## Examples of Expressions And Types

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True &amp;&amp; not False</td>
<td>Bool</td>
</tr>
<tr>
<td>$1 - 2 \times 3 + 4$</td>
<td>Int, Integer, Rational, Double, ...</td>
</tr>
<tr>
<td>'C'</td>
<td>Char</td>
</tr>
<tr>
<td>(True, 'C')</td>
<td>(Bool, Char)</td>
</tr>
<tr>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>3 : 1 : 4 : []</td>
<td>[Int], [Integer], [Rational], [Double], ...</td>
</tr>
<tr>
<td>a.k.a. [3, 1, 4]</td>
<td>String a.k.a. [Char]</td>
</tr>
<tr>
<td>&quot;hello&quot;</td>
<td>Maybe Char</td>
</tr>
<tr>
<td>Just 'C'</td>
<td>Maybe Char, Maybe Int, ...</td>
</tr>
<tr>
<td>Nothing</td>
<td>Either Char Bool, Either Char Int, ...</td>
</tr>
<tr>
<td>Left 'C'</td>
<td>Either Char Bool, Either Int Bool, ...</td>
</tr>
<tr>
<td>Right False</td>
<td>Char -&gt; Bool</td>
</tr>
<tr>
<td>$\lambda x \rightarrow x \geq 'C'$</td>
<td>Char -&gt; Bool</td>
</tr>
</tbody>
</table>

1/34
Definitions (Bindings); Type Signatures

Variable (constant?) definition/binding:
ten = 1 + 2 + 3 + 4
“Define ten to be 1 + 2 + 3 + 4.”
“Bind 1 + 2 + 3 + 4 to ten.”
Special case of “pattern binding” (covered later). Teaser preview:
(four, c) = (2+2, ’C’)

Function definitions/bindings:
square x = x * x
nand a b = not (a && b)

Their names must start with lowercase letter.

Type Signatures:
ten, four :: Integer
square :: Integer -> Integer (simplified story)
nand :: Bool -> Bool -> Bool
Function Application; Function Types

Recall \( \text{nand} \ a \ b = \text{not} \ (a \land b) \)

How to use (function application):

\( \text{nand} \ True \ False \) means \( (\text{nand} \ True) \ False \).

\( \text{nand} \ True \) alone is a function from \( \text{Bool} \) to \( \text{Bool} \).

It is right to think of nested lambdas:
\( \text{nand} = \lambda a \rightarrow \lambda b \rightarrow \text{not} \ (a \land b) \)

and it has a shorthand:
\( \text{nand} = \lambda a \ b \rightarrow \text{not} \ (a \land b) \)

(Therefore \( \text{nand} \) is considered a variable too.)

Recall \( \text{nand} :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \)

It means \( \text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool}) \).
Local Definitions For Expressions

\[
\begin{align*}
\text{let } & \quad x = 4 + 5 \\
\text{let } & \quad y = 4 - 5 \\
\text{in } & \quad x + y + 2*x*y
\end{align*}
\]

Layout version above. Braced version below:

\[
\begin{align*}
\text{let } \{ & \quad x = \text{expr1} \ ; \ y = \text{expr2} \} \text{ in } x + y + 2*x*y
\end{align*}
\]

Style guide: Prefer layout.
Local Definitions For Definitions

\[ \text{foo } u \ v = x + y + 2*x*y \]

\[ \text{where} \]
\[ \quad \text{-- this "where" belongs to "foo u v ="}, \]
\[ \quad \text{-- not to "x + y + 2*x*y"} \]
\[ \quad x = u + v \]
\[ \quad y = u - v \]

Layout version above. Braced version below:

\[ \text{foo } u \ v = x + y + 2*x*y \text{ where} \{ x = u + v ; y = u - v \} \]

Style guide: Prefer layout.

You may like this style guide.
Pattern Matching

Pattern matching in expressions:

```haskell
case expr of
  [] -> 0
  42 : xs -> foo xs
  x : xs -> x + foo xs
```

Pattern matching in function definitions:

```haskell
mySum [] = 0
mySum (x : xs) = x + mySum xs
```

```haskell
nand False _ = True
nand True False = True
nand True True = False
```

Pattern binding:

```haskell
[a, b, c] = take 3 someList
```
Guards

“Guards” are extra conditions imposed on patterns. (Boolean conditions shown. But there are others.)

For case:

```haskell
case expr of
  []  ->  0
  x : xs | x < 0 -> x + foo xs
          | x > 2 -> x - foo xs
          | True  -> x * foo xs
```

For definitions:

```haskell
foo []  =  0
foo (x : xs) | x < 0 = x + foo xs
              | x > 2 = x - foo xs
              | True  = x * foo xs
```

Can use otherwise for True. (Constant defined in library.)
Local Definitions under Patterns+Guards

```haskell
foo :: Either String Integer -> Integer
foo (Left str) | suffix > "albert" = 42
   | otherwise = 24
where
  suffix = drop 10 str
foo (Right x) | x > 0 = 2 * y
   | x < 0 = y
   | otherwise = 0
where
  y = div 10000 x
```

First `where` belongs to all of `foo (Left str)`.

Second `where` belongs to all of `foo (Right x).`
And if \(x = 0\), the division by zero is not performed.

Similar story for `case`. 
List Comprehension

\[ \{ x + y \mid x \leftarrow [10, 20, 30], x > 10, y \leftarrow [4, 5] \} \]
Answer: \[20 + 4, 20 + 5, 30 + 4, 30 + 5\]

Can use pattern matching too:

\[ \{ x + 3 \mid \text{Just } x \leftarrow [\text{Just 10, Nothing, Just 30}] \} \]
Answer: \[10 + 3, 30 + 3\]

(Aside: There is also range notation \([1..5]\) for \([1, 2, 3, 4, 5]\), but it is not part of list comprehension. Although, the two are often used together.)
Algebraic Data Types

data MyType = Nada | Duplet Double String | Uno Integer

Math view: Tagged disjoint union of cartesian products.
Programmer view: Enumerated type with data fields.

Nada, Duplet, Uno are “data constructors”. They must start with uppercase letters. They form expressions and patterns. (Unrelated to Java or C++ constructors.)

Below, patterns on LHS, expressions on RHS:
plus1 :: MyType -> MyType
plus1 Nada = Nada
plus1 (Duplet r s) = Duplet (r+1) s
plus1 (Uno i) = Uno (i+1)

List, unit, tuple, Maybe, and Either are algebraic data types from the standard library.

Recursion is OK: data Stack = Bottom | Push Int Stack
Parametric Polymorphism

\[ \text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \]

Here \(a\) and \(b\) are “type variables”. They start with lowercase letters. (Contrast: Actual data types like `Bool` are capitalized.)

The user chooses what types to use for \(a\) and \(b\).
Think of \(\forall a, b \cdot (a \rightarrow b) \rightarrow [a] \rightarrow [b]\)
The implementer cannot choose. Cannot ask either.

(Exercise: Can you use \text{map} as \((X \rightarrow Y) \rightarrow [Z] \rightarrow [T]?\))

Algebraic data types can be parametrized by type variables too, e.g., `Either` is defined by:
\[ \text{data Either } a \text{ } b \text{ } = \text{ Left } a \mid \text{ Right } b \]
I may like to generalize my stack type to:
\[ \text{data Stack } a \text{ } = \text{ Bottom } \mid \text{ Push } a \text{ (Stack } a) \]
Type-Class Polymorphism: Motivation

\( \text{map can be } (a \rightarrow b) \rightarrow [a] \rightarrow [b] \) because it really doesn’t care.

But:
\( (==), (<) \) cannot be \( a \rightarrow a \rightarrow \text{Bool} \).
Comparison algorithms vary by types.
\( (+), (*) \) cannot be \( a \rightarrow a \rightarrow a \).
Arithmetic implementations vary by types.
etc., etc.

So how does Haskell pull these off?
"hello" < "hi"
(4.5 :: Double) == (4.6 :: Double)
4.5 + 4.6 :: Double
4 - 5 :: Int
Type-Class Polymorphism
Haskell’s Disciplined Answer to Overloading

A “type class” declares overloaded operations (“methods”).

-- From the standard library.
class Eq a where
  (==), (/=) :: a -> a -> Bool

A type can be made an “instance” of that class. This means the type supports the overloaded functions.

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

Exercise: Write code to make MyType an instance of Eq.

Warning: class ≠ type. Eq is not a type. Bool is not a subclass. Eq -> Eq -> ... makes no sense.
Type Signatures of Overloaded Functions

Type signatures of overloaded operations:

\((==), (\neq) :: \text{Eq}\; a \Rightarrow a \rightarrow a \rightarrow \text{Bool}\)

The “\(\text{Eq}\; a \Rightarrow\)” is a “class constraint”.

The user chooses what type to use for \(a\).
But the chosen type has to be an instance of \(\text{Eq}\).
Think of \(\forall a \cdot a \in \text{Eq} \Rightarrow a \rightarrow a \rightarrow \text{Bool}\)

Constraints propagate down the dependency chain:

\[\text{eq3}\; x\; y\; z = x == y \; \&\& \; y == z\]
\[\text{eq3} :: \text{Eq}\; a \Rightarrow a \rightarrow a \rightarrow \text{Bool}\]

\[\text{diff3}\; x\; y\; z = \text{not}\; (\text{eq3}\; x\; y\; z)\]
\[\text{diff3} :: \text{Eq}\; a \Rightarrow a \rightarrow a \rightarrow a \rightarrow \text{Bool}\]
Constrained Instances

How do \([1,3] == [1,3]\) and "ab"=="ab" work? Answer:

-- From the standard library.
instance Eq a => Eq [a] where
    [] == []       = True
    (x:xs) == (y:ys) = x==y && xs==ys
    _ == _          = False

Constraints propagate down the dependency chain, including other instance implementations.

\(x==y\) causes the constraint \(\text{Eq } a\).

\(xs==ys\) is recursion as usual.)
Example User-Defined Class

class ADT a where
    tag :: a -> String

-- tag :: ADT a => a -> String

instance ADT (Either a b) where
    tag (Left _) = "Left"
    tag (Right _) = "Right"

instance ADT MyType where
    tag Nada = "Nada"
    tag (Duplet _ _) = "Duplet"
    tag (Uno _) = "Uno"
Some More Standard Classes

```haskell
class Eq a => Ord a where
    (<>), (<=), (>), (>=) :: a -> a -> Bool
    compare :: a -> a -> Ordering
data Ordering = LT | EQ | GT
```

The “Eq a =>” there means: Every Ord instance is also an Eq instance. ("Superclass" and "subclass".)

- For implementers of types and instances: It’s a prerequisite.
- For users: You can take it for granted.

```haskell
foo x y z = x==y && y<z
foo :: Ord a => a -> a -> a -> Bool
```

Does not have to write (Eq a, Ord a) => a -> a -> a -> Bool
(But you can.)
Some More Standard Classes

class Enum a where
    enumFromTo :: a -> a -> [a]
    ...

This method is behind [1..10] and ['a'..'z']
i.e., Int, Integer, Char are instances.

class Show a where
    show :: a -> String
    ...

Human-readable string representation of your value. Ah but what do you mean “human”?

- Your programmer colleagues?
- End-users?

More likely programmer colleagues, since the REPL uses it too.
Auto-Generating Instance Implementations

The compiler is willing to write some instance code for you, for selected standard classes: Eq, Ord, Enum (but no fields allowed), Show, a few others.

data MyType = Nada | Duplet Double String | Uno Integer deriving (Eq, Ord, Show)

data Browser = Firefox | Chrome | Edge | Safari deriving (Eq, Ord, Enum, Show)

Exercise: Find out what the auto-generated implementations do.
Haskell’s Number System

See the section on numbers in the Haskell Report. Notables:

Number literals like 42 has type `Num a => a`.
Those like 42.0 has type `Fractional a => a`.

Why does the following fail?

```
let mylist :: [Double]
    mylist = [1, 2, 3]
in sum mylist / length mylist
```

Answer: `(6 :: Double) / (3 :: Int)` illegal. Fix: Use `fromIntegral :: (Integral a, Num b) => a -> b` e.g., `sum mylist / fromIntegral (length mylist)`

“You’ve got an Int, but you want Double. Who do ya call? fromIntegral!”
Functor: Motivation

\[ \text{fmap\_List} :: (a \to b) \to [a] \to [b] \]
\[ \text{fmap\_List} = \text{map} \quad -- \text{You know the drill}. \]

-- Maybe is like list of length \(\leq 1\)
\[ \text{fmap\_Maybe} :: (a \to b) \to \text{Maybe} a \to \text{Maybe} b \]
\[ \text{fmap\_Maybe} f \text{ Nothing} = \text{Nothing} \]
\[ \text{fmap\_Maybe} f \text{ (Just} \ a) = \text{Just} \ (f a) \]

\[ \text{fmap\_Either} :: (a \to b) \to \text{Either} \ e \ a \to \text{Either} \ e \ b \]
\[ \text{fmap\_Either} f \text{ (Left} \ e) = \text{Left} \ e \]
\[ \text{fmap\_Either} f \text{ (Right} \ a) = \text{Right} \ (f a) \]

Also arrays, key-value dictionaries, …

Pattern: \(f : a \to b\) induces a corresponding \(F a \to F b\), where \(F\) is a parametrized type.

There is a class for that!
Functor

-- The standard library has:
class Functor f where
    fmap :: (a -> b) -> f a -> f b
instance Functor Maybe where ...
-- NOT: Functor (Maybe a)
instance Functor [] where ...
instance Functor (Either e) where ...

This is nothing like you have ever seen.

You may have seen: Generalize from Maybe Int to Maybe a.

I’m now telling you: Generalize from Maybe Int to f Int.

This is like saying: Generalize in Java from List<Int> to...what?
(Can’t be done. C++ neither.)

Warning exercise: Collection<Int> does not cut it. Why?
Functor: Discussion

Every instance of Functor should satisfy:

\[
\text{fmap } \text{id} \ \text{xs} = \ \text{xs}
\]
\[
\text{fmap } g \ (\text{fmap } f \ \text{xs}) = \ \text{fmap} \ (g \ . \ f) \ \text{xs}
\]

Equivalently:

\[
\text{fmap } \text{id} = \ \text{id}
\]
\[
\text{fmap } g \ . \ \text{fmap } f = \ \text{fmap} \ (g \ . \ f)
\]

DON’T DO THIS:

\[
\text{instance } \text{Functor } \text{Foo where}
\]
\[
\text{fmap } f \ \text{ErrorA} = \ \text{ErrorB}
\]
\[
\ldots
\]

(if your type is \text{data } \text{Foo } a = \ \text{ErrorA} \ | \ \text{ErrorB} \ | \ \text{Good } a)

Lastly, \text{fmap} has an infix alias \text{<$>}, \ e.g., \ \text{sin} \ \text{<$>} [1,2,3]
Applicative: Motivation

You now become ambitious. You ask: What if you have a binary operator, and two lists or two maybes or...

\[
\text{listCross} :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]
\]
\[
\text{listCross} = \ldots \quad -- \text{You did this.}
\]

\[
\text{maybeBoth} :: (a \rightarrow b \rightarrow c) \rightarrow
\]
\[
\quad \text{Maybe a} \rightarrow \text{Maybe b} \rightarrow \text{Maybe c}
\]
\[
\text{maybeBoth \ op \ (Just \ a) \ (Just \ b) = Just \ (op \ a \ b)}
\]
\[
\text{maybeBoth \ op \ _ \ _ \ = \ Nothing}
\]

-- A similar thing for Either e.

And what if you have a tenary operator, and three lists...
Can you implement
\[
ap\_List :: [a \rightarrow b] \rightarrow [a] \rightarrow [b]
\]
such that for example
\[
ap\_List [f,g] [1,2,3] = [f 1, f 2, f 3, g 1, g 2, g 3]
\]

Answer: \(ap\_List = listCross (\lambda f -> \lambda x -> f x)\)

Equivalently \(\text{listCross} \ ($\)\)

Now can implement ternary too:

\[
listTenary :: (a->b->c->d) -> [a]->[b]->[c]->[d]
\]
\[
listTenary tenary as bs cs = 
((tenary <$> as) ‘ap\_List‘ bs) ‘ap\_List‘ cs
\]

Exercise: Use \(ap\_List\) to re-implement \(listCross\).

Analogous stories for Maybe, Either…
There is a class for this too!
Applicative

-- The standard library has:
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

instance Applicative [] where
  pure a = [a]
  (<*>) = ... -- basically ap_List

instance Applicative Maybe where
  pure a = Just a
  Just f <*> Just a = Just (f a)
  _ <*> _ = Nothing

instance Applicative (Either e) where
  pure a = Right a
  (<*>) = ...
Applicative: Discussion

Applicative subsumes Functor—can implement `fmap` as:

`fmap f xs = pure f <*> xs`

In practice it may be more efficient to implement `fmap` directly.

There are also equations that every Applicative instances should satisfy. See the library docs for details.
Monad: Precursor

You have been thinking of List, Maybe, Either... as data structures (containers, collections, whatever).

I now want you to think of them as programs.

- `foo :: Maybe Int` means: a program that may return a number successfully, or may abort.
- `foo :: [Int]` means: a non-deterministic program that returns different numbers in different parallel universes. (Sometimes there are 0 universes—abort.)
- `foo :: Either String Int` means: like Maybe, but if it aborts, it uses `Left` to tell you an error message (say).

Now re-read Functor and Applicative from this angle.
Monad: Motivation

`fmap abs foo` now means: return the absolute value of what `foo` returns.

`(+) <$> foo <*> bar` means: return the sum of what `foo` and `bar` returns.

But `bar` doesn’t know what `foo` returns, or vice versa.

You now become ambitious. Can you combine two programs such that the return value(s) of the 1st is fed to the 2nd, so the 2nd can behave dependently?

Like

`bind :: F a -> (a -> F b) -> F b`

so you can have: `prog1st ‘bind‘ prog2nd`?

(Can think of `prog2nd` as a callback.)
Monad: Preparation

Bind for maybe:

bind_Maybe :: Maybe a -> (a -> Maybe b) -> Maybe b
bind_Maybe Nothing _ = Nothing
bind_Maybe (Just a) k = k a

Bind for List:

bind_List :: [a] -> (a -> [b]) -> [b]
bind_List [] _ = []
bind_List (a:as) k = k a ++ bind_List as k

Actually just

bind_List as k = concat (map k as)

Similarly for Either, etc.

There is a class for that too!
Monad

-- The standard library has:
class Applicative f => Monad f where
  return :: a -> f a
  (>>=) :: f a -> (a -> f b) -> f b

instance Monad [] where
  return a = [a]
  as >>= k = concat (map k as)

instance Monad Maybe where ...

instance Monad (Either e) where ...

Remark: return and pure should be the same thing. Historically Monad came first, Applicative came later, thus the redundancy. There is a proposed change to make return an alias of pure.
Monad: Discussion

Monad subsumes Applicative: Can implement (<*> ) as

\[ \text{fs} \ <\*\> \ \text{as} = \text{fs} \ >>\>
\[ \quad \ f \ -> \ \text{as} \ >>\>
\quad \quad \ a \ -> \ return \ (f \ a) \]

There are equations for Monad too, see the docs, but e.g.,

\[ \text{return} \ a \ >>\> = \text{k} = \text{k} \ a \]

There is “do-notation” so code looks nicer and the computer emits

\[ \text{“} >>\> = \backslash v \ -> \text{”} \text{ for you:} \]

\[ \text{fs} \ <\*\> \ \text{as} = do \]
\[ \quad f \ <- \text{fs} \]
\[ \quad a \ <- \text{as} \]
\[ \quad \text{return} \ (f \ a) \]

(As usual you could also use \{;\} instead of layout. Style guide: As usual, prefer layout.)
Haskell I/O System

Parametrized type “I0” for all I/O commands. Instance of Monad, Applicative, Functor.

```haskell
foo :: I0 Char means: a program that interacts with the outside world, then returns a character (or gets stuck forever, or throws an exception).

putStrLn :: String -> I0 ()
getLine :: I0 String
-- NOT: getLine :: String

DO NOT THINK “how to extract the string”. Use >>= to feed it to the next program (callback).

main = getLine >>= \s -> putStrLn ("It’s " ++ s)
OR
main = do
  s <- getLine
  putStrLn ("It’s " ++ s)
```
Further Reading on Common Type Classes And I/O

Common type classes:
https://wiki.haskell.org/Typeclassopedia

My I/O tutorial:
http://www.vex.net/~trebla/haskell/IO.xhtml