Stack a -> Int
;size s = length (stkToList s) where
           {stkToList Empty = []
           ;stkToList (MkStack x s) = x:xs where {xs = stkToList s

}};pop :: Stack a -> (a, Stack a)
;pop (MkStack x s) = (x, case s of {r -> i r where {i x = x}})

;top :: Stack a -> a
;top (MkStack x s) = x
}
```

Note that some of the more sophisticated forms of expression cannot be written on a single line (and hence entered directly into the Gofer system) without explicit use of the layout characters '{', '}' and ';':

```
? len [1..10] where len [] = 0;  len (x:xs) = 1 + len xs
10
(81 reductions, 108 cells)

? f True where f x = case x of True->n where {n=not x}; False->True
False
(4 reductions, 11 cells)

?
```

One situation in which the layout rule can cause problems is with top-level definitions. For example, the two lines:

```
f x  = 1 + x
 g y = 1 - y
```

will be treated as a single line

```
f x = 1 + x g y = 1 - y
```

which will cause a syntax error. This kind of problem becomes rather more difficult to spot if the two definitions are not on subsequent lines, particularly if they are separated by several lines of comments. For this reason, it is usually a good idea to ensure that all of the top-level definitions in a file start in the same column (the first column is usually the most convenient). COBOL and Fortran programmers are not likely to find this problem too distressing :–)

Chapter 14

Overloading in Gofer

One of the biggest differences between Gofer and most other programming languages (with the exception of Haskell) is the approach to overloading; enabling the definition and use of functions in which the meaning of a function symbol may depend on the types of its arguments.

Like Haskell, overloading in Gofer is based around a system of type classes which allow overloaded functions to be grouped together into related groups of functions. Whilst the precise details of the approach to type classes used by Gofer are quite different from those of Haskell, both rely on the same basic ideas and use a similar syntax for defining and using type classes. It would therefore seem possible that experience gained with the overloading system in one language can readily by applied to the other.

The differences embodied in the Gofer system of classes stem from my own, theoretically based investigations into 'qualified types' some of which is detailed in references [8-12]. In my personal opinion, the Gofer system has some significant advantages over the Haskell approach (see [12] for details) and one of the principal motivations behind the implementation to Gofer was to provide a way of testing such claims. One fact which I believe has already been established using Gofer is that the use and implementation of overloaded functions need not have the significant effect on performance that was anticipated with early implementations of Haskell.

This section outlines the system of type classes used in Gofer, indicating briefly how they can be used and how they are implemented.

14.1 Type classes and predicates

A type class can be thought of as a family of types (or more generally as a family of tuples of types) whose elements are called instances of the class. If C is the name of an n-parameter type class then an expression of the form $C\ t_1\ t_2 \ldots t_n$ where t_1, t_2, $\ldots$, t_n are type expressions is called a predicate and represents the assertion that the specified tuple of types is an instance of the class C.

Given a polymorphic function (e.g. `map::(a->b)->[a]->[b]`), we are free to use the function at any type which can be obtained by substituting arbitrary types for each of the type variables in its type. In Gofer, a type expression may be qualified by one or more predicates which restrict the range of types at which a value can be used. For example, a function of type `C a => a -> a -> a` can be treated as a function of type `t -> t -> t` for any instance `t` of the class `C`.

The predicate `C a` in the type expression in the previous example is called the context of the type. Contexts may contain more than one predicate in which case the predicates involved must be separated by commas and the context enclosed in parentheses as in `(C a, D b)`. The empty context is written `()` and any type expression `t` is equivalent to the qualified type `() => t`. For uniformity, a context with only one element may also be enclosed by parentheses. For technical reasons, type synonyms are not currently permitted in predicates. This is consistent with the use of predicates in Haskell, but may be relaxed, at least in certain cases, in later versions of Gofer.

14.2 The type class Eq

The type class `Eq` is a simple and useful example, whose instances are precisely those types whose elements can be tested for equality. The declaration of this class given in the standard prelude is as follows:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y      = not (x == y)
```

There are three parts in any class declaration. For this particular example we have:

- The first line (called the 'header') of the declaration introduces a name `Eq` for the class and indicates that it has a single parameter, represented by the type variable `a`.
- The second line of the declaration (the 'signature part') indicates that there are functions denoted by the operator symbols `(==)` and `(/=)` of type `a -> a -> Bool` for each instance a of class `Eq`. Using the notation introduced in the previous section, both of these operators have type:

  ```
  Eq a => a -> a -> Bool
  ```

  These functions are called the 'members' (or 'member functions') of the class. (This terminology, taken from Haskell, is rather unfortunate; thinking of a type class as a set of types, the elements of the class are called 'instances', whilst the 'members' of the class correspond more closely to the instance variables that are used in the terminology of object-oriented programming.)

  The intention is that the `(==)` function will be used to implement an equality test for each instance of the class, with the `(/=)` operator providing the corresponding inequality function. The ability to include related groups of functions within a single type class in this way is a useful tool in program design.
- The third line of the class declaration (the 'default definitions') provides a default definition of the `(/=)` operator in terms of the `(==)` operator. Thus it is only necessary to give a definition for the `(==)` operator in order to define all of the member functions for the class `Eq`. It is possible to override default member definitions by giving an alternative definition as appropriate for specific instances of the class.

14.2.1 Implicit overloading

Member functions are clearly marked as overloaded functions by their definition as part of a class declaration, but this is not the only way in which overloaded functions occur in Gofer; the restriction to particular instances of a type class is also carried over into the type of any function defined either directly or indirectly in terms of the member functions of that class. For example, the types inferred for the following two functions:

```
x  `elem`   xs   =   any (x==) xs
xs `subset` ys   =   all (`elem` ys) xs
```

are:

```
elem   :: Eq a => a -> [a] -> Bool
subset :: Eq a => [a] -> [a] -> Bool
```

(On the other hand, if none of the functions used in a particular expression or definition are overloaded then there will not be any overloading in the corresponding value. Gofer does not support the concept of implicit overloading used in some languages where a value of a particular type might automatically be coerced to a value of some supertype. An example of this would be the automatic translation of a badly typed expression `1.0 == 1` to a well-typed expression of the form `1.0 == float 1` for some (potentially overloaded) coercion function `float` mapping numeric values to elements of type Float.)

Note also that the types appearing in the context of a qualified type reflect the types at which overloaded functions are used. Thus:

```
f x ys  =  [x] == ys
```

has type `Eq [a] => a -> [a] -> Bool`, and not `Eq a => a -> [a] -> Bool`, which is the type that would be assigned to `f` in a Haskell system.

14.2.2 Instances of class Eq

Instances of a type class are defined using declarations similar to those used to define the corresponding type class. The following examples, taken from the standard prelude, give the definitions for a number of simple instances of the class `Eq`:

```
instance Eq Int  where  (==) = primEqInt

instance Eq Bool where
    True  == True   =  True
    False == False  =  True
    _     == _      =  False

instance Eq Char  where  c == d  =  ord c == ord d

instance (Eq a, Eq b) => Eq (a,b) where
    (x,y) == (u,v)  =  x==u && y==v

instance Eq a => Eq [a] where
    []     == []     =  True
    []     == (y:ys) =  False
    (x:xs) == []     =  False
    (x:xs) == (y:ys) =  x==y && xs==ys
```

The interpretation of these declarations is as follows:

- The first declaration makes `Int` an instance of class `Eq`. The function `primEqInt` is a primitive Gofer function which tests the equality of two integer values and has type `Int->Int->Bool` which tests the equality of two integer values.
- The second declaration makes `Bool` an instance of class `Eq` with a simple definition involving pattern matching.
- The third declaration makes `Char` an instance of class `Eq`. This definition indicates that a pair of characters are equal if they have the same ASCII value, which is obtained using the `ord` function. Note that the two occurrences of the symbol `(==)` in the equation:

  ```
  c == d  =  ord c == ord d
  ```

  have different meanings; the first denotes equality between characters (elements of type `Char`), whilst the second denotes equality between integers (elements of type `Int`).
- The fourth declaration provides an equality operation on pairs. Given two elements `(x,y)` and `(u,v)` of type `(a,b)` for some `a`, `b`, it must be possible to check that both `x==u` and `y==v` before we can be sure that the two pairs are indeed equal. In other words, both `a` and `b` must also be instances of `Eq` in order to make `(a,b)` an instance of `Eq`. This requirement is described by the first line in the instance declaration using the expression:

  ```
  (Eq a, Eq b) => Eq (a,b)
  ```

- The fifth declaration makes `[a]` an instance of `Eq`, whenever `a` is itself an instance of `Eq` in a similar way to the previous example. The context `Eq` a is used in the last equation in the declaration:

  ```
  (x:xs) == (y:ys)  =  x==y && xs==ys
  ```

  which contains three occurrences of the `(==)` operator; the first and third are used to compare lists of type `[a]`, whilst the second is used to compare elements of type `a`, using the instance `Eq a`.

Combining these five declarations, we obtain definitions for `(==)` on an infinite family of types including `Int`, `Char`, `Bool`, `(Int,Bool)`, `(Char,Int)`, `[Char]`, `(Bool,[Int])`, `[(Bool,Int)]`, etc.:

```
? 2 == 3                            -- using Eq Int
False
(2 reductions, 10 cells)
? (["Hello"],3) == (["Hello"],3)    -- using Eq ([[Char]],Int)
True
(31 reductions, 65 cells)
?
```

On the other hand, any attempt to use `(==)` to compare elements of some type not covered by a suitable instance declaration will result in an error. For example, the standard prelude does not define the equality operation on triples of values:

```
? (1,2,3) == (1,2,3)
ERROR: Cannot derive instance in expression
*** Expression        : (==) d125 (1,2,3) (1,2,3)
*** Required instance : Eq (Int,Int,Int)
?
```

This can be solved by including an instance declaration of the following form into a file of definitions loaded into Gofer:

```
instance (Eq a, Eq b, Eq c) => Eq (a,b,c) where
    (x,y,z) == (u,v,w)  =  x==u && y==v && z==w
```

Giving:

```
? (1,2,3) == (1,2,3)
True
(6 reductions, 20 cells)
?
```

In general, an instance declaration has the form:

`instance` *context* `=>` *predicate* `where`
definitions of member functions

The context part of the declaration gives a list of predicates which must be satisfied for the predicate on the right hand side of the '`=>`' sign to be valid. Constant predicates (i.e. predicates not involving any type variables) required by an instance declaration (such as the predicate `Eq Int` required by the third declaration) need not be included in the context. If the resulting context is empty (as in the first three declarations above) then it may be omitted, together with the corresponding '`=>`' symbol.

14.2.3 Testing equality of represented values

Instances of `Eq` can also be defined for other types, including user-defined datatypes, and unlike the instances described above, the definition of `(==)` need not be used to determine whether the values being compared have the same structure; it is often more useful to check that they represent the same value. As an example, suppose that we introduce a type constructor Set for representing sets of values, using a list to store the values held in the set:

```
data Set a = Set [a]
```

As usual, we say that two sets are equal if they have the same members, ignoring any repetitions or differences in the ordering of the elements in the lists representing the sets. This is achieved using the following instance declaration:

```
instance Eq a => Eq (Set a) where
    Set xs == Set ys  =  xs `subset` ys  &&  ys `subset` xs
                         where xs `subset` ys = all (`elem` ys) xs
```

A couple of examples illustrate the use of this definition:

```
? Set [1,2,3] == Set [3,4,1]
False
(49 reductions, 89 cells)
? Set [1,2,3] == Set [1,2,2,2,1,3]
True
(157 reductions, 240 cells)
?
```

14.2.4 Instance declarations without members

It is possible to give an instance declaration without specifying any definitions for the member functions of the class. For example:

```
instance Eq ()
```

In this case, the definition of `(==)` for the instance `Eq ()` is left completely undefined, and hence so is the definition of `(/=)`, which is defined in terms of `(==)`:

```
? () == ()
{undefined_member (==)}
(3 reductions, 34 cells)
? () /= ()
{undefined_member (==)}
(4 reductions, 36 cells)
?
```

14.2.5 Equality on function types

If an expression requires an instance of a class which cannot be obtained using the rules in the given instance declarations, then an error message will be produced when the expression is type-checked. For example, in general there is no sensible way to determine when a pair of functions are equal, and the standard prelude does not include a definition for an instance of the form `Eq (a -> b)` for any types `a` and `b`:

```
? (1==) == (\x->1==x)
ERROR: Cannot derive instance in expression
*** Expression        : (==) d148 ((==) {dict} 1) (\x->(==) {dict} 1 x)
*** Required instance : Eq (Int -> Bool)
?
```

If for some reason, you would prefer this kind of error to produce an error message when an expression is evaluated, rather than when it is type-checked, you can use an instance declaration to specify the required behaviour. For example:

```
instance Eq (a -> b) where
    (==) = error "Equality not defined between functions"
```

Evaluating the previous expression once this instance declaration has been included now produces the following result:

```
? (1==) == (\x->1==x)
{error "Equality not defined between functions"}
(42 reductions, 173 cells)
?
```

A limited form of equality can be defined for functions of type `(a->b)` if `a` has only finitely many elements, such as the boolean type `Bool`:

```
instance Eq a => Eq (Bool -> a) where
    f == g   =   f False == g False   &&   f True == g True
```

(This instance declaration would not be accepted in a Haskell program which insists that the predicate on the right of the '=>' symbol contains precisely one type constructor symbol.)

Using this instance declaration once for each argument, we can now test two functions taking boolean arguments for equality (assuming of course that their result type is also an instance of Eq).

```
? (&&) == (||)
False
(9 reductions, 21 cells)
? not == (\x -> if x then False else True)
True
(8 reductions, 16 cells)
? (&&) == (\x y-> if x then y else False)
True
(16 reductions, 30 cells)
?
```

14.2.6 Non-overlapping instances

Other instance declarations for types of the form a->b can be used at the same time, so long as no pair of declarations overlap. For example, adding the following instance declaration

```
instance Eq a => Eq (() -> a)  where  f == g  =  f () == g ()
```

enables us to evaluate expressions such as:

```
? (\()->"Hello") == const "Hello"
True
(30 reductions, 55 cells)
?
```

If however, we try to use instance declarations for types of the form (a->b) and (Bool->a) at the same time, then Gofer produces an error message similar to the following:

```
ERROR "file" (line 37): Overlapping instances for class "Eq"
*** This instance   : Eq (a -> b)
*** Overlaps with   : Eq (Bool -> a)
*** Common instance : Eq (Bool -> a)
?
```

indicating that, given the task of testing two values of type (Bool->a) for equality, there are (at least) two definitions of (==) that could be used, with potentially different results being obtained in each case.

Here is a further example of the use of non-overlapping instances of a class to define a function cat (inspired by the Unix command of the same name) which uses the I/O facilities of Gofer to print the contents of one or more files on the terminal:

```
class    Cat a        where cat  :: a -> Dialogue
instance Cat [Char]   where cat n = showFile n done
instance Cat [[Char]] where cat   = foldr showFile done

showFile name cont = readFile name abort
                          (\s->appendChan stdout s abort cont)
```

Given these declarations, an expression of the form:

```
cat "file"
```

can be used to display the contents of the named file, whilst a list of files can be printed one after the other using an expression of the form:

```
cat ["file1", "file2", ..., "filen"].
```

14.3 Dictionaries

In order to understand some of the messages produced by Gofer, as well as some of the more subtle problems associated with overloaded functions, it is useful to have a rough idea of the way in which overloaded functions are implemented.

The basic idea is that a function with a qualified type `context => type` where `context` is a non-empty list of predicates is implemented by a function which takes an extra argument for each predicate in the context. When the function is used, each of these parameters is filled by a 'dictionary' which gives the values of each of the member functions in the appropriate class. None of these extra parameters is entered by the programmer. Instead, they are inserted automatically during type-checking.

For the class `Eq`, each dictionary has at least two elements containing definitions for each of the functions `(==)` and `(/=)`. A dictionary for an instance of `Eq` can be depicted by a diagram of the form:

(==)	(/=)	...

In order to produce useful error messages and indicate the way in which dictionary expressions are being used, Gofer uses a number of special notations for printing expressions involving dictionaries:

`(#1` *d*`)`	selects the first element of the dictionary *d*
`(#2` *d*`)`	selects the second element of the dictionary *d*
`(#`*n* *d*`)`	selects the *n*th element of the dictionary *d* (note that `(#0 d)` is equivalent to the dictionary *d*).
`{`*dict*`}`	denotes a specific dictionary (the contents are not displayed).
`d`*nnn*	a dictionary variable representing an unknown dictionary is printed as a lower case letter '`d`' followed by an integer; e.g. `d231`.

Note that, whilst these notations are used in output produced by Gofer and in the following explanations, they cannot be entered directly into Gofer expressions or programs – even if you use a variable such as `d1` in an expression, Gofer will not confuse this with a dictionary variable with the same name (although Gofer might confuse you by using the same name in two different contexts!).

Using these notations, the member functions `(==)` and `(/=)` of the class `Eq` behave as if they were defined by the expressions:

```
(==) d1  =  (#1 d1)
(/=) d1  =  (#2 d1)
```

To understand how these definitions work, we need to take a look at a specific dictionary. Following the original instance declaration used for `Eq Int`, the corresponding dictionary is:

`d ::  Eq Int`

primEqInt	defNeq d

Note that the dictionary variable `d` is used as a name for the dictionary in this diagram, indicating how values within a dictionary can include references to the same dictionary.

It turns out that predicates play a very similar role for dictionaries as types play for normal values. This motivates our use of the notation `d :: Eq Int` to indicate that `d` is a dictionary for the instance `Eq Int`. One difference between these, particularly important for theoretical work, is that dictionary values are uniquely determined by predicates; if `d1::p` and `d2::p` for some predicate `p`, then `d1 = d2`.

The value held in the first element of the dictionary is the primitive equality function on integers, `primEqInt`. The following diagram shows how the dictionary is used to evaluate the expression `2 == 3`. Note that this expression will first be translated to `(==) d 2 3` by the type checker. The evaluation then proceeds as follows:

```
(==) d 2 3  ==>  (#1 d) 2 3
            ==>  primEqInt 2 3
            ==>  False
```

The second element of the dictionary is a little more interesting because it uses the default definition for (/=) given in the original class definition which, after translation, is represented by the function **defNeq** defined by:

```
defNeq d1 x y = not ((==) d1 x y)
```

Notice the way in which the extra dictionary parameter is used to obtain the appropriate overloading. For example, evaluation of the expression `2 /= 3`, which becomes `(/=) d 2 3` after translation, proceeds as follows:

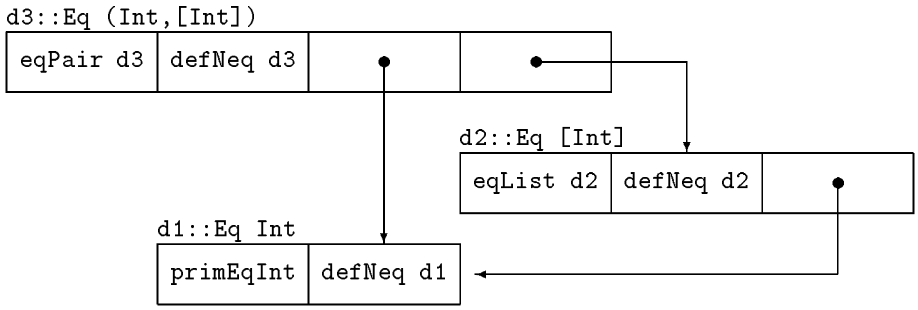
```
(/=) d 2 3  ==>  (#2 d) 2 3
            ==>  defNeq d 2 3
            ==>  not ((==) d 2 3)
            ==>  not ((#1 d) 2 3)
            ==>  not (primEqInt 2 3)
            ==>  not False
            ==>  True
```

(Clearly there is some scope for optimisation here; whilst the actual reduction sequences used by Gofer are equivalent to those illustrated above, the precise details are a little different.)

If an instance is obtained from an instance declaration with a non-empty context, then the basic two element dictionary used in the examples above is extended with an extra dictionary value for each predicate in the context. As an example, the diagram below shows the dictionaries that will be created from the instance definitions in section 14.2.2 for the instance `Eq (Int, [Int])`. The functions `eqPair` and `eqList` which are used in these dictionaries are obtained from the definitions of (==) given in the instance declarations for `Eq (a,b)` and `Eq [a]` respectively:

```
eqPair d (x,y) (u,v)   = (==) (#3 d) x u && (==) (#4 d) y v

eqList d [] []         = True
eqList d []     (y:ys) = False
eqList d (x:xs) []     = False
eqList d (x:xs) (y:ys) = (==) (#3 d) x y && (==) d xs ys
```

The dictionary structure for `Eq (Int, [Int])` is as follows. Note that the Gofer system ensures that there is at most one dictionary for a particular instance of a class, and that the dictionary `d1 :: Eq Int` in this system is automatically shared between `d2` and `d3`:



Once again, it may be useful to see how these definitions are used to evaluate the expression `(2,[1]) == (2,[1,3])` which, after translation, becomes `(==) d3 (2,[1]) (2,[1,3])`:

```
(==) d3 (2,[1]) (2,[1,3])
        ==>   (#1 d3) (2,[1]) (2,[1,3])
        ==>   eqPair d3 (2,[1]) (2,[1,3])
        ==>   (==) (#3 d3) 2 2  &&  (==) (#4 d3) [1] [1,3]
        ==>   (==) d1 2 2       &&  (==) (#4 d3) [1] [1,3]
        ==>   (#1 d1) 2 2       &&  (==) (#4 d3) [1] [1,3]
        ==>   primEqInt 2 2     &&  (==) (#4 d3) [1] [1,3]
        ==>   True              &&  (==) (#4 d3) [1] [1,3]
        ==>   (==) (#4 d3) [1] [1,3]
```

```
==>   (==) d2 [1] [1,3]
==>   (#1 d2) [1] [1,3]
==>   eqList d2 [1] [1,3]
==>   (==) (#3 d2) 1 1  &&  (==) d2 [] [3]
==>   (==) d1 1 1       &&  (==) d2 [] [3]
==>   (#1 d1) 1 1       &&  (==) d2 [] [3]
==>   primEqInt 1 1     &&  (==) d2 [] [3]
==>   True              &&  (==) d2 [] [3]
==>   (==) d2 [] [3]
==>   False
```

14.3.1 Superclasses

In general, a type class declaration has the form:

> `class` *context* `=>` *Class* $a_1 \ldots a_n$ `where`
> *type declarations for member functions*
> *default definitions of member functions*

where *Class* is the name of the new type class which takes n arguments, represented by distinct type variables $a_1, \ldots, a_n$. As in the case of instance declarations, the context that appears on the left hand side of the '`=>`' symbol specifies a list of predicates that must be satisfied in order to construct any instance of *Class*.

The predicates in the *context* part of a class declaration are called the superclasses of *Class*. This terminology is taken from Haskell where all classes have a single parameter and each of the predicates in the context part of a class declaration has the form $C\ a_1$; in this situation, any instance of *Class* must also be an instance of each class C named in the context. In other words, each such C contains a superset of the types in *Class*.

As an example of a class declaration with a non-empty context, consider the following declaration from the standard prelude which introduces a class Ord whose instances are types with both strict `(<)`, `(>)` and non-strict `(<=)`, `(>=)` versions of an ordering defined on their elements:

```
class Eq a => Ord a where
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a

    x <  y               = x <= y && x /= y
    x >= y               = y <= x
    x >  y               = y < x

    max x y | x >= y  = x
            | y >= x  = y
    min x y | x <= y  = x
            | y <= x  = y
```

Notice that this definition provides default definitions for all of the member functions except `(<=)`, so that in general only this single function needs to be defined to construct an instance of class `Ord`.

There are two reasons for defining `Eq` as a superclass of `Ord`:

- The default definition for `(<)` relies on the use of `(/=)` taken from class `Eq`. In order to guarantee that this is always valid we must ensure that every instance of `Ord` must also be an instance of `Eq`.
- Given the definition of a non-strict ordering `(<=)` on the elements of a type, it is always possible to construct a definition for the `(==)` operator (and hence for `(/=)`) using the equation:

  ```
  x==y   =   x<=y && y<=x
  ```

  There will therefore be no loss in generality by requiring `Eq` to be a superclass of `Ord`, and conversely, no difficulty in defining an instance of `Eq` to accompany any instance of `Ord` for which an instance of `Eq` has not already be provided.

As an example, the following definitions provide an alternative way to implement the equality operation on elements of the Set datatype described in section 14.2.3, in terms of the subset ordering defined in class Ord:

```
instance Ord (Set a) => Eq (Set a) where
    x == y   =   x <= y  &&  y <= x

instance Eq a => Ord (Set a) where
    Set xs <= Set ys  =  all (`elem` ys) xs
```

This definition is in fact no less efficient or effective than the original version.

Dictionaries for superclasses are dealt with in much the same way as the instance specific dictionaries described above. For example, the general layout of a dictionary for an instance of Ord is illustrated in the following diagram:

(<)	(<=)	(>)	(>=)	max	min	Eq a	...

Note the use of the seventh element of this dictionary which points to the dictionary for the appropriate instance of Eq. This is used in the translation of the default definition for (<) which is equivalent to:

```
defLessThan d x y  =  (<=) d x y  &&  (/=) (#7 d) x y
```

14.3.2 Combining classes

In general, a dictionary is made up of three separate parts:

Implementation of class members	Superclass dictionaries	Instance specific dictionaries

Each of these may be empty. We have already seen examples in which there are no superclass dictionaries (e.g. instances of Eq) and in which there are no instance specific dictionaries (e.g. Eq Int). Classes with no member functions (corresponding to dictionaries with no member functions) are sometimes useful as a convenient abbreviation for a list of predicates. For example:

```
class  C a  where  cee :: a -> a
class  D a  where  dee :: a -> a

class  (C a, D a) => CandD a
```

makes CandD a an abbreviation for the context (C a, D a). Thinking of single parameter type classes as sets of types, the type class CandD corresponds to the intersection of classes C and D.

Just as the type inferred for a particular function definition or expression does not involve type synonyms unless explicit type signatures are used, the Gofer type system will not use a single predicate of the form CandD a instead of the two predicates C a and D a unless explicit signatures are used:

```
? :t dee . cee
\d129 d130 -> dee d130 . cee d129 :: (C a, D a) => a -> a
? :t dee . cee :: CandD a => a -> a
\d129 -> dee (#2 d129) . cee (#1 d129) :: CandD a => a -> a
?
```

In Haskell, all instances of a class such as CandD must have explicit declarations, in addition to the corresponding declarations for instances for C and D. This problem can be avoided by using the more general form of instance declaration permitted in Gofer; a single instance declaration:

```
instance CandD a
```

is all that is required to ensure that any instance of CandD can be obtained, so long as corresponding instances for C and D can be found.

14.3.3 Simplified contexts

Consider the function defined by the following equation:

```
eg1 x    =    [x] == [x]   ||   x == x
```

This definition does not restrict the type of `x` in any way except that, if `x::a`, then there must be instances `Eq [a]` and `Eq a` which are used for the two occurrences of the `(==)` operator in the equation. We might therefore expect the type of `eg1` to be:

```
(Eq [a], Eq a) => a -> Bool
```

with translation:

```
eg1 d1 d2 x  =  (==) d1 [x] [x]   ||   (==) d2 x x
```

However, as can be seen from the case where `a=Int` illustrated in section 14.3, given `d1::Eq [a]` we can always find a dictionary for `Eq a` by taking the third element of `d1` i.e. `(#3 d1)::Eq a`. Since it is more efficient to select an element from a dictionary than to complicate both type and translation with extra parameters, the type assigned to `eg1` by default is:

```
Eq [a] => a -> Bool
```

with translation:

```
eg1 d1 x  =  (==) d1 [x] [x]  || (==) (#3 d1) x x
```

In general, given a set of predicates corresponding to the instances required by an expression, Gofer will always attempt to find the smallest possible subset of these predicates such that all of the required dictionaries can still be obtained, whilst minimising the number of dictionary parameters that are used.

The original type and translation for eg1 given above can be produced by including an explicit type signature in the file containing the definition of `eg1`:

```
eg1   ::  (Eq [a], Eq a) => a -> Bool
eg1 x  =  [x] == [x]   ||   x == x
```

But even with this definition, Gofer will still always try to minimise the number of dictionaries used in any particular expression:

```
? :t eg1
\d153 -> eg1 d153 (#3 d153) :: Eq [a] => a -> Bool
?
```

As another example, consider the expression `(\x y->  x==x  ||  y==y)`. The type and translation assigned to this term can be found directly using Gofer:

```
? :t (\x y-> x==x || y==y)
\d121 d122 x y -> (==) d122 x x ||
                  (==) d121 y y
               :: (Eq b, Eq a) => a -> b -> Bool
?
```

Note that the translation has two dictionary parameters `d121` and `d122` corresponding to the two predicates `Eq a` and `Eq b` respectively. Since both of these dictionaries can be obtained from a dictionary for the predicate `Eq (a,b)`, we can use an explicit type signature to produce a translation which needs only one dictionary parameter:

```
? :t (\x y-> x==x || y==y) :: Eq (a,b) => a -> b -> Bool
\d121 x y -> (==) (#3 d121) x x ||
             (==) (#4 d121) y y
          :: Eq (a,b) => a -> b -> Bool
?
```

14.4 Other issues

14.4.1 Unresolved overloading

Consider the use of the (==) operator in the following three situations:

- In the expression `2==3`, it is clear that the appropriate value for the equality operator in this case is `primIntEq` as defined by the instance declaration for `Eq Int`. The expression can therefore be translated to `primEqInt 2 3`.
- In the function definition `f x = x==x`, we cannot completely determine the appropriate value for (==) because it depends on the type assigned to the variable `x`, which may itself vary with different uses of the function `f`. It is however possible to add an extra parameter to the definition, giving `f d x = (==) d x x` and taking the type of `f` to be `Eq a => a -> Bool`.

  In this way, the problem of finding the appropriate definition for the (==) operator is deferred until the function is actually used.
- In the expression `[]==[]`, the appropriate value for (==) must be obtained from the dictionary for some instance of the form `Eq [a]`, but there is not sufficient information in the expression to determine what the value of the type variable a should be.

  Looking back to the instance declaration for `Eq [a]`, we find that the definition of (==) depends on the value of the dictionary for the instance `Eq a`. In this particular case, it is clear that the expression will always evaluate to `True`, regardless of the value of this dictionary. Unfortunately, the only way that this can be detected is by evaluating the expression to see if the calculation can be completed without reference to the dictionary value (see the comments in the aside at the end of this section).

  Attempting to evaluate this expression in Gofer will therefore result in an error message indicating that the expression does not contain sufficient information to resolve the use of overloading in the expression:

  ```
  ? [] == []
  ERROR: Unresolved overloading
  *** type        : Eq [a] => Bool
  *** translation : \d129 -> (==) d129 [] []
  ?
  ```

  Note that the expression has been converted into a lambda expression using the dictionary variable `d129` to represent the dictionary for the unknown instance `Eq [a]`.

  One simple way to resolve the overloading in an expression of this kind is to use an explicit type signature. For example, if we specify that the second empty list is an empty list of type `[Int]`:

  ```
  ? [] == ([]::[Int])
  True
  (2 reductions, 9 cells)
  ?
  ```

The same problem occurs in Haskell, where it is described using the idea of an 'ambiguous type' – i.e. a type expression of the form `context => type` where one or more of the type variables appearing in the given context do not appear in the remaining part of the type expression.

Further examples of unresolved overloading occur with other classes. As an example consider the class `Reader` defined by:

```
class Reader a where
    parse   :: String -> a
    unparse :: a -> String
```

whose member functions provide methods for obtaining the string representation of an element of an instance type, and for converting such representations back into the original values. (The standard Haskell `Text` class contains similar functions.) Now consider the expression `parse . unparse` which maps values from some instance of Reader to values of another instance via an intermediate string representation.

```
? parse . unparse
ERROR: Unresolved overloading
*** type        : (Reader a, Reader b) => a -> b
*** translation : \d129 d130 -> parse d130 . unparse d129
?
```

One of the first things that might surprise the reader here is that the value produced by `parse.unparse` does not have to be of the same type as the argument; for example, we would not usually expect to have any sensible interpretation for a floating point number obtained from the string representation of a boolean value!

This can be fixed by using an explicit type declaration, although the expression still produces unresolved overloading:

```
? (parse . unparse) :: Reader a => a -> a
ERROR: Unresolved overloading
*** type        : Reader a => a -> a
*** translation : \d130 -> parse d130 . unparse d130
?
```

Notice however that the type of this expression is not ambiguous so that the unresolved overloading in this example can be eliminated when the function is actually used:

```
? ((parse . unparse) :: Reader a => a -> a) 'a'
'a'
(4 reductions, 11 cells)
?
```

A more serious problem occurs with the expression `unparse . parse` which maps string values to string values via some intermediate type. Clearly this will lead to a problem with unresolved overloading:

```
? unparse . parse
ERROR: Unresolved overloading
*** type        : Reader a => String -> String
*** translation : \d130 -> unparse d130 . parse (#0 d130)
?
```

Notice that the type obtained in this case is ambiguous; the type variable `a` which appears in the predicate `Reader a` does not appear in the type `String -> String`. There are a number of ways of resolving this kind of ambiguity:

- Using an explicitly typed expression: Assuming for example that `Char` is an instance of `Reader`, we can write:

  ```
  ? unparse . (parse :: String -> Char)
  v113 {dict} . v112 {dict}
  (5 reductions, 42 cells)
  ?
  ```

  without any ambiguity. If such type signatures are used in a number of places, it might be better to define an auxiliary function and use that instead:

  ```
  charParse :: String -> Char
  charParse  = parse

  ? unparse . charParse
  v113 {dict} . charParse
  (4 reductions, 37 cells)
  ?
  ```

  In such situations, it is perhaps worth asking if overloaded functions are in fact the most appropriate solution for the problem at hand!

- Using an extra dummy parameter in a function definition. In a definition such as:

  ```
  f = unparse . parse
  ```

  we can introduce an additional dummy parameter `x` which is not used except to determine the type of the result produced by parse in `f`:

  ```
  f x  =  unparse . (parse `asTypeOf` (\""->x))
  ```

  where the standard prelude operator `asTypeOf` defined by:

  ```
  asTypeOf      :: a -> a -> a
  x `asTypeOf` _ = x
  ```

  is used to ensure that the type of parse in the definition of `f` is the same as that of the function `(\""->x)` – in other words, the type must be `String -> a` where `a` is the type of the variable `x`.

  The resulting type for `f` is:

  ```
  f :: Reader a => a -> String -> String
  ```

  Notice how the addition of the dummy parameter has been used to eliminate the ambiguity present in the original type.

  This kind of 'coding trick' is rather messy and is not recommended for anything but the simplest examples.

The idea of evaluating an expression with an ambiguous type to see if it does actually need the unspecified dictionaries could have been implemented quite easily in Gofer using an otherwise unused datatype `Unresolved` and generating instance declarations such as:

```
instance Eq Unresolved where
    (==) = error "unresolved overloading for (==)"
    (/=) = error "unresolved overloading for (/=)"
```

for each class. Given a particular expression, we can then use the type `Unresolved` in place of any ambiguous type variables in its type. The evaluation of the expression could then be attempted, either completing successfully if the dictionaries are not required, but otherwise resulting in a run-time error.

This approach is not used in Gofer; instead, the programmer is notified of any unresolved polymorphism when the program is type checked, avoiding the possibility that a program might contain an undetected ambiguity.

14.4.2 'Recursive' dictionaries

Unlike Haskell, there are no restrictions on the form of the predicates that may appear in the context part of a Gofer class or instance declaration. This has a number of potentially useful applications because it enables the Gofer programs to use mutually 'recursive' systems of dictionaries.

One example of this is the ability to implement a large family of related functions using a group of classes instead of having to use a single class. The following example illustrates the technique with an alternative definition for the class `Eq` in which the `(==)` and `(/=)` operators are placed in different classes:

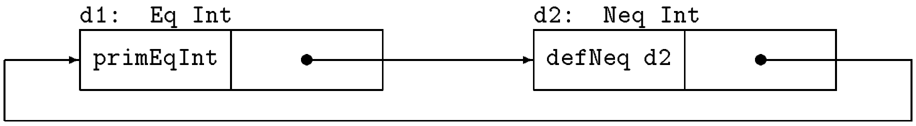
```
class Neq a => Eq a  where  (==) :: a -> a -> Bool

class Eq a => Neq a  where  (/=) :: a -> a -> Bool
                            x/=y  = not (x == y)
```

(These declarations clash with those in the standard prelude and hence cannot actually be used in Gofer unless a modified version of the standard prelude is used instead.)

If we then give instance declarations:

```
instance Eq Int  where (==) = primEqInt
instance Neq Int
```

and try to evaluate the expression `2==3` then the following system of dictionaries will be generated:



where the function `defNeq` is derived from the default definition in the class `Neq` and is equivalent to:

```
defNeq d x y  =  not ((==) (#2 d) x y)
```

Incidentally, if the instance declaration for `Neq Int` above had been replaced by:

```
instance Neq a
```

then the effect of these declarations would be similar to the standard definition of the class `Eq`, except that it would not be possible to override the default definition for `(/=)`. In other words, this approach would give the same effect as defining `(/=)` as a top-level function rather than a member function in the class `Eq`:

```
class Eq a  where  (==) :: a -> a -> Bool

(/=)   ::  Eq a => a -> a -> Bool
x /= y  =  not (x == y)
```

There are other situations in which recursive dictionaries of the kind described above can be used. A further example is given in the following section. Unfortunately, the lack of restrictions on the form of class and instance declarations can also lead to problems in some (mostly pathological) cases. As an example, consider the class:

```
class Bad [a] => Bad a  where  bad :: a -> a
```

Without defining any instances of `Bad`, it is not possible to construct any dictionaries for instances of `Bad`:

```
? bad 2
ERROR: Cannot derive instance in expression
*** Expression        : bad d126 2
*** Required instance : Bad Int
?
```

If however we add the instance declarations:

```
instance Bad Int where bad = id
instance Bad [a] where bad = id
```

then any attempt to construct a dictionary for `Bad Int` will also require a dictionary for the superclass `Bad [Int]` and then for the superclass of that instance `Bad [[Int]]` etc. Since Gofer has only a finite amount of space for storing dictionaries, this process will eventually terminate when that space has been used up:

```
? bad 2
ERROR: Dictionary storage space exhausted
?
```

(Depending on the configuration of your particular version of Gofer and on the nature of the class and instance declarations that are involved, an alternative error message `ERROR: Too many type variables in type checker` may be produced instead of the message shown above.)

From a practical point of view, this problem is unlikely to cause too many real difficulties:

- Class declarations involving predicates such as those in the declaration of `Bad` are unlikely to be used in realistic programs.

- All dictionaries are constructed before evaluation begins. This process is guaranteed to terminate because each new dictionary that is created uses up part of the space used to hold Gofer dictionaries. The construction process will either terminate successfully once complete, or be aborted as soon as all of the dictionary space has been used.

It remains to see what impact (if any) this has on realistic programs, and if later versions of Gofer should be modified to impose some syntactic restrictions (as in Haskell) or perhaps some form of static checking of the contexts appearing in class and instance declarations.

14.4.3 Classes with multiple parameters

Gofer is the first language to support the use of type classes with multiple parameters. This again is an experimental feature of the language, intended to make it possible to explore the claims from a number of researchers about the use of such classes.

Initial experiments suggest that multiple parameter type classes are likely to lead to large numbers of problems with unresolved overloading. Ultimately, this may mean that such classes are only of practical use in explicitly typed languages, or alternatively that a more powerful and general defaulting mechanism (similar to that used in Haskell with numeric classes) is required to support user controlled overloading resolution.

The following declaration introduces a class `Iso` whose elements are pairs of isomorphic types:

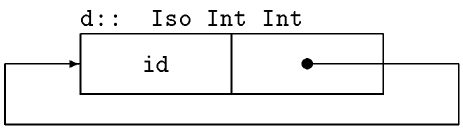
```
class Iso b a => Iso a b  where  iso :: a -> b
```

The single member function `iso` represents the isomorphism mapping elements of type `a` to corresponding elements of type `b`. Note the 'superclass' context in this declaration which formalises the idea that if `a` is isomorphic to `b` then `b` is also isomorphic to `a`. The class `Iso` therefore provides further examples of the recursive dictionaries described in the previous section.

The fact that any type is isomorphic to itself can be described by the following instance declaration:

```
instance Iso a a where iso x = x
```

For example, the dictionary structure created in order to evaluate the expression `iso 2 == 3` is:




```
? iso 2 == 3
False
(4 reductions, 11 cells)
?
```

Our first taste of the problems to come occurs when we try to evaluate the expression `iso 2 == iso 3`:

```
? iso 2 == iso 3
ERROR: Unresolved overloading
*** type        : (Eq a, Iso Int a) => Bool
*** translation : \d130 d132 -> (==) d130 (iso d132 2) (iso d132 3)
?
```

In this case, the `iso` function is used to map the integers 2 and 3 to elements of some type `a`, isomorphic to `Int`, and the values produced are the compared using `(==)` at the instance `Eq a`; there is no way of discovering what the value of a should be without using an explicit type signature.

Further instances can be defined. The following two declarations are needed to describe the (approximate) isomorphism between lists of pairs and pairs of lists:

```
instance Iso [(a,b)] ([a],[b]) where
    iso xs = (map fst xs, map snd xs)
```

```
instance Iso ([a],[b]) [(a,b)] where
    iso (xs,ys) = zip xs ys
```

Unfortunately, even apparently straightforward examples give problems with unresolved overloading, forcing the use of explicit type declarations:

```
? iso [(1,2),(3,4)]
ERROR: Unresolved overloading
*** type        : Iso [(Int,Int)] a => a
*** translation : \d126 -> iso d126 [(1,2),(3,4)]

? (iso [(1,2),(3,4)]) :: ([Int],[Int])
([1, 3],[2, 4])
(22 reductions, 64 cells)
?
```

A second example of a multiple parameter type class is defined as follows:

```
class Ord a => Collects a b where
    emptyCollection :: b
    addToCollection :: a -> b -> b
    listCollection  :: b -> [a]
```

The basic intuition is that the predicate `Collects a b` indicates that elements of type `b` can be used to represent collections of elements of type `a`. A number of people have suggested using type classes in this way to provide features similar to the (similarly named, but otherwise different) classes that occur in object-oriented languages.

Obvious implementations involve the use of ordered lists or binary search trees defined by instances of the form:

```
data STree a = Empty | Node a (STree a) (STree a)

instance Collects a [a] where ....
instance Collects a (STree a) where ....
```

Once again, there are significant problems even with simple examples using these functions. As an example, the standard way of defining a function of type:

```
Collects a b => [a] -> b
```

mapping a list of values to a collection of those values using the higher order function `foldr`:

```
listToCollection = foldr addToCollection emptyCollection
```

actually produces a function with ambiguous type:

```
? :t foldr addToCollection emptyCollection
\d139 d140 -> foldr (addToCollection d140) (emptyCollection d139)
          :: (Collects c b, Collects a b) => [a] -> b
?
```

which cannot be resolved, even with an explicit type declaration.

14.4.4 Overloading and numeric values

One of the most common uses of overloading is to allow the use of the standard arithmetic operators such as `(+)`, `(*)` etc. on the elements of a range of numeric types including integers and floating point values in addition to user defined numeric types such as arbitrary precision integers, complex and rational numbers, vectors and matrices, polynomials etc. In Haskell, these features are supported by a number of built-in types and a complex hierarchy of type classes describing the operations defined on the elements of each numeric type.

As an experimental language, intended primarily for the investigation of general purpose overloading, Gofer has only two built-in numeric types; `Int` and `Float` (the second of which is not supported in all implementations). Similarly, although the Gofer system could be used to implement the fully hierarchy of Haskell numeric classes, the standard prelude uses a single numeric type class Num defined by:

```
class Eq a => Num a where           -- simplified numeric class
    (+), (-), (*), (/) :: a -> a -> a
    negate             :: a -> a
    fromInteger        :: Int -> a
```

The first four member functions `(+)`, `(-)`, `(*)`, `(/)` are the standard arithmetic functions on instances of `Num`, whilst `negate` denotes unary negation. The final member function, fromInteger is used to coerce any integer value to the corresponding value in another instance of `Num`. An expression such as `fromInteger 3` is called an overloaded numeric constant and has type `Num a => a` indicating that it can be used as a value of any instance of `Num`. See below for examples.

Both `Float` and `Int` are defined as instances of `Num` using primitive functions for integer and floating point arithmetic:

```
instance Num Int where
    (+)           = primPlusInt
    (-)           = primMinusInt
    (*)           = primMulInt
    (/)           = primDivInt
    negate        = primNegInt
    fromInteger x = x

instance Num Float where
    (+)         = primPlusFloat
    (-)         = primMinusFloat
    (*)         = primMulFloat
    (/)         = primDivFloat
    negate      = primNegFloat
    fromInteger = primIntToFloat
```

These definitions make it possible to evaluate numeric expressions involving both types:

```
? 2 + 3
5
(3 reductions, 6 cells)
? 3.2 + 4.321
7.521
(3 reductions, 13 cells)
?
```

Note however that any attempt to evaluate an expression mixing different arithmetic types is likely to cause a type error:

```
? 4.2 * 4
ERROR: Type error in application
*** expression     : 4.2 * 4
*** term           : 4.2
*** type           : Float
*** does not match : Int
?
```

Further problems occur when we try to define functions intended to be used with arbitrary instances of `Num` rather than specific numeric types. As an example of this, the standard prelude function `sum`, roughly equivalent to:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

has type `[Int] -> Int`, rather than the more general `Num a => [a] -> a` which could be used to find the sum of a list of numeric values in any instance of `Num`. The problem in this particular case is caused by the integer constant 0 in the first line of the definition. Replacing this with the expression fromInteger 0 leads to the following definition for a generic sum function of the required type:

```
genericSum       :: Num a => [a] -> a
genericSum []     = fromInteger 0
genericSum (x:xs) = x + genericSum xs
```

For example:

```
? genericSum [1,2,3]
6
(10 reductions, 18 cells)
? genericSum [1.0,2.0,3.0]
6.0
(11 reductions, 27 cells)
?
```

The `fromInteger` function can also be used to solve the previous problem:

```
? 4.2 * fromInteger 4
16.8
(3 reductions, 13 cells)
?
```

In Haskell, any integer constant `k` appearing in an expression is treated as if the programmer had actually written `fromInteger k` so that both of the preceding problems are automatically resolved. Unfortunately, this also creates some new problems; applying the function fromInteger to each integer constant in the previous examples causes problems with unresolved overloading:

```
? fromInteger 2 + fromInteger 3
ERROR: Unresolved overloading
*** type        : Num a => a
*** translation : \d143 -> (+) d143 (fromInteger d143 2)
                                    (fromInteger d143 3)
?
```

Once again, Haskell provides a solution to this problem in the form of a 'default mechanism' for numeric types which, once the following problem has been detected, will typically 'default' the unknown type represented by the type variable a above to be Int, so that the result is actually equivalent to the following:

```
? (fromInteger 2 + fromInteger 3) :: Int
5
(4 reductions, 8 cells)
?
```

There are a number of problems with the Haskell default mechanism; both theoretical and practical. In addition, if a default mechanism of some form is used then it should also be capable of dealing with arbitrary user-defined type classes, rather than a small group of 'standard' classes, in order to provide solutions to the unresolved overloading problems described in previous sections. Therefore, for the time being, Gofer does not support any form of default mechanism and overloaded numeric constants can only be obtained by explicit use of the fromInteger function.

14.4.5 Constants in dictionaries

The Gofer system constructs new dictionaries as necessary, and deletes them when they are no longer required. At any one time, there is at most one dictionary for each instance of a class. Coupled with lazy evaluation, this has a number of advantages for classes in which member functions are defined by variable declarations as in section 9.10. As an example, consider the class Finite defined by:

```
class Finite a  where  members :: [a]
```

The only member in this class is a list enumerating the elements of the type. For example:

```
instance Finite Bool  where  members = [False, True]

instance (Finite a, Finite b) => Finite (a,b) where
    members = [ (x,y) | x<-members, y<-members ]
```

In order to overcome any problems with unresolved overloading, explicit type signatures are often needed to resolve overloading:

```
? members :: [Bool]
[False, True]
(6 reductions, 26 cells)
? length (members :: [((Bool,Bool),(Bool,Bool))])
16
(103 reductions, 195 cells)
?
```

In some cases, the required overloading is implicit from the context and no additional type information is required, as in the following example:

```
? [ x && y | (x,y) <- members ]
[False, False, False, True]
(29 reductions, 90 cells)
?
```

We can also use the technique of passing a 'dummy' parameter to resolve overloading problems in a function definition:

```
size  :: Finite a => a -> Int
size x = length (members `asTypeOf` [x])
```

which calculates the number of elements of a finite type, given an arbitrary element of that type:

```
? size (True,False)
4
(31 reductions, 60 cells)
?
```

Now consider the expression `size (True,False) + size (True,False)`. At first glance, we expect this to repeat the calculation in the previous example two times, requiring approximately twice as many reductions and cells as before. However, before this expression is evaluated, Gofer constructs a dictionary for `Finite (Bool,Bool)`. The evaluation of the first summand forces Gofer to evaluate the value for "members" in this dictionary. Since precisely the same dictionary is used to calculate the value of the second summand, the evaluation of "members" is not repeated and the complete calculation actually uses rather fewer reductions and cells:

```
? size (True,False) + size (True,False)
8
(51 reductions, 90 cells)
?
```

On the other hand, repeating the original calculation gives exactly the same number of reductions and cells as before, because the dictionaries constructed at the beginning of each calculation are not retained for use in subsequent calculations.

We can force Gofer to construct specific dictionaries whilst reading from a file of definitions, so that they are not deleted at the end of each calculation, using an explicitly typed variable definition such as:

```
boolBoolMembers = members :: [(Bool,Bool)]
```

This forces Gofer to construct the dictionary `Finite (Bool,Bool)` when the file of definitions is loaded and prevents it from being deleted at the end of each calculation. Having loaded a file containing this definition, the first two attempts to evaluate `size (True,False)` give:

```
? size (True,False)
4
(31 reductions, 60 cells)
? size (True,False)
4
(20 reductions, 32 cells)
?
```

14.4.6 The monomorphism restriction

This section describes a technique used to limit the amount of overloading used in the definition of certain values to avoid a number of technical problems. This particular topic has attracted quite a lot of attention within the Haskell community where it is affectionately known as the 'dreaded monomorphism restriction'. Although the initial formulation of the rule was rather cumbersome and limiting, the current version used in both Gofer and Haskell is unlikely to cause any problems in practice. In addition, many of the examples used to motivate the need for the monomorphism restriction in Haskell occur as a result of the use of implicitly overloaded numeric constants, described in section 14.4.4, and hence do not occur in Gofer.

The monomorphism restriction takes its name from the way in which it limits the amount of polymorphism that can be used in particular kinds of declaration. Although we touch on this point in the following discussion, the description given here uses an equivalent, but less abstract approach, based on observations about the implementation of overloaded functions.

Basic ideas: As we have seen, the implementation of overloading used by Gofer depends on being able to add extra arguments to a function definition to supply the required dictionary parameters. For example, given a function definition such as:

```
isElement x []     =  False
isElement x (y:ys) =  x==y || isElement x ys
```

we first add a dictionary parameter for the use of the overloaded `(==)` operator on the right hand side, obtaining:

```
isElement x []     = False
isElement x (y:ys) = (==) d x y || isElement x ys
```

Finally, we have to add the variable `d` as a new parameter for the function `isElement`, on both the left and right hand sides of the definition:

```
isElement d x []     = False
isElement d x (y:ys) = (==) d x y || isElement d x ys
```

The monomorphism restriction imposes conditions which prevent this last step from being used for certain kinds of value binding.

Declaration groups: Before giving the full details, it is worth pointing out that, in general, the monomorphism restriction affects groups of value declarations rather than just individual definitions. To illustrate this point, consider the function definitions:

```
f x y  =  x==y || g x y
g x y  =  not (f x y)
```

Adding an appropriate dictionary parameter for the `(==)` operator gives:

```
f x y  =  (==) d x y || g x y
g x y  =  not (f x y)
```

The next stage is to make this dictionary variable into an extra parameter to the function `f` wherever it appears, giving:

```
f d x y  =  (==) d x y || g x y
g x y    =  not (f d x y)
```

But now the right hand side of the second definition mentions the dictionary variable `d` which must therefore be added as an extra parameter to `g`:

```
f d x y  =  (==) d x y || g d x y
g d x y  =  not (f d x y)
```

In other words, if dictionary parameters are added to any particular function definition, then each use of that function in another definition will also be require extra dictionary parameters. As a result, the monomorphism restriction has to be applied to the smallest groups of declarations such that any pair of mutually recursive bindings are in the same group.

As the example above shows, if one (or more) of the bindings in a given declaration group is affected by the monomorphism restriction so that the appropriate dictionary parameters cannot be added as parameters for that definition, then the same condition must also be imposed on all of the other bindings in the group. (Adding the extra parameter to `f` in the example forces us to add an extra parameter for `g`; if extra parameters were not permitted for `g` then they could not be added to `f`.)

Restricted bindings: There are three main reasons for avoiding adding dictionary parameters to a particular value binding:

- Dictionary parameters unnecessary. If the dictionary values are completely determined by context then it is not necessary to pass the appropriate values as dictionary parameters. For example, the function definition:

  ```
  f x  =   x == 0  ||  x == 2
  ```

  can be translated as:

  ```
  f x  =   (==) {dict} x 0  ||  (==) {dict} x 2
  ```

  where, in both cases, the symbol `{dict}` denotes the dictionary for `Eq Int`. As a further optimisation, once the dictionary is fully determined, this can be simplified to:

  ```
  f x  =   primEqInt x 0 || primEqInt x 2
  ```

- Dictionary parameters cannot be added in a pattern binding. One potential solution to this problem would be to replace the pattern binding by an equivalent set of function bindings. In practice, we do not use this technique because it typically causes ambiguity problems, as illustrated by the pattern binding:

  ```
  (plus,times) = ((+), (*))
  ```

  Translating this into a group of function bindings gives:

  ```
  newVariable  = ((+), (*))
  plus         = fst newVariable     -- fst (x,_) = x
  times        = snd newVariable     -- snd (_,y) = y
  ```

  The type of `newVariable` is `(Num a, Num b) => (a->a->a, b->b->b)` so that the correct translation of these bindings using two dictionary variables gives:

  ```
  newVariable da db = ((+) da, (*) db)
  plus da db        = fst (newVariable da db)
  times da db       = snd (newVariable da db)
  ```

and hence the correct types for `plus` and `times` are:

```
plus  :: (Num a, Num b) => a -> a -> a
times :: (Num a, Num b) => b -> b -> b
```

both of which are ambiguous.

- Adding dictionary parameters may translate a variable definition into a function definition, loosing the benefits of shared evaluation. As an example, consider the following definition using the function `size` and the class `Finite` described in the previous section:

  ```
  twiceSize x = n + n  where n = size x
  ```

  Since the variable n is defined using a local definition, we would not expect to have to evaluate `size x` more than once to determine the value of twiceSize. However, adding extra dictionary parameters without restriction gives:

  ```
  twiceSize d x  = n d + n d  where  n d = size d x
  ```

  Now that `n` has been replaced by a function, the evaluation will be repeated, once for each occurrence of the expression `n d`. In order to avoid this kind of problem, the monomorphism restriction does not usually allow extra parameters to be added to a variable definition. Thus the original definition above will be translated to give:

  ```
  twiceSize d x  =  n + n  where n = size d x
  ```

  Note that the same rule is applied to variable definitions at the top-level of a file of definitions, resulting in an error if any dictionary parameters are required for the right hand side of the definition. As an example of this:

  ```
  twiceMembers = members ++ members
  ```

  which produces an error message of the form:

  ```
  ERROR "ex" (line 157): Unresolved top-level overloading
  *** Binding             : twiceMembers
  *** Inferred type       : [_7]
  *** Outstanding context : Finite _7
  ?
  ```

  (A type expression of the form `_n` (such as `_7` in this particular example) represents a fixed (i.e. monomorphic) type variable.)

  In the case of a variable declaration, the monomorphism restriction can be overcome by giving an explicit type signature including an appropriate context, to indicate that the variable defined is intended to be used as an overloaded value. In this case, we need only include the declaration:

  ```
  twiceMembers :: Finite a => [a]
  ```

  in the file containing the definition for `twiceMembers` to suppress the previous error message and allow the function to be used as a fully overloaded variable.

  Note that the monomorphism restriction interferes with the use of polymorphism. For example, the definition:

  ```
  aNumber = length (twiceMembers::[Bool]) +
            length (twiceMembers::[(Bool,Bool)])
            where twiceMembers = members ++ members
  ```

  will not be accepted because the monomorphism restriction forces the local definition of `twiceMembers` to be restricted to a single overloading (the dictionary parameter supplied to each use of members must be constant throughout the local definition):

```
ERROR "ex" (line 12): Type error in type signature expression
*** term           : twiceMembers
*** type           : [(Bool,Bool)]
*** does not match : [Bool]
?
```

Once again, this problem can be fixed using an explicit type declaration:

```
aNumber = length (twiceMembers::[Bool]) +
          length (twiceMembers::[(Bool,Bool)])
          where twiceMembers :: Finite a => [a]
                twiceMembers  = members ++ members
```

Formal definition: The examples above describe the motivation for the monomorphism restriction, captured by the following definition:

Dictionary variables will not be used as extra parameters in the definition of a value in a given declaration group G if:

- either: G includes a pattern binding
- or: G includes a variable declaration, but does not include an explicit type signature for any of the variables in the group.

If neither of these conditions hold, then equivalent sets of dictionary parameters will be added to each declaration in the group.

Appendix A

Summary of grammar

This section gives a summary of the grammar for the language used by Gofer. The non-terminals 'interp' and 'module' describe the syntax of expressions that can be entered into the Gofer interpreter and that of files of definitions that can be loaded into Gofer respectively.

The following notational conventions are used in the Grammar which is specified using a variant of BNF:

- nonterminals are set in roman type;
- vertical '|' bars are used to separate alternatives;
- {braces} enclose items which may be repeated zero or more times;
- [brackets] are used for optional items;
- (parentheses) are used for grouping;
- terminal sybols are enclosed in $\boxed{\text{boxes}}$ and are set in `typewriter type`.

The following terminal symbols are used but not defined by the grammar:

varid	identifier beginning with lower case letter as described in section 6
conid	like *varid*, but beginning with upper case letter
varop	operator symbol not beginning with a colon, as described in section 6
conop	constructor function operator, like *varop*, but beginning with a colon character
integer	integer constant, as described in section 7.3
float	floating point constant, as described in section 7.4
char	character constant, as described in section 7.5
string	string constant, as described in section 7.7

Top-level grammar

module	::=	`{` topdecls `}`	module
interp	::=	exp [where]	top-level expression
topdecls	::=	topdecls `;` topdecls	multiple declarations
	|	`data` typeLhs `=` constrs	datatype declaration
	|	`type` typeLhs `=` type	synonym declaration
	|	`infixl` [digit] op {`,` op}	fixity declarations
	|	`infixr` [digit] op {`,` op}	
	|	`infix` [digit] op {`,` op}	
	|	`primitive` prims `::` type	primitive bindings
	|	class	class declaration
	|	inst	instance declaration
	|	decls	value declarations

typeLhs	::=	*conid* {*varid*}	type declaration lhs
constrs	::=	constrs `|` constrs	multiple constructors
	|	type *conop* type	infix constructor
	|	*conid* {type}	constructor
prims	::=	prims `,` prims	multiple bindings
	|	var *string*	primitive binding

Type expressions

sigType	::=	[context `=>`]type	[qualified] type
context	::=	`(` [pred {`,` pred}] `)`	general form
	|	pred	singleton context
pred	::=	*conid* type {type}	predicate
type	::=	ctype [`->` type]	function type
ctype	::=	*conid* {atype}	datatype or synonym
	|	atype	
atype	::=	*varid*	type variable
	|	`()`	unit type
	|	`(` type `)`	parenthesised type
	|	`(` type `,` type {`,` type} `)`	tuple type
	|	`[` type `]`	list type

Class and instance declarations

class	::=	`class` [context `=>`] pred [cbody]	
cbody	::=	`where` `{` cdecls `}`	class body
cdecls	::=	cdecls `;` cdecls	multiple declarations
	|	var {`,` var} `::` type	member functions
	|	fun rhs [where]	default bindings
inst	::=	`instance` [context `=>`] pred [ibody]	
ibody	::=	`where` `{` idecls `}`	instance body
idecls	::=	idecls `;` idecls	multiple declarations
	|	fun rhs [where]	member definition

Value declarations

decls	::=	decls `;` decls	multiple declarations
	|	var {`,` var} `::` sigType	type declaration
	|	fun rhs [where]	function binding
	|	pat rhs [where]	pattern binding
rhs	::=	`=` exp	simple right hand side
	|	gdRhs {gdRhs}	guarded right hand sides
gdRhs	::=	`|` exp `=` exp	guarded right hand side
where	::=	`where` `{` decls `}`	local definitions
fun	::=	var	function of arity 0
	|	pat varop pat	infix operator
	|	`(` pat varop `)`	section-like notation
	|	`(` varop pat `)`	
	|	fun apat	function with argument
	|	`(` fun `)`	parenthesised lhs

Expressions

exp	::=	`\` apat {apat} `->` exp	lambda expression
	|	`let` `{` decls `}` `in` exp	local definition
	|	`if` exp `then` exp `else` exp	conditional expression
	|	`case` exp `of` `{` alts `}`	case expression
	|	opExp `::` sigType	typed expression
	|	opExp	
opExp	::=	opExp op opExp	operator application
	|	pfxExp	
pfxExp	::=	`-` appExp	negation
	|	appExp	
appExp	::=	appExp atomic	function application
	|	atomic	
atomic	::=	var	variable
	|	conid	constructor
	|	*integer*	integer literal
	|	*float*	floating point literal
	|	*char*	character literal
	|	*string*	string literal
	|	`()`	unit element
	|	`(` exp `)`	parenthesised expr.
	|	`(` exp op `)`	sections
	|	`(` op exp `)`	
	|	`[` list `]`	list expression
	|	`(` exp `,` exp {`,` exp} `)`	tuple
list	::=	[exp {`,` exp}]	enumerated list
	|	exp `|` quals	list comprehension
	|	exp `..`	arithmetic sequence
	|	exp `,` exp `..`	
	|	exp `..` exp	
	|	exp `,` exp `..` exp	
quals	::=	quals `,` quals	multiple qualifiers
	|	pat `<-` exp	generator
	|	pat `=` exp	local definition
	|	exp	boolean guard
alts	::=	alts `,` alts	multiple alternatives
	|	pat altRhs [where]	alternative
altRhs	::=	`->` exp	single alternative
	|	gdAlt gdAlt	guarded alternatives
gdAlt	::=	`|` exp `->` exp	guarded alternative

Patterns

pat	::=	pat conop pat	operator application
	|	var `+` *integer*	$(n+k)$ pattern
	|	appPat	
appPat	::=	appPat apat	application
	|	apat	
apat	::=	var	variable
	|	var `@` pat	as pattern
	|	`~` pat	irrefutable pattern
	|	`_`	wildcard
	|	conid	constructor
	|	*integer*	integer literal
	|	*char*	character literal
	|	*string*	string literal
	|	`()`	unit element
	|	`(` pat `)`	parenthesised expr.
	|	`(` pat conop `)`	sections
	|	`(` conop pat `)`	
	|	`[` [pat {`,` pat}] `]`	list
	|	`(` pat `,` pat {`,` pat} `)`	tuple

Variables and operators

var	::=	varid	
	|	`(-)`	variable
op	::=	varop	
	|	conop	
	|	`-`	operator
varid	::=	*varid*	
	|	`(` *varop* `)`	variable identifier
varop	::=	*varop*	
	|	`` ` `` *varid* `` ` ``	variable operator
conid	::=	*conid*	
	|	`(` *conop* `)`	constructor identifier
conop	::=	*conop*	
	|	`` ` `` *conid* `` ` ``	constructor operator

Appendix B

Contents of standard prelude

```
--          __________   __________   __________   __________   ________
--         /  _______/  /  ____   /  /  _______/  /  _______/  /  ____  \
--        /  / _____   /  /   /  /  /  /______   /  /______   /  /___/  /
--       /  / /_   /  /  /   /  /  /  _______/  /  _______/  /  __   __/
--      /  /___/  /  /  /___/  /  /  /         /  /______   /  /  \  \
--     /_________/  /_________/  /__/         /_________/  /__/    \__\
--
--     Functional programming environment, Version 2.20 (beta-release)
--     Copyright Mark P Jones 1991.
--
--     Standard prelude for use of overloaded values using type classes.
--     Based on the Haskell standard prelude version 1.1.

help = "press :? for a list of commands"
```

Operator precedence table

```
infixl 9 !!
infixr 9 .
infixr 8 ^
infixl 7 *
infix  7 /, `div`, `rem`, `mod`
infixl 6 +, -
infix  5 \\
infixr 5 ++, :
infix  4 ==, /=, <, <=, >=, >
infix  4 `elem`, `notElem`
infixr 3 &&
infixr 2 ||
```

Standard combinators

```
primitive strict "primStrict" :: (a -> b) -> a -> b

const          :: a -> b -> a
const k x       = k

id             :: a -> a
id    x         = x

curry          :: ((a,b) -> c) -> a -> b -> c
curry f a b     =  f (a,b)

uncurry        :: (a -> b -> c) -> (a,b) -> c
```

```
uncurry f (a,b) = f a b

fst             :: (a,b) -> a
fst (x,_)        = x

snd             :: (a,b) -> b
snd (_,y)        = y

fst3            :: (a,b,c) -> a
fst3 (x,_,_)     = x

snd3            :: (a,b,c) -> b
snd3 (_,x,_)     = x

thd3            :: (a,b,c) -> c
thd3 (_,_,x)     = x

(.)         :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x        = f (g x)

flip            :: (a -> b -> c) -> b -> a -> c
flip  f x y      = f y x
```

Boolean functions

```
(&&), (||)      :: Bool -> Bool -> Bool
False && x       = False
True  && x       = x

False || x       = x
True  || x       = True

not             :: Bool -> Bool
not True         = False
not False        = True

and, or         :: [Bool] -> Bool
and              = foldr (&&) True
or               = foldr (||) False

any, all        :: (a -> Bool) -> [a] -> Bool
any p            = or  . map p
all p            = and . map p

otherwise       :: Bool
otherwise        = True
```

Character functions

```
primitive ord "primCharToInt" :: Char -> Int
primitive chr "primIntToChar" :: Int -> Char


isAscii, isControl, isPrint, isSpace            :: Char -> Bool
isUpper, isLower, isAlpha, isDigit, isAlphanum  :: Char -> Bool

isAscii c     =  ord c < 128

isControl c   =  c < ' '    ||  c == '\DEL'
```

```
isPrint c     =  c >= ' '   &&  c <= '~'

isSpace c     =  c == ' '   || c == '\t'  || c == '\n'  || c == '\r'  ||
                               c == '\f'  || c == '\v'

isUpper c     =  c >= 'A'   &&  c <= 'Z'
isLower c     =  c >= 'a'   &&  c <= 'z'
isAlpha c     =  isUpper c  ||  isLower c
isDigit c     =  c >= '0'   &&  c <= '9'
isAlphanum c  =  isAlpha c  ||  isDigit c


toUpper, toLower      :: Char -> Char

toUpper c | isLower c  = chr (ord c - ord 'a' + ord 'A')
          | otherwise  = c

toLower c | isUpper c  = chr (ord c - ord 'A' + ord 'a')
          | otherwise  = c
```

Standard type classes

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y      = not (x == y)

class Eq a => Ord a where
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a

    x <  y           = x <= y && x /= y
    x >= y           = y <= x
    x >  y           = y < x

    max x y | x >= y  = x
            | y >= x  = y
    min x y | x <= y  = x
            | y <= x  = y

class Ord a => Ix a where
    range   :: (a,a) -> [a]
    index   :: (a,a) -> a -> Int
    inRange :: (a,a) -> a -> Bool

class Ord a => Enum a where
    enumFrom       :: a -> [a]              -- [n..]
    enumFromThen   :: a -> a -> [a]         -- [n,m..]
    enumFromTo     :: a -> a -> [a]         -- [n..m]
    enumFromThenTo :: a -> a -> a -> [a]    -- [n,n'..m]

    enumFromTo n m        = takeWhile (m>=) (enumFrom n)
    enumFromThenTo n n' m = takeWhile ((if n'>=n then (>=) else (<=)) m)
                                      (enumFromThen n n')

class Eq a => Num a where               -- simplified numeric class
    (+), (-), (*), (/) :: a -> a -> a
    negate             :: a -> a
    fromInteger        :: Int -> a
```

Type class instances

```
primitive primEqInt    "primEqInt",
  primLeInt    "primLeInt"   :: Int -> Int -> Bool
primitive primPlusInt  "primPlusInt",
  primMinusInt "primMinusInt",
  primDivInt   "primDivInt",
  primMulInt   "primMulInt"  :: Int -> Int -> Int
primitive primNegInt   "primNegInt"  :: Int -> Int

instance Eq Int  where (==) = primEqInt

instance Ord Int where (<=) = primLeInt

instance Ix Int where
    range (m,n)      = [m..n]
    index (m,n) i    = i - m
    inRange (m,n) i  = m <= i && i <= n

instance Enum Int where
    enumFrom n       = iterate (1+) n
    enumFromThen n m = iterate ((m-n)+) n

instance Num Int where
    (+)          = primPlusInt
    (-)          = primMinusInt
    (*)          = primMulInt
    (/)          = primDivInt
    negate       = primNegInt
    fromInteger x = x

primitive primEqFloat    "primEqFloat",
          primLeFloat    "primLeFloat"    :: Float -> Float -> Bool
primitive primPlusFloat  "primPlusFloat",
          primMinusFloat "primMinusFloat",
          primDivFloat   "primDivFloat",
          primMulFloat   "primMulFloat"   :: Float -> Float -> Float
primitive primNegFloat   "primNegFloat"   :: Float -> Float
primitive primIntToFloat "primIntToFloat" :: Int -> Float

instance Eq Float where (==) = primEqFloat

instance Ord Float where (<=) = primLeFloat

instance Enum Float where
    enumFrom n       = iterate (1.0+) n
    enumFromThen n m = iterate ((m-n)+) n

instance Num Float where
    (+)         = primPlusFloat
    (-)         = primMinusFloat
    (*)         = primMulFloat
    (/)         = primDivFloat
    negate      = primNegFloat
    fromInteger = primIntToFloat

instance Eq Char where c == d  =  ord c == ord d

instance Ord Char where c <= d  =  ord c <= ord d
```

```
instance Ix Char where
    range (c,c')      = [c..c']
    index (c,c') ci   = ord ci - ord c
    inRange (c,c') ci = ord c <= i && i <= ord c' where i = ord ci

instance Enum Char where
    enumFrom c        = map chr [ord c ..]
    enumFromThen c c' = map chr [ord c, ord c' ..]

instance Eq a => Eq [a] where
    []     == []     =  True
    []     == (y:ys) =  False
    (x:xs) == []     =  False
    (x:xs) == (y:ys) =  x==y && xs==ys

instance Ord a => Ord [a] where
    []     <= _      =  True
    (_:_)  <= []     =  False
    (x:xs) <= (y:ys) =  x<y || (x==y && xs<=ys)

instance (Eq a, Eq b) => Eq (a,b) where
    (x,y) == (u,v)  =  x==u && y==v

instance Eq Bool where
    True  == True   =  True
    False == False  =  True
    _     == _      =  False
```

Standard numerical functions

```
primitive div    "primDivInt",
          rem    "primRemInt",
          mod    "primModInt"    :: Int -> Int -> Int

subtract  :: Num a => a -> a -> a
subtract   = flip (-)

even, odd :: Int -> Bool
even x     = x `rem` 2 == 0
odd        = not . even

gcd       :: Int -> Int -> Int
gcd x y    = gcd' (abs x) (abs y)
             where gcd' x 0 = x
                   gcd' x y = gcd' y (x `rem` y)

lcm       :: Int -> Int -> Int
lcm _ 0    = 0
lcm 0 _    = 0
lcm x y    = abs ((x `div` gcd x y) * y)

(^)       :: Int -> Int -> Int
x ^ 0      = 1
x ^ (n+1)  = f x n x
             where f _ 0 y = y
                   f x n y = g x n where
                             g x n | even n    = g (x*x) (n`div`2)
                                   | otherwise = f x (n-1) (x*y)
```

```
abs :: Int -> Int
abs x    | x >= 0  = x
         | x <  0  = - x

signum :: Int -> Int
signum x | x == 0  = 0
         | x > 0   = 1
         | x < 0   = -1

sum, product    :: [Int] -> Int
sum              = foldl' (+) 0
product          = foldl' (*) 1

sums, products :: [Int] -> [Int]
sums             = scanl (+) 0
products         = scanl (*) 1
```

Standard list processing functions

```
head             :: [a] -> a
head (x:_)        = x

last             :: [a] -> a
last [x]          = x
last (_:xs)       = last xs

tail             :: [a] -> [a]
tail (_:xs)       = xs

init             :: [a] -> [a]
init [x]          = [x]
init (x:xs)       = x : init xs

(++)             :: [a] -> [a] -> [a]    -- append lists.  Associative with
[]     ++ ys      = ys                   -- left and right identity [].
(x:xs) ++ ys      = x:(xs++ys)

length  :: [a] -> Int           -- calculate length of list
length            = foldl' (\n _ -> n+1) 0

(!!)             :: [a] -> Int -> a      -- xs!!n selects the nth element of
(x:_)  !! 0       = x                    -- the list xs (first element xs!!0)
(_:xs) !! (n+1)   = xs !! n              -- for any n < length xs.

iterate          :: (a -> a) -> a -> [a] -- generate the infinite list
iterate f x       = x : iterate f (f x)  -- [x, f x, f (f x), ...

repeat           :: a -> [a]             -- generate the infinite list
repeat x          = xs where xs = x:xs   -- [x, x, x, x, ...

cycle            :: [a] -> [a]           -- generate the infinite list
cycle xs          = xs' where xs'=xs++xs'-- xs ++ xs ++ xs ++ ...

copy             :: Int -> a -> [a]      -- make list of n copies of x
copy n x          = take n xs where xs = x:xs

nub              :: Eq a => [a] -> [a]   -- remove duplicates from list
nub []            = []
nub (x:xs)        = x : nub (filter (x/=) xs)
```

```
reverse          :: [a] -> [a]           -- reverse elements of list
reverse           = foldl (flip (:)) []

elem, notElem    :: Eq a => a -> [a] -> Bool
elem              = any . (==)           -- test for membership in list
notElem           = all . (/=)           -- test for non-membership

maximum, minimum :: Ord a => [a] -> a
maximum           = foldl1 max          -- max element in non-empty list
minimum           = foldl1 min          -- min element in non-empty list

concat           :: [[a]] -> [a]        -- concatenate list of lists
concat            = foldr (++) []

transpose        :: [[a]] -> [[a]]      -- transpose list of lists
transpose         = foldr
                      (\xs xss -> zipWith (:) xs (xss ++ repeat []))
                      []

-- null provides a simple and efficient way of determining whether a given
-- list is empty, without using (==) and hence avoiding a constraint of the
-- form Eq [a].

null             :: [a] -> Bool
null []           = True
null (_:_)        = False

-- (\\) is used to remove the first occurrence of each element in the second
-- list from the first list.  It is a kind of inverse of (++) in the sense
-- that  (xs ++ ys) \\ xs = ys for any finite list xs of proper values xs.

(\\)             :: Eq a => [a] -> [a] -> [a]
(\\)              = foldl del
                    where []     `del` _  = []
                          (x:xs) `del` y
                             | x == y     = xs
                             | otherwise  = x : xs `del` y


-- map f xs applies the function f to each element of the list xs returning
-- the corresponding list of results.  filter p xs returns the sublist of xs
-- containing those elements which satisfy the predicate p.

map              :: (a -> b) -> [a] -> [b]
map f []          = []
map f (x:xs)      = f x : map f xs

filter           :: (a -> Bool) -> [a] -> [a]
filter _ []       = []
filter p (x:xs)
    | p x         = x : xs'
    | otherwise   = xs'
                  where xs' = filter p xs

-- Fold primitives:  The foldl and scanl functions, variants foldl1 and
-- scanl1 for non-empty lists, and strict variants foldl' scanl' describe
-- common patterns of recursion over lists.  Informally:
--
```

```
--  foldl f a [x1, x2, ..., xn]  = f (...(f (f a x1) x2)...) xn
--                               = (...((a `f` x1) `f` x2)...) `f` xn
-- etc...
--
-- The functions foldr, scanr and variants foldr1, scanr1 are duals of these
-- functions:
-- e.g.  foldr f a xs = foldl (flip f) a (reverse xs)  for finite lists xs.

foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []      = z
foldl f z (x:xs)  = foldl f (f z x) xs

foldl1           :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)   = foldl f x xs

foldl'           :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []     =  a
foldl' f a (x:xs) =  strict (foldl' f) (f a x) xs

scanl            :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs      = q : (case xs of
                         []   -> []
                         x:xs -> scanl f (f q x) xs)

scanl1           :: (a -> a -> a) -> [a] -> [a]
scanl1 f (x:xs)   = scanl f x xs

scanl'           :: (a -> b -> a) -> a -> [b] -> [a]
scanl' f q xs     = q : (case xs of
                         []   -> []
                         x:xs -> strict (scanl' f) (f q x) xs)

foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f z []      = z
foldr f z (x:xs)  = f x (foldr f z xs)

foldr1           :: (a -> a -> a) -> [a] -> a
foldr1 f [x]      = x
foldr1 f (x:xs)   = f x (foldr1 f xs)

scanr            :: (a -> b -> b) -> b -> [a] -> [b]
scanr f q0 []     = [q0]
scanr f q0 (x:xs) = f x q : qs
                    where qs@(q:_) = scanr f q0 xs

scanr1           :: (a -> a -> a) -> [a] -> [a]
scanr1 f [x]      = [x]
scanr1 f (x:xs)   = f x q : qs
                    where qs@(q:_) = scanr1 f xs
```

List breaking functions

```
--
--   take n xs       returns the first n elements of xs
--   drop n xs       returns the remaining elements of xs
--   splitAt n xs    = (take n xs, drop n xs)
--
--   takeWhile p xs  returns the longest initial segment of xs whose
--                   elements satisfy p
```

```
--   dropWhile p xs  returns the remaining portion of the list
--   span p xs       = (takeWhile p xs, dropWhile p xs)
--
--   takeUntil p xs  returns the list of elements upto and including the
--                   first element of xs which satisfies p

take                :: Int -> [a] -> [a]
take 0     _         = []
take _     []        = []
take (n+1) (x:xs)    = x : take n xs

drop                :: Int -> [a] -> [a]
drop 0     xs        = xs
drop _     []        = []
drop (n+1) (_:xs)    = drop n xs

splitAt             :: Int -> [a] -> ([a], [a])
splitAt 0     xs     = ([],xs)
splitAt _     []     = ([],[])
splitAt (n+1) (x:xs) = (x:xs',xs'') where (xs',xs'') = splitAt n xs

takeWhile           :: (a -> Bool) -> [a] -> [a]
takeWhile p []       = []
takeWhile p (x:xs)
         | p x       = x : takeWhile p xs
         | otherwise = []

takeUntil           :: (a -> Bool) -> [a] -> [a]
takeUntil p []       = []
takeUntil p (x:xs)
       | p x         = [x]
       | otherwise   = x : takeUntil p xs

dropWhile           :: (a -> Bool) -> [a] -> [a]
dropWhile p []       = []
dropWhile p xs@(x:xs')
         | p x       = dropWhile p xs'
         | otherwise = xs

span, break         :: (a -> Bool) -> [a] -> ([a],[a])
span p []            = ([],[])
span p xs@(x:xs')
         | p x       = let (ys,zs) = span p xs' in (x:ys,zs)
         | otherwise = ([],xs)
break p              = span (not . p)

-- Text processing:
--   lines s     returns the list of lines in the string s.
--   words s     returns the list of words in the string s.
--   unlines ls  joins the list of lines ls into a single string
--               with lines separated by newline characters.
--   unwords ws  joins the list of words ws into a single string
--               with words separated by spaces.

lines     :: String -> [String]
lines ""   = []
lines s    = l : (if null s' then [] else lines (tail s'))
             where (l, s') = break ('\n'==) s
```

```
words     :: String -> [String]
words s    = case dropWhile isSpace s of
                  "" -> []
                  s' -> w : words s''
                        where (w,s'') = break isSpace s'

unlines   :: [String] -> String
unlines    = concat . map (\l -> l ++ "\n")

unwords   :: [String] -> String
unwords [] = []
unwords ws = foldr1 (\w s -> w ++ ' ':s) ws
```

Merging and sorting lists

```
merge               :: Ord a => [a] -> [a] -> [a]
merge []     ys      = ys
merge xs     []      = xs
merge (x:xs) (y:ys)
        | x <= y     = x : merge xs (y:ys)
        | otherwise  = y : merge (x:xs) ys

sort                :: Ord a => [a] -> [a]
sort                 = foldr insert []

insert              :: Ord a => a -> [a] -> [a]
insert x []          = [x]
insert x (y:ys)
        | x <= y     = x:y:ys
        | otherwise  = y:insert x ys

qsort               :: Ord a => [a] -> [a]
qsort []             = []
qsort (x:xs)         = qsort [ u | u<-xs, u<x ] ++
                             [ x ] ++
                       qsort [ u | u<-xs, u>=x ]
```

zip and zipWith families of functions

```
zip  :: [a] -> [b] -> [(a,b)]
zip   = zipWith  (\a b -> (a,b))

zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]
zip3  = zipWith3 (\a b c -> (a,b,c))

zip4 :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
zip4  = zipWith4 (\a b c d -> (a,b,c,d))

zip5 :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a,b,c,d,e)]
zip5  = zipWith5 (\a b c d e -> (a,b,c,d,e))

zip6 :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [(a,b,c,d,e,f)]
zip6  = zipWith6 (\a b c d e f -> (a,b,c,d,e,f))

zip7 :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g] -> [(a,b,c,d,e,f,g)]
zip7  = zipWith7 (\a b c d e f g -> (a,b,c,d,e,f,g))


zipWith                  :: (a->b->c) -> [a]->[b]->[c]
```

```
zipWith z (a:as) (b:bs)   = z a b : zipWith z as bs
zipWith _ _      _        = []

zipWith3                 :: (a->b->c->d) -> [a]->[b]->[c]->[d]
zipWith3 z (a:as) (b:bs) (c:cs)
                          = z a b c : zipWith3 z as bs cs
zipWith3 _ _ _ _          = []

zipWith4                 :: (a->b->c->d->e) -> [a]->[b]->[c]->[d]->[e]
zipWith4 z (a:as) (b:bs) (c:cs) (d:ds)
                          = z a b c d : zipWith4 z as bs cs ds
zipWith4 _ _ _ _ _        = []

zipWith5                 :: (a->b->c->d->e->f) -> [a]->[b]->[c]->[d]->[e]->[f]
zipWith5 z (a:as) (b:bs) (c:cs) (d:ds) (e:es)
                          = z a b c d e : zipWith5 z as bs cs ds es
zipWith5 _ _ _ _ _ _      = []

zipWith6                 :: (a->b->c->d->e->f->g)
                             -> [a]->[b]->[c]->[d]->[e]->[f]->[g]
zipWith6 z (a:as) (b:bs) (c:cs) (d:ds) (e:es) (f:fs)
                          = z a b c d e f : zipWith6 z as bs cs ds es fs
zipWith6 _ _ _ _ _ _ _    = []

zipWith7                 :: (a->b->c->d->e->f->g->h)
                              -> [a]->[b]->[c]->[d]->[e]->[f]->[g]->[h]
zipWith7 z (a:as) (b:bs) (c:cs) (d:ds) (e:es) (f:fs) (g:gs)
                          = z a b c d e f g : zipWith7 z as bs cs ds es fs gs
zipWith7 _ _ _ _ _ _ _ _  = []
```

Formatted output

```
primitive primPrint "primPrint"  :: Int -> a -> String -> String

show'       :: a -> String
show' x      = primPrint 0 x []

cjustify, ljustify, rjustify :: Int -> String -> String

cjustify n s = space halfm ++ s ++ space (m - halfm)
               where m     = n - length s
                     halfm = m `div` 2
ljustify n s = s ++ space (n - length s)
rjustify n s = space (n - length s) ++ s

space       :: Int -> String
space n      = copy n ' '

layn        :: [String] -> String
layn         = lay 1 where lay _ []     = []
                           lay n (x:xs) = rjustify 4 (show n) ++ ") "
                                           ++ x ++ "\n" ++ lay (n+1) xs
```

Miscellaneous

```
until                  :: (a -> Bool) -> (a -> a) -> a -> a
until p f x | p x       = x
            | otherwise = until p f (f x)
```

```
until'                :: (a -> Bool) -> (a -> a) -> a -> [a]
until' p f             = takeUntil p . iterate f

error                 :: String -> a
error msg | False      = error msg

undefined             :: a
undefined | False      = undefined

asTypeOf              :: a -> a -> a
x `asTypeOf` _         = x
```

A trimmed down version of the Haskell Text class

```
type  ShowS   = String -> String

class Text a where
    showsPrec      :: Int -> a -> ShowS
    showList       :: [a] -> ShowS

    showsPrec       = primPrint
    showList []     = showString "[]"
    showList (x:xs) = showChar '[' . shows x . showl xs
                      where showl []     = showChar ']'
                            showl (x:xs) = showChar ',' . shows x . showl xs

shows      :: Text a => a -> ShowS
shows       =  showsPrec 0

show       :: Text a => a -> String
show x      =  shows x ""

showChar   :: Char -> ShowS
showChar    =  (:)

showString :: String -> ShowS
showString  =  (++)

instance Text ()

instance Text Int

instance Text Char where
    showList cs = showChar '"' . showl cs
                  where showl ""       = showChar '"'
                        showl ('"':cs) = showString "\\\"" . showl cs
                        showl (c:cs)   = showChar c . showl cs
-- Haskell has   showLitChar c . showl cs

instance Text a => Text [a]  where
    showsPrec p = showList

instance (Text a, Text b) => Text (a,b) where
    showsPrec p (x,y) = showChar '(' . shows x . showChar ',' .
                                       shows y . showChar ')'
```

I/O functions and definitions

```
stdin         =  "stdin"
```

```
stdout        =  "stdout"
stderr        =  "stderr"
stdecho       =  "stdecho"

data Request  =  -- file system requests:
                ReadFile      String
              | WriteFile     String String
              | AppendFile    String String
                 -- channel system requests:
              | ReadChan      String
              | AppendChan    String String
                 -- environment requests:
              | Echo          Bool

data Response = Success
              | Str String
              | Failure IOError

data IOError  = WriteError   String
              | ReadError    String
              | SearchError  String
              | FormatError  String
              | OtherError   String

type Dialogue  =  [Response] -> [Request]
type SuccCont  =                Dialogue
type StrCont   =  String     -> Dialogue
type FailCont  =  IOError    -> Dialogue

done          ::                                                Dialogue
readFile      :: String ->           FailCont -> StrCont     -> Dialogue
writeFile     :: String -> String -> FailCont -> SuccCont    -> Dialogue
appendFile    :: String -> String -> FailCont -> SuccCont    -> Dialogue
readChan      :: String ->           FailCont -> StrCont     -> Dialogue
appendChan    :: String -> String -> FailCont -> SuccCont    -> Dialogue
echo          :: Bool ->             FailCont -> SuccCont    -> Dialogue

done resps    =  []
readFile name fail succ resps =
     (ReadFile name) : strDispatch fail succ resps
writeFile name contents fail succ resps =
    (WriteFile name contents) : succDispatch fail succ resps
appendFile name contents fail succ resps =
    (AppendFile name contents) : succDispatch fail succ resps
readChan name fail succ resps =
    (ReadChan name) : strDispatch fail succ resps
appendChan name contents fail succ resps =
    (AppendChan name contents) : succDispatch fail succ resps
echo bool fail succ resps =
    (Echo bool) : succDispatch fail succ resps

strDispatch fail succ (resp:resps) =
            case resp of Str val     -> succ val resps
                         Failure msg -> fail msg resps

succDispatch fail succ (resp:resps) =
            case resp of Success     -> succ resps
                         Failure msg -> fail msg resps
```

```
abort           :: FailCont
abort err        = done

exit            :: FailCont
exit err         = appendChan stdout msg abort done
                   where msg = case err of ReadError s   -> s
                                           WriteError s  -> s
                                           SearchError s -> s
                                           FormatError s -> s
                                           OtherError s  -> s

print           :: Text a => a -> Dialogue
print x          = appendChan stdout (show x) abort done

prints          :: Text a => a -> String -> Dialogue
prints x s       = appendChan stdout (shows x s) abort done

interact :: (String -> String) -> Dialogue
interact f  = readChan stdin abort
    (\x -> appendChan stdout (f x) abort done)

run :: (String -> String) -> Dialogue
run f  = echo False abort (interact f)
```

Appendix C

Relationship with Haskell 1.1

The language supported by Gofer is both syntactically and semantically similar to that of the functional programming language Haskell as defined in the report for Haskell version 1.1 [5]. This section details the differences between the two languages, outlined briefly in section 2.

Haskell features not included in Gofer:

- Modules
- Arrays
- Derived instances for standard classes – the ability to construct instances of particular classes automatically.
- Default mechanism for eliminating unresolved overloading involving numeric and standard classes. Since Gofer is an experimental system, it can be used with a range of completely different prelude files; there is no concept of 'standard classes'.
- Overloaded numeric constants. In the absence of a defaulting mechanism as mentioned in the previous item, problems with unresolved overloading make implicitly typed programming involving numeric constants impractical in an interpreter based system.
- Full range of numeric types and classes. Gofer has only two primitive numeric types `Int` and `Float` (the second of which is not supported in the PC version). Although is would be possible to modify the standard prelude so that Gofer uses the same class hierarchy as Haskell, this is unnecessarily sophisticated for the intended uses of Gofer.
- Datatype definitions in Haskell may involve class constraints such as:

  ```
  data  Ord a => Set a = Set [a]
  ```

  It is not clear how such constraints should be interpreted (particularly in the light of the extended form of constraints used by Gofer) in such a way to make them useful whilst avoiding unwanted ambiguity problems.

Gofer features not supported in Haskell:

- Type classes may have multiple parameters.
- Predicates in type expressions may involve arbitrary type expressions, not just type variables as used in Haskell.
- Instances of type classes can be defined at non-overlapping, but otherwise arbitrary types, as described in section 14.2.5.
- List comprehensions may include local definitions, specified by qualifiers of the form `pat=expr` as described in section 10.2.

- No restrictions are placed on the form of predicates that appear in the context for a class or instance declaration. This has a number of consequences, including the possibility of using (mutually) recursive groups of dictionaries, but means that decidability of the predicate entailment relation may be lost. This is not a great problem in practice, since all dictionary construction is performed before evaluation and supposedly non-terminating dictionary constructions will actually generate an error due to the limited amount of space available for holding dictionaries (see section 14.4.2).

Other differences:

- Whilst superficially similar the approach to type classes in Gofer is quite different from that used in Haskell. In particular, the approach used in Gofer ensures that all necessary dictionaries are constructed before the evaluation of an expression begins, rather than being built (possibly several times) during the evaluation as is the case with Haskell. See section 14 and reference [11] for further details.
- Input/Output facilities - Gofer supports only a subset of the requests available in Haskell. In principle, it should not be too difficult to add most of the remaining forms of request (with the exception of those associated with binary files) to Gofer. The principal motivation for including the I/O facilities in Gofer was to make it possible to experiment with simple interactive programs.
- In Gofer, unary minus has greater precedence than any operator symbol, but lower than that of function application. In Haskell, the precedence of unary minus is the same as that of the infix (subtraction) operator of the same name.
- In Haskell, the character '`-`' can only be used as the first character of an operator symbol. In Gofer, this character may appear in any position in an operator (except for symbols beginning with `--`, which indicates the start of a comment). The only problems that I am aware of with this is that a lambda expression such as `\-2->2` will be parsed as such by a Haskell system, but cause a syntax error in Gofer. This form of lambda expression is sufficiently unusual that I do not believe this will cause any problems in practice; in any case, the parsing problem can be solved by inserting a space: `\ -2->2`.
- Pattern bindings are not currently permitted in either instance or class declarations. This restriction has been made simply for ease of implementation, is not an inherent problem with the type class system and is likely to be relaxed in later versions of Gofer if appropriate. I have yet to see any examples in which the lack of pattern bindings in class and instance declarations causes any kind of deficiency.
- Qualified type signatures are not permitted for the member functions in Gofer class declarations. Once again, this restriction was made for ease of implementation rather than any pressing technical issues. It is likely that this restriction will be relaxed in future versions of Gofer, although I am not convinced that proper use can be made of such member functions without some form of nested instance declarations (yuk!).
- The definition of the class Text given in the standard prelude does not include the Haskell functions for reading/parsing values from strings; the only reason for omitting these functions was to try to avoid unnecessary complexity in the standard prelude. The standard prelude can be modified to include the appropriate additional definitions if these are required.

Known problems in Gofer:

- The null escape sequence `"\&"` is not generated in the printable representations of strings produced by both the primitive function primPrint (used to implement the `show'` function) and the version of show defined in the standard prelude. This means that certain strings values are not printed correctly e.g. `show' "\245\&123"` produces the string `"\245123"`. This is unlikely to cause too many problems in practice.
- Unification of a type variable a with a type expression of the form `T a` where `T` is a synonym name whose expansion does not involve a will fail. It is not entirely clear whether this behaviour is correct or not.
- Formfeeds `"\f"` and vertical tabs `"\v"` are not treated as valid whitespace characters in the way suggested by the Haskell report.

- Inability to recover from program stack overlow errors in some situations. This problem only affects the PC implementation of Gofer.

- Implementation of `ReadFile` may lose referential transparency; the response to a particular `ReadFile` request may be affected by a later `WriteFile` or `AppendFile` request for the same file. Whilst this problem can be solved for Unix based implementations, I have not yet found a portable solution suitable for all of the systems on which Gofer can be used.

Areas for possible future improvement:

- Relaxing the restriction on type synonyms in predicates.

- General purpose automatic default mechanism for eliminating certain forms of unresolved overloading.

- Improved checking and use of superclass and instance constraints during static analysis and type checking.

- Simple facility to force dictionary construction at load-time.

- Provision for shell escapes :! etc within the Gofer interpreter.

- Debugging facilities, including breakpoints and tracing from within interpreter.

- Separate interpreter and compiler programs for creating standalone applications using Gofer.

Appendix D

Using Gofer with Bird&Wadler

Bird and Wadler's textbook [1] gives an excellent introduction to functional programming, providing an insight into both basic techniques and matters of programming style as well as describing the underlying mathematics and its use for program development and derivation. Most of the programs in that book can be used with Gofer although there are a number of differences between the two notations. Fortunately, it is not difficult to translate from one notation to the other. The following points are particularly useful for this:

- Type constructors in Gofer begin with capital letters (e.g. `Bool`, `Char` etc.) where lower case is used in [1] (e.g. `bool`, `char`, etc.). Note that Gofer has no general numeric type `num` as used in [1]; Use either `Int`, `Float`, or overloading in Gofer as appropriate.

- Datatype definitions in [1] are written in the form `lhs::=constrs`. The equivalent definition in Gofer is: `data lhs = constrs`.

  Similarly, a type synonym definition in [1] of the form `lhs == rhs` can be written in Gofer as: `type lhs = rhs`.

- The differences between the syntax used for guarded equations in Gofer compared with the notation used in [1] have already been discussed in section 9.2. For example:

  Using the notation of [1]:

  ```
  filter p (x:xs)
    = x : filter p xs, if p x
    = filter p xs,     otherwise
  ```

  Using Gofer:

  ```
  filter p (x:xs)
     | p x       = x : filter p xs
     | otherwise = filter p xs
  ```

- In Gofer, list comprehension qualifiers are separated by commas rather than semicolons as used in [1].

- A number of the function names and types in the standard prelude are different:

[1]	Gofer	[1]	Gofer
`(#)`	`length`	`takewhile`	`takeWhile`
`(~)`	`not`	`dropwhile`	`dropWhile`
`(/\)`	`(&&)`	`zipwith`	`zipWith`
`(\/)`	`(||)`	`swap`	`flip`
`(!)`	`(!!)`	`in`	`elem`
`(--)`	`(\\)`	`scan`	`scanl`
`hd`	`head`	`some`	`any`
`tl`	`tail`	`listmin`	`minimum`
`decode`	`chr`	`listmax`	`maximum`
`code`	`ord`		

  See appendix B for a complete list of standard functions in Gofer.

  The version of `foldl` using `strict` which appears in [1] is available in Gofer as the function `foldl'`. The role of `zip` and `zipwith` in [1] is filled by the `zip` and `zipWith` families of functions in Gofer. An expression of the form `zip (xs,ys)` in [1] is equivalent to `zip xs ys` in Gofer etc.

- Gofer does not enforce the condition assumed in [1] that the left hand sides of each of the equations defining a function must be disjoint.

- The equality operator in Gofer is written as `==` and the single equality character `=` is a reserved symbol used to separate left and right hand sides of equations. Many C programmers will be familiar with this kind of notation (together with the kinds of problems it can create!).

- Some of the identifiers used in [1] are reserved words in Gofer. Examples that are particularly likely to occur include `in` and `then`.

Appendix E

Primitives

Warning: the features described in this appendix are typically only needed when alternative versions of the standard prelude are created. These features should only be used by expert users; misuse may lead to failure and runtime errors in the Gofer interpreter. It is not usually a good idea to use primitive functions directly in your programs.

A number of primitive functions are builtin to the Gofer interpreter, and may be bound to function symbols using a declaration of the form:

```
primitive name1 code1, name2 code2, ...., namen coden :: type
```

where each name is an identifier (or an operator symbol enclosed by parentheses) and each code is a string literal taken from the table below. The type specified to the right of the `::` symbol must be a valid type for the functions being defined – *warning: gofer does not attempt to check for suitability of the declared type.* The following definition, taken from the standard prelude, illustrates the use of this feature to bind a function named `primPrint` to the primitive function with code name string `"primPrint"` and type `Int -> a ->  String -> String`:

```
primitive primPrint "primPrint"  :: Int -> a -> String -> String
```

The primitive functions currently available are:

- integer arithmetic:

```
primPlusInt         Int -> Int -> Int
primMinusInt        Int -> Int -> Int
primMulInt          Int -> Int -> Int
primDivInt          Int -> Int -> Int
primModInt          Int -> Int -> Int
primRemInt          Int -> Int -> Int
primNegInt          Int -> Int -> Int
```

- floating point arithmetic:

```
primPlusFloat       Float -> Float -> Float
primMinusFloat      Float -> Float -> Float
primMulFloat        Float -> Float -> Float
primDivFloat        Float -> Float -> Float
primNegFloat        Float -> Float -> Float
```

- coercion functions:

```
primIntToChar       Int -> Char  -- chr in the standard prelude
primCharToInt       Char -> Int  -- ord in the standard prelude
primIntToFloat      Int -> Float -- implements fromInteger
```

- equality and $\leq$ primitives:

```
primEqInt           Int -> Int -> Bool
primLeInt           Int -> Int -> Bool
primEqFloat         Float -> Float -> Bool
primLeFloat         Float -> Float -> Bool
```

- generic ordering primitives:

```
primGenericEq       a -> a -> Bool
primGenericNe       a -> a -> Bool
primGenericGt       a -> a -> Bool
primGenericLe       a -> a -> Bool
primGenericGe       a -> a -> Bool
primGenericLt       a -> a -> Bool
```

  These functions implement the standard generic (i.e. non overloaded) ordering primitives. They are not currently used in the standard prelude. A simplified prelude may be created by binding the standard operator symbols `(==)`, `(/=)`, `(>)`, `(<=)`, `(>=)` and `(<)` to these functions respectively.

- output:

```
primPrint           Int -> a -> String -> String
```

  This function is used to implement the `show'` function in the standard prelude and is not usually used directly.

  `primPrint d e s` produces a textual representation of the value of the expression `e` as a string, followed by the string `s`. The integer parameter `d` is used as an indicator of the current precedence level. The `primPrint` function is the standard method of printing the value of an expression whose type is not equivalent to the type `String` used by the top-level of the Gofer interpreter.

- sequencing:

```
primStrict          (a -> b) -> a -> b
```

  The `"primStrict"` function (bound to the identifier `strict` in the standard prelude) forces the evaluation of its second argument before the function supplied as the first argument is applied to it. See section 9.4 for an illustration.

Appendix F

Interpreter command summary

- *expr*

  Analyse expression for errors, typecheck and evaluate. If the expression has type `Dialogue`, execute as a program using the I/O facilities as described in section 12. If the expression has type `String`, evaluate and print result as a lazy list of characters. In any other case, the standard prelude function `show'` is applied to the expression and used to print the value of the result in the form of a string, as in the previous case.

- `:t` *expr*
  `:type` *expr*
  `:T`

  Analyse expression for errors, typecheck and print the translation and inferred type of the term.

- `:q`
  `:quit`
  `:Q`

  Exit Gofer interpreter.

- `:?`
  `:h`
  `:H`

  Display summary of interpreter commands.

- `:l` $f_1 \ldots f_n$
  `:load`
  `:L`

  Removes any previously loaded files of definitions and attempts to load the contents of the files f_1 upto f_n one after the other. If no filenames are provided, only those functions and values defined in the standard prelude will still be be available.

- `:a` $f_1 \ldots f_n$
  `:also`
  `:A`

  Load the contents of the files f_1 upto f_n in addition to any previously loaded files. If any of the files of definitions which have already been loaded have been modified since they were last read then they are automatically reloaded before any of the files f_1 upto f_n are read.

  If successful, a command of the form `:l f1 ... fn` is equivalent to the sequence of commands: `:l`, `:a f1` ... `:a fn`.

- `:r`
  `:reload`
  `:R`

  Repeat the last load command, attempting to reload any files which have subsequently been modified. Since later files may depend on the definitions in earlier ones, once one file has been reloaded, all subsequent files will also need to be reloaded.

- `:e` *file*
  `:edit`
  `:E`

  Suspend current Gofer session and start an editor program to modify or view the named file. The Gofer session will be resumed when the editor program terminates, and any script files that have been changed will be reloaded automatically.

  Note that a separate editor program is required and that Gofer must be properly installed to use this feature. The default editor is usually `vi` (Calvin version 2.0 is a good substitute for a PC), although this may have been changed when your system was installed. In any case, you can always substitute an editor of your choice by setting the environment variable `EDITOR` to the name of your favourite editor program.

  There are a number of factors which will affect your choice of editor. On a slow machine, with only a limited amount of memory, you will probably need to choose a relatively small editor which can be loaded reasonably quickly and does not require too much memory. On a more powerful system, you may find it more convenient to use Gofer from a window based environment, running your editor in one window with Gofer in another.

  Using the `:e` command without specifying a particular file to be edited starts up an editor program as described above either for the file of definitions most recently loaded into Gofer or, if an error occurred whilst loading a file of definitions, for the file of definitions in which the error was last detected.

  With many editor programs, it is even possible to start the editor at the line where the error occurred. As before, it is possible to change the default behaviour of Gofer in this case by setting the environment variable `EDITLINE` to a command string which can be used to start the editor program with a given file at a specific line number. The positions in the string at which the file name and line number values should be inserted should be indicated by the strings `"%s"` and `"%d"` respectively, and may appear in either order. The default command string, which is used if `EDITLINE` is not set is `"vi +%d %s"`.

Appendix G

Bibliography

1. Richard Bird and Philip Wadler, *Introduction to functional programming*, Prentice Hall International, 1989.

2. Simon L. Peyton Jones, The Implementation of functional programming languages, Prentice Hall International, 1987.

3. Thomas Johnsson, Lambda Lifting: Transforming Programs to Recursive Equations, in Lecture Notes in Computer Science 201, Springer Verlag, 1985. [but try to get a copy of the version of this paper included in Johnsson's thesis which benefits from an extended typeface and is a little easier to read!]

4. Philip Wadler and Stephen Blott, How to make ad-hoc polymorphism less ad-hoc, in the proceedings of the 16th ACM annual symposium on Principles of Programming Languages, Austin, Texas, January 1989.

5. Paul Hudak, Philip Wadler et al., Report on the programming language Haskell, a non-strict purely functional language (Version 1.1), Technical report Yale University/Glasgow University. August, 1991.

6. Philip Wadler and Quentin Miller, *Introduction to Orwell 6.00*, University of Oxford, 1990.

7. Lennart Augustsson and Thomas Johnsson, *Lazy ML user's manual*, 1990.

8. Mark P. Jones, *Computing with lattices: An application of type classes*, Technical report PRG-TR-11-90, Programming Research Group, Oxford University Computing Laboratory, June 1990.

9. Mark P. Jones, *Towards a theory of qualified types*, Technical report PRG-TR-6-91, Programming Research Group, Oxford University Computing Laboratory, April 1991.

10. Mark P. Jones, *Type inference for qualified types*, Technical report PRG-TR-10-91, Programming Research Group, Oxford University Computing Laboratory, June 1991.

11. Mark P. Jones, *A new approach to type classes*, distributed to Haskell mailing list 1991.

12. Mark P. Jones, *Practical issues in the implementation of qualified types*, Forthcoming 1991.