

# Haskell Cheat Sheet

## Strings

- "abc" – Unicode string, sugar for
- ['a', 'b', 'c'].
- ,a' – Single character.

This cheat sheet lays out the fundamental elements of the Haskell language: syntax, keywords and other elements. It is presented as both an executable Haskell file and a printable document. Load the source into your favorite interpreter to play with code samples shown.

## Basic Syntax

**Multi-line Strings** Normally, it is a syntax error if a string has any newline characters. That is, this is a syntax error:

```
string1 = "My long  
string."
```

Backslashes ('\\') can "escape" a newline:

```
string1 = "My long \\  
\\string."
```

A single line comment starts with '--' and extends to the end of the line. Multi-line comments start with '{-' and extend to '-}' . Comments can be nested.

Comments above function definitions should start with '{-' '!' and those next to parameter types with '-- ~' for compatibility with Haddock, a system for documenting Haskell code.

The area between the backslashes is ignored. Newlines in the string must be represented explicitly:

```
string2 = "My long \\n\\  
\\string."
```

That is, string1 evaluates to:

```
My long string.
```

While string2 evaluates to:

```
My long  
string.
```

The following words are reserved in Haskell. It is a syntax error to give a variable or a function one of these names.

- case
- import
- of
- module
- in
- newtype
- infix
- then
- deriving
- infixl
- type
- do
- instance
- where
- else
- let
- if

- '\\ – A "null" escape character which allows numeric escape codes next to numeric literals. For example, '\x2C4 is ^ (in Unicode) while '\x2C1&4 is ,4. This sequence cannot be used in character literals.

## Numbers

- 1 – Integer or floating point value.
- 1.0, 1e10 – Floating point value.
- 0o1, 001 – Octal value.
- 0x1, 0X1 – Hexadecimal value.
- -1 – Negative number; the minus sign ("–") cannot be separated from the number.

## Enumerations

- [1..10] – List of numbers –1, 2, ..., 10.
- [100..] – Infinite list of numbers – 100, 101, 102, ... .
- [110..100] – Empty list, but [110, 109 .. 100] will give a list from 110 to 100.
- [0, -1 ..] – Negative integers.
- [-110..-100] – Syntax error; need [-110.. -100] for negatives.
- [1,3..99], [-1,3 .. 99] – List from 1 to 99 by 2, -1 to 99 by 4.

In fact, any value which is in the Enum class can be used:

- ['a' .. 'z'] – List of characters – a, b, ..., z.
- '\n, '\r, '\t, etc. – The standard codes for newline, carriage return, form feed, etc. are supported.
- ['z', 'y' .. 'a'] – z, y, x, ..., a.
- [1.0, 1.5 .. 2] – [1.0, 1.5, 2.0].
- [UppercaseLetter ..] – List of GeneralCategory values (from Data.Char).

**Escape Codes** The following escape codes can be used in characters or strings:

The following escape codes can be used in characters or strings:

## Lists & Tuples

- [] – Empty list.
- [1,2,3] – List of three numbers.
- 1 : 2 : 3 : [] – Alternate way to write lists using “cons” (:) and “nil” ([]).
- "abc" – List of three characters (strings are lists).
- 'a' : 'b' : 'c' : [] – List of characters (same as "abc").
- (1, "a") – 2-element tuple of a number and a string.
- (head, tail, 3, 'a') – 4-element tuple of two functions, a number and a character.

**Function Definition** Indent the body at least one space from the function name:

```
square x =  
  x * x
```

Unless a where clause is present. In that case, indent the where clause at least one space from the function name and any function bodies at least one space from the where keyword:

```
square x =  
  x2  
  where x2 =  
    x * x
```

## “Layout” rule, braces and semi-colons.

Haskell can be written using braces and semi-colons, just like C. However, no one does. Instead, the “layout” rule is used, where spaces represent scope. The general rule is: always indent. When the compiler complains, indent more.

**Braces and semi-colons** Semi-colons terminate an expression, and braces represent scope. They can be used after several keywords: where, let, do and of. They cannot be used when defining a function body. For example, the below will not compile.

```
square2 x = { x * x; }
```

However, this will work fine:

```
square2 x = result  
  where { result = x * x; }
```

## Function Definition

Functions are defined by declaring their name, any arguments, and an equals sign:

```
square x = x * x
```

All functions names must start with a lowercase letter or “\_”. It is a syntax error otherwise.

**Pattern Matching** Multiple “clauses” of a function can be defined by “pattern-matching” on the values of arguments. Here, the agree function has four separate cases:

```
-- Matches when the string "y" is given.  
agree1 "y" = "Great!"  
-- Matches when the string "n" is given.  
agree1 "n" = "Too bad."  
-- Matches when string beginning  
-- with 'y' given.  
agree1 ('y':_) = "YAHOO!"  
-- Matches for any other value given.  
agree1 _ = "SO SAD."
```

Note that the ‘\_’ character is a wildcard and matches any value.

Pattern matching can extend to nested values. Assuming this data declaration:

```
data Bar = Bil (Maybe Int) | Baz
```

and recalling the [definition of Maybe](#) from page 7 we can match on nested Maybe values when B11 is present:

```
f (Bil (Just _)) = ...  
f (Bil Nothing) = ...  
f Baz = ...
```

The following section details rules on function declarations, list comprehensions, and other areas of the language.

## Declarations, Etc.

Pattern-matching also allows values to be assigned to variables. For example, this function determines if the string given is empty or not. If not, the value bound to str is converted to lower case:

```
toLowerCaseStr [] = []
toLowerCaseStr str = map toLower str
```

Note that str above is similer to \_ in that it will match anything; the only difference is that the value matched is also given a name.

**n + k Patterns** This (sometimes controversial) pattern-matching facility makes it easy to match certain kinds of numeric expressions. The idea is to define a base case (the "n" portion) with a constant number for matching, and then to define other matches (the "k" portion) as additives to the base case. Here is a rather inefficient way of testing if a number is even or not:

```
isEven 0 = True
isEven 1 = False
isEven (n + 2) = isEven n
```

**Argument Capture** Argument capture is useful for pattern-matching a value *and* using it, without declaring an extra variable. Use an @ symbol in between the pattern to match and the variable to bind the value to. This facility is used below to bind the head of the list in l1 for display, while also binding the entire list to ls in order to compute its length:

```
len ls@(1:_ ) = "List starts with " ++
  show l ++ " and is " ++
  show (length ls) ++ " items long."
len [] = "List is empty!"
```

Pattern-matching also allows values to be assigned to variables. For example, this function determines if the string given is empty or not. If not, the value bound to str is converted to lower case:

```
toLowerCaseStr [] = []
toLowerCaseStr str = map toLower str
```

Note that str above is similer to \_ in that it will match anything; the only difference is that the value matched is also given a name.

This (sometimes controversial) pattern-matching facility makes it easy to match certain kinds of numeric expressions. The idea is to define a base case (the "n" portion) with a constant number for matching, and then to define other matches (the "k" portion) as additives to the base case. Here is a rather inefficient way of testing if a number is even or not:

```
isEven 0 = True
isEven 1 = False
isEven (n + 2) = isEven n
```

Argument capture is useful for pattern-matching a value *and* using it, without declaring an extra variable. Use an @ symbol in between the pattern to match and the variable to bind the value to. This facility is used below to bind the head of the list in l1 for display, while also binding the entire list to ls in order to compute its length:

```
len ls@(1:_ ) = "List starts with " ++
  show l ++ " and is " ++
  show (length ls) ++ " items long."
len [] = "List is empty!"
```

**Guards** Boolean functions can be used as "guards" in function definitions along with pattern matching. An example without pattern matching:

```
which n
| n == 0 = "zero!"
| even n = "even!"
| otherwise = "odd!"
```

Notice otherwise – it always evaluates to True and can be used to specify a "default" branch. Guards can be used with patterns. Here is a function that determines if the first character in a string is upper or lower case:

```
what [] = "empty string!"
what (c:_)
```

**Matching & Guard Order** Pattern-matching proceeds in top to bottom order. Similarly, guard expressions are tested from top to bottom. For example, neither of these functions would be very interesting:

```
allEmpty _ = False
allEmpty [] = True
```

```
alwaysEven n
| otherwise = False
| n `div` 2 == 0 = True
```

Boolean functions can be used as "guards" in function definitions along with pattern matching. An example without pattern matching:

```
which n
| n == 0 = "zero!"
| even n = "even!"
| otherwise = "odd!"
```

Notice otherwise – it always evaluates to True and can be used to specify a "default" branch. Guards can be used with patterns. Here is a function that determines if the first character in a string is upper or lower case:

```
what [] = "empty string!"
what (c:_)
```

**Matching & Guard Order** Pattern-matching proceeds in top to bottom order. Similarly, guard expressions are tested from top to bottom. For example, neither of these functions would be very interesting:

```
allEmpty _ = False
allEmpty [] = True
```

```
alwaysEven n
| otherwise = False
| n `div` 2 == 0 = True
```

syntax, however, can match based on those record names. Given this data type:

```
data Color = C { red
  , green
  , blue :: Int }
```

we can match on green only:

```
isGreenZero (C { green = 0 }) = True
isGreenZero _ = False
```

syntax, however, can match based on those record names. Given this data type:

```
data Color = C { red
  , green
  , blue :: Int }
```

we can match on green only:

```
isGreenZero (C { green = 0 }) = True
isGreenZero _ = False
```

Argument capture is possible with this syntax, although it gets chunky. Continuing the above, we now define a Pixel type and a function to replace values with non-zero green components with all black:

```
data Pixel = P Color
-- Color value untouched if green is 0
setGreen (P col@(C { green = 0 })) = P col
setGreen _ = P (C 0 0 0)
```

**Lazy Patterns**

This syntax, also known as *irrefutable* patterns, allows pattern matches which always succeed. That means any clause using the pattern will succeed, but if it tries to actually use the matched value an error may occur. This is generally useful when an action should be taken on the *type* of a particular value, even if the value isn't present.

For example, define a class for default values:

```
class Def a where
  defValue :: a -> a
```

The idea is you give defValue a value of the right type and it gives you back a default value for that type. Defining instances for basic types is easy:

```
instance Def Bool where  
  defValue _ = False
```

```
instance Def Char where  
  defValue _ = , ,
```

Maybe is a little trickier, because we want to get a default value for the type, but the constructor might be `Nothing`. The following definition would work, but it's not optimal since we get `Nothing` when `Nothing` is passed in.

```
instance Def a => Def (Maybe a) where  
  defValue (Just x) = Just (defValue x)  
  defValue Nothing = Nothing
```

We'd rather get a `Just` (*default value*) back instead. Here is where a lazy pattern saves us – we can pretend that we've matched `Just x` and use that to get a default value, even if `Nothing` is given:

```
instance Def a => Def (Maybe a) where  
  defValue ~(Just x) = Just (defValue x)
```

As long as the value `x` is not actually evaluated, we're safe. None of the base types need to look at `x` (see the `"_"` matches they use), so things will work just fine.

One wrinkle with the above is that we must provide type annotations in the interpreter or the code when using a `Nothing` constructor. Nothing has type `Maybe a`, but, if not enough other information is available, Haskell must be told what `a` is. Some example default values:

```
-- Return "Just False"  
defMB = defValue (Nothing :: Maybe Bool)  
-- Return "Just , "  
defMC = defValue (Nothing :: Maybe Char)
```

## List Comprehensions

A list comprehension consists of four types of elements: *generators*, *guards*, *local bindings*, and *targets*. A list comprehension creates a list of target values based on the generators and guards given. This comprehension generates all squares:

```
squares = [x * x | x <- [1..]]
```

`x <- [1..]` generates a list of all `Integer` values and puts them in `x`, one by one. `x * x` creates each element of the list by multiplying `x` by itself.

Guards allow certain elements to be excluded. The following shows how divisors for a given number (excluding itself) can be calculated. Notice how `d` is used in both the guard and target expression.

```
divisors n =  
  [d | d <- [1..(n `div` 2)]  
    , n `mod` d == 0]
```

Local bindings provide new definitions for use in the generated expression or subsequent generators and guards. Below, `z` is used to represent the minimum of `a` and `b`:

```
strange = [(a,z) | a <- [1..3]  
                  ; b <- [1..3]  
                  ; c <- [1..3]  
                  , let z = min a b  
                  , z < c ]
```

Comprehensions are not limited to numbers. Any list will do. All upper case letters can be generated:

```
ups =  
  [c | c <- [minBound .. maxBound]  
      , isUpper c]
```

Or, to find all occurrences of a particular break value `br` in a list word (indexing from 0):

```
idxs word br =  
  [i | (i, c) <- zip [0..] word  
      , c == br]
```

A unique feature of list comprehensions is that pattern matching failures do not cause an error; they are just excluded from the resulting list.

## Operators

There are very few predefined “operators” in Haskell—most that appear predefined are actually syntax (e.g., `=`). Instead, operators are simply functions that take two arguments and have special syntactic support. Any so-called operator can be applied as a prefix function using parentheses:

```
3 + 4 == (+) 3 4
```

To define a new operator, simply define it as a normal function, except the operator appears between the two arguments. Here's one which inserts a comma between two strings and ensures no extra spaces appear:

```
first ## last =  
  let trim s = dropWhile isSpace  
    (reverse (dropWhile isSpace  
              (reverse s)))  
    in trim last ++ ", " ++ trim first  
    > " Haskell " ## " Curry "  
Curry, Haskell
```

Of course, full pattern matching, guards, etc. are available in this form. Type signatures are a bit different, though. The operator “name” must appear in parentheses:

```
(##) :: String -> String -> String
Allowable symbols which can be used to define
operators are:
# $ % & * + . / < = > ? @ \ ~ | -
```

However, there are several “operators” which cannot be redefined. They are: <, -> and =. The last, =, cannot be redefined by itself, but can be used as part of multi-character operator. The “bind” function, >>=, is one example.

**Precedence & Associativity** The precedence and associativity, collectively called *fixity*, of any operator can be set through the infixr, infix, infixl and infixl keywords. These can be applied both to top-level functions and to local definitions. The syntax is:

```
{infix | infixr | infixl} precedence op
```

where *precedence* varies from 0 to 9. *Op* can actually be any function which takes two arguments (i.e., any binary operation). Whether the operator is left or right associative is specified by infixl or infixr, respectively. Such infix declarations have no associativity.

Precedence and associativity make many of the rules of arithmetic work “as expected.” For example, consider these minor updates to the precedence of addition and multiplication:

```
infixl 8 `plus1'
plus1 a b = a + b
infixl 7 `mult1'
mult1 a b = a * b
```

The results are surprising:

```
> 2 + 3 * 5
17
> 2 `plus1` 3 `mult1` 5
25
```

Reversing associativity also has interesting effects. Redefining division as right associative:

```
infixr 7 `div1'
div1 a b = a / b
```

We get interesting results:

```
> 20 / 2 / 2
5.0
> 20 `div1` 2 `div1` 2
20.0
```

## Currying

In Haskell, functions do not have to get all of their arguments at once. For example, consider the convert0Only function, which only converts certain elements of string depending on a test:

```
convert0Only test change str =
  map (\c -> if test c
        then change c
        else c) str
```

Using convert0Only, we can write the 133t function which converts certain letters to numbers:

```
133t = convert0Only isL33t tol33t
      where
            isL33t 'o' = True
            isL33t 'a' = True
            -- etc.
            isL33t _ = False
            tol33t 'o' = '0,
```

Notice that 133t has no arguments specified. Also, the final argument to convert0Only is not given. However, the type signature of 133t tells the whole story:

```
133t :: String -> String
```

That is, 133t takes a string and produces a string. It is a “constant”, in the sense that 133t always returns a value that is a function which takes a string and produces a string. 133t returns a “curried” form of convert0Only, where only two of its three arguments have been supplied.

This can be taken further. Say we want to write a function which only changes upper case letters.

We know the test to apply, isUpper, but we don’t want to specify the conversion. That function can be written as:

```
convertUpper = convert0Only isUpper
```

```
convertUpper :: (Char -> Char)
              -> String -> String
```

That is, convertUpper can take two arguments. The first is the conversion function which converts individual characters and the second is the string to be converted.

A curried form of any function which takes multiple arguments can be created. One way to think of this is that each “arrow” in the function’s signature represents a new function which can be created by supplying one more argument.

**Sections** Operators are functions, and they can be curried like any other. For example, a curried version of “+” can be written as:

```
add10 = (+) 10
```

However, this can be unwieldy and hard to read. “Sections” are curried operators, using parentheses. Here is add10 using sections:

```
add10 = (10 +)
```

The supplied argument can be on the right or left, which indicates what position it should take. This is important for operations such as concatenation:

```
onLeft str = (++ str)
onRight str = (str ++)
```

Which produces quite different results:

```
> onLeft "foo" "bar"
"barfoo"
> onRight "foo" "bar"
"foobar"
```

```
noGreen2 c = c { green = 0 }
```

Here we capture the Color value in *c* and return a new Color value. That value happens to have the same value for red and blue as *c* and its green component is 0. We can combine this with pattern matching to set the green and blue fields to equal the red field:

```
makeGrey c@(C { red = r }) =
  c { green = r, blue = r }
```

The supplied argument can be on the right or left, which indicates what position it should take. This is important for operations such as concatenation:

```
onLeft str = (++ str)
onRight str = (str ++)
```

Which produces quite different results:

```
> onLeft "foo" "bar"
"barfoo"
> onRight "foo" "bar"
"foobar"
```

```
multBy n = \m -> n * m
```

For example:

```
> let mult10 = multBy 10
   > mult10 10
100
```

## Type Signatures

Haskell supports full type inference, meaning in most cases no types have to be written down. Type signatures are still useful for at least two reasons.

## Anonymous Functions

An anonymous function (i.e., a *lambda expression* or *lambda* for short), is a function without a name. They can be defined at any time like so:

```
\c -> (c, c)
```

which defines a function that takes an argument and returns a tuple containing that argument in both positions. They are useful for simple functions which don't need a name. The following determines if a string consists only of mixed case letters and whitespace.

```
mixedCase str =
  all (\c -> isSpace c ||
       isLower c ||
       isUpper c) str
```

Haskell is a pure language and, as such, has no mutable state. That is, once a value is set it never changes. “Updating” is really a copy operation, with new values in the fields that “changed.” For example, using the Color type defined earlier, we can write a function that sets the green field to zero easily:

```
noGreen1 (C r - b) = C r 0 b
```

The above is a bit verbose and can be rewritten using record syntax. This kind of “update” only sets values for the field(s) specified and copies the rest:

**Documentation**—Even if the compiler can figure out the types of your functions, other programmers or even yourself might not be able to later. Writing the type signatures on all top-level functions is considered very good form.

**Specialization**—Typeclasses allow functions with overloading. For example, a function to negate any list of numbers has the signature:

```
negateAll :: Num a => [a] -> [a]
```

However, for efficiency or other reasons you may only want to allow Int types. You would accomplish that with a type signature:

```
negateAll :: [Int] -> [Int]
```

Of course, lambdas can be the returned from functions too. This classic returns a function which will then multiply its argument by the one originally given:

Type signatures can appear on top-level functions and nested let or where definitions. Generally this is useful for documentation, although in some cases they are needed to prevent polymorphism. A type signature is first the name of the item which will be typed, followed by a `:`, followed by the types. An example of this has already been seen above.

Type signatures do not need to appear directly above their implementation. They can be specified anywhere in the containing module (yes, even below!). Multiple items with the same signature can also be defined together:

```
pos, neg :: Int -> Int
...

```

```
pos x | x < 0 = negate x
      | otherwise = x
neg y | y > 0 = negate y
      | otherwise = y
```

**Type Annotations** Sometimes Haskell cannot determine what type is meant. The classic demonstration of this is the so-called “show . read” problem:

```
canParseInt x = show (read x)
```

Haskell cannot compile that function because it does not know the type of `read x`. We must limit the type through an annotation:

```
canParseInt x = show (read x :: Int)
```

Annotations have the same syntax as type signatures, but may adorn any expression. Note that the `_` token is a “wildcard” matching any value.

the annotation above is on the expression `read x`, not on the variable `x`. Only function application (e.g., `read x`) binds tighter than annotations. If that was not the case, the above would need to be written `(read x) :: Int`.

## Unit

`()` – “unit” type and “unit” value. The value and type that represents no useful information.

## Keywords

Haskell keywords are listed below, in alphabetical order.

## Case

case is similar to a switch statement in C# or Java, but can match a pattern: the shape of the value being inspected. Consider a simple data type:

```
data Choices = First String | Second |
              Third | Fourth
```

case can be used to determine which choice was given:

```
whichChoice ch =
  case ch of
    First -> "1st!"
    Second -> "2nd!"
    -> "Something else."
```

As with pattern-matching in function definitions, the `_` token is a “wildcard” matching any value.

**Nesting & Capture** Nested matching and binding are also allowed. For example, here is the definition of the `Maybe` type:

```
data Maybe a = Just a | Nothing
```

Using `Maybe` we can determine if any choice was given using a nested match:

```
anyChoice1 ch =
  case ch of
    Nothing -> "No choice!"
    Just (First _) -> "First!"
    Just Second -> "Second!"
    -> "Something else."
```

Binding can be used to manipulate the value matched:

```
anyChoice2 ch =
  case ch of
    Nothing -> "No choice!"
    Just score@(First "gold") ->
      "First with gold!"
    Just score@(First _) ->
      "First with something else:"
    -> "Not first."
```

**Matching Order** Matching proceeds from top to bottom. If `anyChoice1` is reordered as follows, the first pattern will always succeed:

```
anyChoice3 ch =
  case ch of
    -> "Something else."
    Nothing -> "No choice!"
    Just (First _) -> "First!"
    Just Second -> "Second!"
```

**Guards** Guards, or conditional matches, can be used in cases just like function definitions. The only difference is the use of the `->` instead of `=`. Here is a simple function which does a case-insensitive string match:

```
strcmp s1 s2 = case (s1, s2) of
  ([] , []) -> True
  (s1:ss1, s2:ss2)
    | toUpper s1 == toUpper s2 ->
      strcmp ss1 ss2
    | otherwise -> False
    -> False
```

## Class

A Haskell function is defined to work on a certain type or set of types and cannot be defined more than once. Most languages support the idea of “overloading”, where a function can have different behavior depending on the type of its arguments. Haskell accomplishes overloading through class and instance declarations. A class defines one or more functions that can be applied to any types which are members (i.e., instances) of that class. A class is analogous to an interface in Java or C#, and instances to a concrete implementation of the interface.

A class must be declared with one or more type variables. Technically, Haskell 98 only allows one type variable, but most implementations of Haskell support so-called *multi-parameter type classes*, which allow more than one type variable. We can define a class which supplies a flavor for a given type:

```
class Flavor a where
  flavor :: a -> String
```

Notice that the declaration only gives the type signature of the function—no implementation is given here (with some exceptions, see “[Defaults](#)” on page 8). Continuing, we can define several instances:

```
instance Flavor Bool where
  flavor _ = "sweet"

instance Flavor Char where
  flavor _ = "sour"

Evaluating flavor True gives:
> flavor True
"sweet"
```

While flavor 'x' gives:

```
> flavor 'x'
"sour"
```

**Default Implementations** Default implementations can be given for functions in a class. These are useful when certain functions can be defined in terms of others in the class. A default is defined by giving a body to one of the member functions. The canonical example is `Eq`, which defines `/=` (not equal) in terms of `==`:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  (/=) a b = not (a == b)
```

Recursive definitions can be created. Continuing the `Eq` example, `==` can be defined in terms of `/=`:

```
(==) a b = not (a /= b)
```

However, if instances do not provide enough concrete implementations of member functions then any program using those instances will loop.

## Data

So-called *algebraic data types* can be declared as follows:

```
data MyType = MyValue1 | MyValue2

MyType is the type's name. MyValue1 and MyValue are values of the type and are called constructors. Multiple constructors are separated with the ‘|’ character. Note that type and constructor names must start with a capital letter. It is a syntax error otherwise.
```

## Constructors with Arguments

The type above is not very interesting except as an enumeration. Constructors that take arguments can be declared, allowing more information to be stored:

```
data Point = TwoD Int Int
           | ThreeD Int Int Int
```

Notice that the arguments for each constructor are *type names*, not constructors. That means this kind of declaration is illegal:

```
data Poly = Triangle TwoD TwoD TwoD
```

instead, the `Point` type must be used:

```
data Poly = Triangle Point Point Point
```

**Type and Constructor Names** Type and constructor names can be the same, because they will never be used in a place that would cause confusion. For example:

```
data User String | Admin String  
which declares a type named User with two constructors, User and Admin. Using this type in a function makes the difference clear:
```

```
whatUser (User _) = "normal user."  
whatUser (Admin _) = "admin user."
```

Some literature refers to this practice as *type punning*.

**Type Variables** Declaring so-called *polymorphic* data types is as easy as adding type variables in the declaration:

```
data Slot1 a = Slot1 a | Empty1
```

This declares a type Slot1 with two constructors, Slot1 and Empty1. The Slot1 constructor can take an argument of *any* type, which is represented by the type variable a above.

We can also mix type variables and specific types in constructors:

```
data Slot2 a = Slot2 a Int | Empty2
```

Above, the Slot2 constructor can take a value of any type and an Int value.

**Record Syntax** Constructor arguments can be declared either positionally, as above, or using record syntax, which gives a name to each argument. For example, here we declare a Contact type with names for appropriate arguments:

```
data Contact = Contact { ctName :: String  
    , ctEmail :: String  
    , ctPhone :: String }  
data Priority = Low | Medium | High  
deriving Show
```

These names are referred to as *selector* or *accessor* functions and are just that, functions. They must start with a lowercase letter or underscore and cannot have the same name as another function in scope. Thus the “ct” prefix on each above. Multiple constructors (of the same type) can use the same accessor function for values of the same type, but that can be dangerous if the accessor is not used by all constructors. Consider this rather contrived example:

```
data Con = Con { conValue :: String }  
| Uncon { conValue :: String }  
| Noncon  
whichCon con = "convalue is " ++  
    conValue con
```

If whichCon is called with a Noncon value, a runtime error will occur. Finally, as explained elsewhere, these names can be used for pattern matching, argument capture and “updating.”

Two forms of deriving are possible. The first is used when a type only derives one class:

```
data Alarm = Soft | Loud | Deafening  
deriving (Read, Show)
```

The second is used when multiple classes are derived:

```
data Alarm = Soft | Loud | Deafening  
deriving (Read, Show)
```

It is a syntax error to specify deriving for any other classes besides the seven given above.

**Class Constraints** Data types can be declared with class constraints on the type variables, but this practice is discouraged. It is better to hide the “raw” data constructors using the module system and instead export “smart” constructors which apply appropriate constraints. In any case, the syntax used is:

```
data (Num a) => SomeNumber a = Two a a  
| Three a a a
```

This declares a type SomeNumber which has one type variable argument. Valid types are those in the Num class.

**Deriving**

See the section on [deriving](#) under the data keyword on page 9.

**Do**

The do keyword indicates that the code to follow will be in a *monadic* context. Statements are separated by newlines, assignment is indicated by <-, and a let form is introduced which does not require the in keyword.

**If and IO** if can be tricky when used with IO. Conceptually it is no different from an if in any other context, but intuitively it is hard to develop. Consider the function `doesFileExists` from `System.Directory`:

```
doesFileExist :: FilePath -> IO Bool
The if statement has this "signature":
if-then-else :: Bool -> a -> a -> a
```

That is, it takes a `Bool` value and evaluates to some other value based on the condition. From the type signatures it is clear that `doesFileExist` cannot be used directly by if:

```
wrong fileName =
  if doesFileExist fileName
    then ...
  else ...
```

That is, `doesFileExist` results in an `IO Bool` value, while if wants a `Bool` value. Instead, the correct value must be "extracted" by running the IO action:

```
right1 fileName = do
  exists <- doesFileExist fileName
  if exists
    then return 1
    else return 0
```

Notice the use of `return`. Because `do` puts us "inside" the `IO` monad, we can't "get out" except through `return`. Note that we don't have to use `if inline here`—we can also use `let` to evaluate the condition and get a value first:

```
right2 fileName = do
  exists <- doesFileExist fileName
```

```
let result =
  if exists
    then 1
    else 0
  return result
```

Again, notice where `return` is. We don't put it in the `let` statement. Instead we use it once at the end of the function.

**Multiple do's** When using `do` with `if` or `case`, another `do` is required if either branch has multiple statements. An example with `if`:

```
countBytes1 f =
  do
    putStrLn "Enter a filename."
    args <- getLine
    case args of
      [] -> putStrLn "No args given."
      file -> do f <- readFile file;
                  putStrLn ("The file is " ++
                                show (length f)
                                ++ " bytes long."); }
```

## Export

See the section on `module` on page 11.

## If, Then, Else

An alternative syntax uses semi-colons and braces. A `do` is still required, but indentation is unnecessary. This code shows a case example, but the principle applies to `if` as well:

```
countBytes3 =
  do
    putStrLn "Enter a filename."
    args <- getLine
    case args of
      [] -> putStrLn "No args given."
      file -> do f <- readFile file;
                  putStrLn ("The file is " ++
                                show (length f)
                                ++ " bytes long."); }
```

## Export

See the section on `module` on page 11.

Remember, if always "returns" a value. It is an expression, not just a control flow statement. This function tests if the string given starts with a lower case letter and, if so, converts it to upper case:

```
f <- readFile args
putStrLn ("The file is " ++
          show (length f)
          ++ " bytes long.")
```

And one with `case`:

```
countBytes2 =
  do
    putStrLn "Enter a filename."
    args <- getLine
    case args of
      [] -> putStrLn "No args given."
```

```
-- Anything else is empty string
sentenceCase _ = []
```

## Import

See the section on [module](#) on page 11.

## In

See [let](#) on page 11.

## Infix, infixl and infixr

See the section on [operators](#) on page 4.

## Instance

See the section on [class](#) on page 8.

## Let

Local functions can be defined within a function using `let`. The `let` keyword must always be followed by `in`. The `in` must appear in the same column as the `let` keyword. Functions defined have access to all other functions and variables within the same scope (including those defined by `let`). In this example, `mult` multiplies its argument `n` by `x`, which was passed to the original multiples. `mult` is used by `map` to give the multiples of `x` up to 10:

```
multiples x =
  let mult n = n * x
  in map mult [1..10]
```

Let "functions" with no arguments are actually constants and, once evaluated, will not evaluate again for that invocation of the outer function.

This is useful for capturing common portions of your function and re-using them. Here is a silly example which gives the sum of a list of numbers and their average. The numbers definition captures the list of numbers from 1 to `m`, and will only be evaluated once per invocation of `ListStats`; similarly, `total` and `avg` are only evaluated once per invocation:

```
listStats m =
  let numbers = [1 .. m]
  total = sum numbers
  avg = total / m
  in "total: " ++ show total ++
      ", avg: " ++ show avg
```

**Deconstruction** The left-hand side of a `let` definition can also destructure its argument, in case sub-components are to be accessed. This definition would extract the first three characters from a string

```
firstThree str =
  let (a:b:c:_)=str
  in "Initial three characters are: " ++
    show a ++ " , " ++
    show b ++ " , and " ++
    show c
```

Note that this is different than the following, which only works if the string has exactly three characters:

```
onlyThree str =
  let (a:b:c:[_])=str
  in "The characters given are: " ++
    show a ++ " , " ++
    show b ++ " , and " ++
    show c
```

**Of**  
See the section on [case](#) on page 7.

## Module

A module is a compilation unit which exports functions, types, classes, instances, and other modules. A module can only be defined in one file, though its exports may come from multiple sources. To make a Haskell file a module, just add a module declaration at the top:

```
module MyModule where
```

Module names must start with a capital letter but otherwise can include periods, numbers and underscores. Periods are used to give sense of structure, and Haskell compilers will use them as indications of the directory a particular source file is, but otherwise they have no meaning.

The Haskell community has standardized a set of top-level module names such as `Data`, `System`, `Network`, etc. Be sure to consult them for an appropriate place for your own module if you plan on releasing it to the public.

**Imports** The Haskell standard libraries are divided into a number of modules. The functionality provided by those libraries is accessed by importing into your source file. To import everything exported by a library, just use the module name:

```
import Text.Read
```

Everything means *everything*: functions, data types and constructors, class declarations, and even other modules imported and then exported

by the that module. Importing selectively is accomplished by giving a list of names to import. For example, here we import some functions from `Text.Read`:

```
import Text.Read (readParen, lex)
```

Data types can be imported in a number of ways. We can just import the type and no constructors:

```
import Text.Read (Lexeme)
```

Of course, this prevents our module from pattern-matching on the values of type `Lexeme`. We can import one or more constructors explicitly:

```
import Text.Read (Lexeme(Ident, Symbol))
```

All constructors for a given type can also be imported:

```
import Text.Read (Lexeme(..))
```

We can also import types and classes defined in the module:

```
import Text.Read (Read, ReadS)
```

In the case of classes, we can import the functions defined for a class using syntax similar to importing constructors for data types:

```
import Text.Read (Read(readsPrec  
, readList))
```

Note that, unlike data types, all class functions are imported unless explicitly excluded. To *only* import the class, we use this syntax:

```
import Text.Read (Read())
```

**Exclusions** If most, but not all, names are to be imported from a module, it would be tedious to list them all. For that reason, imports can also be specified via the `hiding` keyword:

```
import Data.Char hiding (isControl  
, isMark)
```

Except for instance declarations, any type, function, constructor or class can be hidden.

**Instance Declarations** It must be noted that instance declarations *cannot* be excluded from import: all instance declarations in a module will be imported when the module is imported.

**Qualified Imports** The names exported by a module (i.e., functions, types, operators, etc.) can have a prefix attached through qualified imports. This is particularly useful for modules which have a large number of functions having the same name as Prelude functions. `Data.Set` is a good example:

```
import qualified Data.Set as Set
```

This form requires any function, type, constructor or other name exported by `Data.Set` to now be prefixed with the *alias* (i.e., `Set`) given. Here is one way to remove all duplicates from a list:

```
removeDups a =  
  Set.toList (Set.fromList a)
```

```
import Text.Read (Read(readsPrec  
, readList))
```

A second form does not create an alias. Instead, the prefix becomes the module name. We can write a simple function to check if a string is all upper case:

```
import qualified Char
```

```
allUpper str =  
  all Char.isUpper str
```

Except for the prefix specified, qualified imports support the same syntax as normal imports. The name imported can be limited in the same ways as described above.

**Exports** If an export list is not provided, then all functions, types, constructors, etc. will be available to anyone importing the module. Note that any imported modules are *not* exported in this case. Limiting the names exported is accomplished by adding a parenthesized list of names before the `where` keyword:

```
module MyModule (MyType  
, MyClass  
, myFunc1  
...)
```

The same syntax as used for importing can be used here to specify which functions, types, constructors, and classes are exported, with a few differences. If a module imports another module, it can also export that module:

```
module MyBigModule (module Data.Set  
, module Data.Char)
```

```
where
```

```
import Data.Set  
import Data.Char
```

A module can even re-export itself, which can be useful when all local definitions and a given imported module are to be exported. Below we export ourselves and `Data.Set`, but not `Data.Char`:

```
module AnotherBigModule (module Data.Set
  , module AnotherBigModule)
where
```

```
import Data.Set
import Data.Char
```

## Newtype

While `data` introduces new values and type just creates synonyms, `newtype` falls somewhere between. The syntax for `newtype` is quite restricted—only one constructor can be defined, and that constructor can only take one argument. Continuing the above example, we can define a `Phone` type as follows:

```
newtype Home = H String
newtype Work = W String
data Phone = Phone Home Work
```

As opposed to `type`, the `H` and `W` “values” on `Phone` are *not* just `String` values. The typechecker treats them as entirely new types. That means our `lowerName` function from above would not compile. The following produces a type error:

```
1Phone (Phone hm wk) =
  Phone (lower hm) (lower wk)
```

Instead, we must use pattern-matching to get to the “values” to which we apply `lower`:

```
1Phone (Phone (H hm) (W wk)) =
  Phone (H (lower hm)) (W (lower wk))
```

The key observation is that this keyword does not introduce a new value; instead it introduces a new type. This gives us two very useful properties:

- No runtime cost is associated with the new type, since it does not actually produce new values. In other words, newtypes are absolutely free!

- The type-checker is able to enforce that common types such as `Int` or `String` are used in restricted ways, specified by the programmer.

Finally, it should be noted that any deriving clause which can be attached to a `data` declaration can also be used when declaring a newtype.

## Return

See [do](#) on page [9](#).

## Type

This keyword defines a *type synonym* (i.e., alias). This keyword does not define a new type, like `data` or `newtype`. It is useful for documenting code but otherwise has no effect on the actual type of a given function or value. For example, a `Person` data type could be defined as:

```
data Person = Person String String
```

where the first constructor argument represents their first name and the second their last. However, the order and meaning of the two arguments is not very clear. A type declaration can help:

```
type FirstName = String
type LastName = String
data Person = Person FirstName LastName
```

Because `type` introduces a synonym, type checking is not affected in any way. The function `lower`, defined as:

```
lower s = map toLower s
```

- which has the type

- `lower :: String -> String`

can be used on values with the type `FirstName` or `LastName` just as easily:

```
1Name (Person f 1) =
  Person (lower f) (lower 1)
```

## Where

Similar to `let`, `where` defines local functions and constants. The scope of a `where` definition is the current function. If a function is broken into multiple definitions through pattern-matching, then the scope of a particular `where` clause only applies to that definition. For example, the function `strlen` below has a different meaning depending on the arguments given to the function `strlen`:

```
strlen [] = result
where result = "No string given!"
strlen f = result ++ " characters long!"
where result = show (length f)
```

**Where vs. Let** A `where` clause can only be defined at the level of a function definition. Usually, that is identical to the scope of `let` definition. The only difference is when guards are being used. The scope of the `where` clause extends over all guards. In contrast, the scope of a `let` expression is only the current function clause *and* guard, if any.

## Contributors

My thanks to those who contributed patches and useful suggestions: Dave Bayer, Evgenij Belikov, Paul Butler, Elisa Firth, Marc Fontaine, Brian Gi-anforcaro, Cale Gibbard, Andrew Harris, Stephen Hicks, Kurt Hutchinson, Johan Kiviniemi, Patrik Jansson, Adrian Neumann, Barak Pearlmut-

ter, Lanny Ripple, Markus Roberts, Holger Siegel, Falko Spiller, Adam Vogt, Leif Warner, and Jeff Zaroyko.

## Version

This is version 2.9. The source can be found at GitHub (<https://github.com/m4dc4p/>

cheatsheet). The latest released version of the PDF can be downloaded from <http://cheatsheet.codeslower.com>. Visit CodeSlower.com (<http://blog.codeslower.com/>) for other projects and writings.