What are Haskell's case rules?

Variables begin with a lowercase letter.

Type names begin with an uppercase letter.

CIS 554 Haskell

Give the comment syntax.

-- comment out the rest of the line.

```
{- multiline comment -}
(nesting is allowed)
```

CIS 554

Haskell

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 `/=`

CIS 554

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

Haskell

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

CIS 554

Haskell

Convert Float to Int.



CIS 554

Haskell

Convert a Char to an Int and back.

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

CIS 554

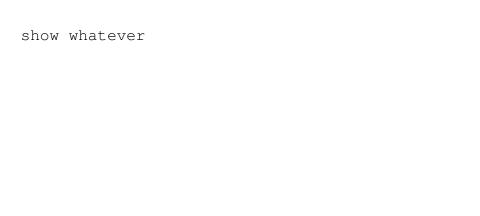
Haskell

What is a string, really?

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

CIS 554 Haskell

Get string representations of objects.



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.

Haskell

Index a list.

Test for membership.

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

1

Remove duplicates in a list.

Haskell

Concatenate two lists.

nub

concat

Access the elements of a 2-tuple.

fst, snd

CIS 554

Haskell

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.

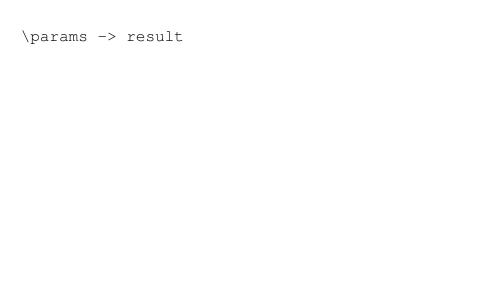
CIS 554 Haskell

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?



CIS 554 Haskell

How are new variables introduced?

expression where declarations

let declarations in expressions

The expression can have multiple new variables in it.

CIS 554 Haskell

How are case expressions introduced?

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

CIS 554

How are function guards introduced?

Haskell

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

What is a lazy data structure?

What's an easy way to make them in Clojure?

A data structure whose parts don't exist until they are accessed.

iterate takes a function  ${\tt f}$  and a starting value  ${\tt n}$  and produces a lazy infinite series:

```
(n, f(n), f(f(n)), f(f(f(n))), ...)
```

CIS 554

Haskell

# What does the & mean in a function signature?

It comes before a vararg, which is available as a list in the body.

#### Describe do.

(do exprs\*)

Evaluates the expressions in order and returns the value of the last.

Describe list comprehension syntax.

(for seq-exprs body-expr)

Takes a vector of one or more binding-form/collection-expr pairs, each followed by zero or more modifiers, and yields a lazy sequence of evaluations of expr.

Supported modifiers are: :let [binding-form expr ...], :while test, :when test.

How do you partially apply functions?

## Use partial, followed by a function and fewer than the normal number of arguments.

```
user=> (def equals5 (partial = 5))
#'user/equals5
user=> (equals5 5)
true
```

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.

Curlies 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.

Haskell

What does + do in a pattern?

28

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

```
subtractFive (n + 5) = n
```

CIS 554

Haskell

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.

Haskell

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

Haskell

A monad consists of what?

- A type constructor M.
- A bind operation
- (>>=) :: (Monad m) => m a -> (a -> m b) -> m b
- A return operation
- return :: (Monad m) => a -> m a

What rules do monads obey?

```
return x \gg 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 replaces with ...

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

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