4

Defining functions

In this chapter we introduce a range of mechanisms for defining functions in Haskell. We start with conditional expressions and guarded equations, then introduce the simple but powerful idea of pattern matching, and conclude with the concepts of lambda expressions and operator sections.

4.1 New from old

Perhaps the most straightforward way to define new functions is simply by combining one or more existing functions. For example, a few library functions that can be defined in this way are shown below.

- Decide if an integer is even:

```
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0
```

- Split a list at the nth element:

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)
```

- Reciprocation:

```
recip :: Fractional a => a -> a
recip n = 1/n
```

Note the use of the class constraints in the types for `even` and `recip` above, which make precise the idea that these functions can be applied to numbers of any integral and fractional types, respectively.

4.2 Conditional expressions

Haskell provides a range of different ways to define functions that choose between a number of possible results. The simplest are *conditional expressions*, which use a logical expression called a *condition* to choose between two results of the same type. If the condition is `True`, then the first result is chosen, and if it is `False`, then the second result is chosen. For example, the library function `abs` that returns the absolute value of an integer can be defined as follows:

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

Conditional expressions may be nested, in the sense that they can contain other conditional expressions. For example, the library function `signum` that returns the sign of an integer can be defined as follows:

```
signum :: Int -> Int
signum n = if n < 0 then -1 else
               if n == 0 then 0 else 1
```

Note that unlike in some programming languages, conditional expressions in Haskell must always have an `else` branch, which avoids the well-known *dangling else* problem. For example, if `else` branches

were optional, then the expression `if True then if False then 1 else 2` could either return the result `2` or produce an error, depending upon whether the single `else` branch was assumed to be part of the inner or outer conditional expression.

4.3 Guarded equations

As an alternative to using conditional expressions, functions can also be defined using *guarded equations*, in which a sequence of logical expressions called *guards* is used to choose between a sequence of results of the same type. If the first guard is `True`, then the first result is chosen; otherwise, if the second is `True`, then the second result is chosen, and so on. For example, the library function `abs` can also be defined using guarded equations as follows:

```
abs n | n >= 0    = n
      | otherwise = -n
```

The symbol `|` is read as *such that*, and the guard `otherwise` is defined in the standard prelude simply by `otherwise = True`. Ending a sequence of guards with `otherwise` is not necessary, but provides a convenient way of handling all other cases, as well as avoiding the possibility that none of the guards in the sequence is `True`, which would otherwise result in an error.

The main benefit of guarded equations over conditional expressions is that definitions with multiple guards are easier to read. For example, the library function `signum` is easier to understand when defined as follows:

```
signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

4.4 Pattern matching

Many functions have a simple and intuitive definition using *pattern matching*, in which a sequence of syntactic expressions called *patterns* is used to choose between a sequence of results of the same type. If the first pattern is *matched*, then the first result is chosen; otherwise, if the second is matched, then the second result is chosen, and so on. For example, the library function `not` that returns the negation of a logical value can be defined as follows:

```
not :: Bool -> Bool
not False = True
not True  = False
```

Functions with more than one argument can also be defined using pattern matching, in which case the patterns for each argument are matched in order within each equation. For example, the library operator `&&` that returns the conjunction of two logical values can be defined as follows:

```
(&&) :: Bool -> Bool -> Bool
True && True   = True
True && False  = False
False && True  = False
False && False = False
```

However, this definition can be simplified by combining the last three equations into a single equation that returns `False` independent of the values of the two arguments, using the *wildcard pattern* `_` that matches any value:

```
True && True = True
_    && _    = False
```

This version also has the benefit that, under lazy evaluation as discussed in chapter 15, if the first argument is `False`, then the result `False` is returned without the need to evaluate the second argument. In practice, the prelude defines `&&` using equations that have this same property, but make the choice about which equation applies using the value of the first argument only:

```
True && b  = b
False && _ = False
```

That is, if the first argument is `True`, then the result is the value of the second argument, and, if the first argument is `False`, then the result is `False`.

Note that Haskell does not permit the same name to be used for more than one argument in a single equation. For example, the following definition for the operator `&&` is based upon the observation that, if the two logical arguments are equal, then the result is the same value, otherwise the result is `False`, but is invalid because of the above naming requirement:

```
b && b = b
_ && _ = False
```

If desired, however, a valid version of this definition can be obtained by using a guard to decide if the two arguments are equal:

```
b && c | b == c    = b
       | otherwise = False
```

So far, we have only considered basic patterns that are either values, variables, or the wildcard pattern. In the remainder of this section we introduce two useful ways to build larger patterns by combining smaller patterns.

Tuple patterns

A tuple of patterns is itself a pattern, which matches any tuple of the same arity whose components all match the corresponding patterns in order. For example, the library functions `fst` and `snd` that respectively select the first and second components of a pair are defined as follows:

```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y
```

List patterns

Similarly, a list of patterns is itself a pattern, which matches any list of the same length whose elements all match the corresponding patterns in order. For example, a function `test` that decides if a list contains

precisely three characters beginning with the letter `'a'` can be defined as follows:

```
test :: [Char] -> Bool
test ['a',_,_] = True
test _         = False
```

Up to this point, we have viewed lists as a primitive notion in Haskell. In fact they are not primitive as such, but are constructed one element at a time starting from the empty list `[]` using an operator `:` called *cons* that *cons*tructs a new list by prepending a new element to the start of an existing list. For example, the list `[1,2,3]` can be decomposed as follows:

```
  [1,2,3]
=     { list notation }
  1 : [2,3]
=     { list notation }
  1 : (2 : [3])
=     { list notation }
  1 : (2 : (3 : []))
```

That is, `[1,2,3]` is just an abbreviation for `1:(2:(3:[]))`. To avoid excess parentheses when working with such lists, the cons operator is assumed to associate to the right. For example, `1:2:3:[]` means `1:(2:(3:[]))`.

As well as being used to construct lists, the cons operator can also be used to construct patterns, which match any non-empty list whose first and remaining elements match the corresponding patterns in order. For example, we can now define a more general version of the function `test` that decides if a list containing any number of characters begins with the letter `'a'`:

```
test :: [Char] -> Bool
test ('a':_) = True
test _       = False
```

Similarly, the library functions `head` and `tail` that respectively select and remove the first element of a non-empty list are defined as follows:

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
```

Note that cons patterns must be parenthesised, because function application has higher priority than all other operators in the language. For example, the definition `head x:_ = x` without parentheses means `(head x):_ = x`, which is both the incorrect meaning and an invalid definition.

4.5 Lambda expressions

As an alternative to defining functions using equations, functions can also be constructed using *lambda expressions*, which comprise a pattern for each of the arguments, a body that specifies how the result can be calculated in terms of the arguments, but do not give a name for the function itself. In other words, lambda expressions are nameless functions.

For example, the nameless function that takes a single number `x` as its argument, and produces the result `x + x`, can be constructed as follows: