

What are Haskell's case rules?

Variables begin with a lowercase letter.

Type names begin with an uppercase letter.

Give the comment syntax.

-- comment out the rest of the line.

```
{- multiline comment -}
```

(nesting is allowed)

Describe Haskell's Boolean types?

True and False are of type Bool.

The type is strict, only admitting those two literals.

What are the Boolean operators?

Equality/Inequality?

&&, ||, not

'==' and '/='

Haskell's math operators have types that are
...

... strict, since there's no coercion. They have integer and floating point versions.

Convert `Float` to `Int`.

truncate

Convert a `Char` to an `Int` and back.

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

What is a string, really?

A list of `Chars`, that is, `[Char]`.

Get string representations of objects.

show whatever

Make a prefix operator infix.

Make an infix operator prefix.

Surrounding an identifier with back ticks makes it infix.

Surrounding an operator with parens makes it prefix.

Index a list.

Test for membership.

```
ghci> ['a', 'b', 'c'] !! 0  
'a'
```

elem, notElem

Remove duplicates in a list.

Concatenate two lists.

nub

concat

Access the elements of a 2-tuple.

fst, snd

What is the range syntax?

$[a..b]$ is a list of all the values from a to b , inclusive.

$[a..]$ is an infinite list from a up.

What is the syntax of list comprehensions?

[expr | generatorOrGuard1, ... ,generatorOrGuardN]

Guards are just expressions that result in `Bool`. No `if` is used.

How are lambdas written?

\params -> result

How are new variables introduced?

expression where declarations

let declarations in expressions

The expression can have multiple new variables in it.

How are `case` expressions introduced?

```
case expr1 of  
  expr2 -> ...  
  expr3 -> ...
```

How are function guards introduced?

```
func params =  
  | boolean1 -> ...  
  | boolean2 -> ...
```

Access the elements of a tuple.

fst **and** snd

What determines indentation in common expressions?

What can be used instead.

The first nonblank character following `where`, `let`, or `of` determines the starting column.

Curlyes can be used instead, but indentation alone is generally preferred.

Each case of a function must have ...

... exactly the same signature. There is no overloading.

What does `+` do in a pattern?

Allows you to create variables with an offset from what is matched:

`subtractFive (n + 5) = n`

Define the *bind* operator.

`(>>=) :: (Monad m) => m a -> (a -> m b) -> m b`

It passes the "state of the world" resulting from one function to the next function.

Define the *then* operator.

What does it provide over *bind*?

`(>>) :: (Monad m) => m a -> m b -> m b`

It is convenient to have another function that doesn't demand a function as its second argument.

Define `return`.

```
return :: (Monad m) => a -> m a
```

It creates a monad container for arbitrary values.

Do notation is really syntactic sugar for what?

One `>>=/>>` after another. It also allows the `let` form.

Give an infinite loop that allows I/O.

```
import Control.Monad
forever :: (Monad m) => m a -> m b
forever a = a >> forever a
```


A monad consists of what?

- A type constructor M .

- A bind operation

$(\gg=) :: (Monad\ m) \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

- A return operation

$return :: (Monad\ m) \Rightarrow a \rightarrow m\ a$

What rules do monads obey?

`return x >>= f` `=` `f x`

`m >>= return` `=` `m`

`>>=` is associative

What problem does `when` solve?

Define it.

It's often encessary to check a condition in an I/O function and in one case `return IO`.

`when :: (Monad m) => Bool -> m () -> m ()`

It does the `return ()` for you in the event the condition fails.

In some cases `do` can be replaced with ...

... sequence.

```
sequence :: (Monad m) => [m a] -> m [a]
```

This will work only if all statements in the `do` return the same type.

Read the contents of a file.

```
import System.IO  
handle <- openFile "myFile.txt" ReadMode  
contents <- hGetContents handle  
hClose handle
```

Give the signature of `openFile`.

How can the file automatically be closed for
you?

```
ghci> :t openFile
openFile :: FilePath -> IOMode -> IO Handle
ghci> :t withFile
withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r
```

Give shortcuts for quickly reading/writing files.

```
readFile :: FilePath -> IO String  
writeFile :: FilePath -> String -> IO ()  
appendFile :: FilePath -> String -> IO ()
```