Outline

1. Using GHCI – the Glasgow Haskell compiler interactive system
   [screen/master2.tex]
2. Basic expressions and primitive data types. Integers, reals, characters, unit.
3. Tuples, lists, list comprehension, basic functions
4. Patterns
5. Polymorphism (lists, functions), type variables.
6. Curry, uncurry, isomorphism [fun.tex]
7. Higher-order functions, partial application, map, fold, type reconstruction
8. Algebraic data types [haskell_data.tex]
9. Main program, interact, compilation [haskell.tex]
Canonical value. A canonical value is one which cannot be rewritten further. For example, 2+3 is not canonical, it evaluates to 5; 5 is a canonical value. For example, not True is not canonical, it evaluates to False; False is a canonical value. See canonical in the “The on-line hacker Jargon File,” version 4.4.7, 29 Dec 2003.
Recall. The principle, Haskell primitive data types are: Integer (arbitrary precision),
double (floating-point), char, bool, () (unit).
The Haskell pre-defined, compound data types are: tuples, lists, functions.
The programmer wants to define their own data types. Haskell has one primary
mechanism for creating data types. The types so-created are called algebraic types.
Algebraic type definitions are simple, easy and influencing programming language
design.
Data, the values the programmer computes with, is divided up into distinct collections called data types.

We write:

\[ \text{data value :: data type} \]

\[ \text{data type :: *} \]

to mean that that data value has that data type. Occasionally we write:

\[ \text{data type :: *} \]

to mean that that data type has kind *, i.e., it is a data type.
Data is manipulated by functions according to rules described by function definitions.

Data is constructed by *constructors* according to rules described by data type definitions.

We now examine data type definitions. Data type definitions in Haskell give rise to an elegant system of types called algebraic data types.
Here are four simple examples of data type definitions (which are called enumerated types in other programming languages):

```haskell
data Bool  = True | False
data Color = Red  | Green | Blue
data Unit  = Unit
data Day   = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Types and constructors capitalized.

These definitions completely describe the data types `Bool`, `Color`, `Unit`, and `Day`. These types have exactly two, three, one, and seven elements (data values), respectively.
These data type definitions introduce the elements of the data type. Here is a list of five of the 13 elements introduced above.

\[
\begin{align*}
\text{True} & \::= \ Bool \\
\text{False} & \::= \ Bool \\
\text{Red} & \::= \ Color \\
\text{Unit} & \::= \ Unit \\
\text{Wed} & \::= \ Day \\
\end{align*}
\]
You will have noticed the ambiguity of the identifier `Unit`

\[
\text{Unit} :: \text{Unit} \\
\text{Unit} :: \ast
\]

Since the universe of data types is kept separate from the universe of data values, this will cause little confusion. Since it is sometime burdensome to create name, it is not unusual to use the same name in both universes. A data type with one element is less useful, than the very common data type `Bool`.
A new data type might be desired whose elements are constructed out of elements of another data type. This is more useful, usually, than simply enumerating all the elements.

Next is an example of food with nuts and colored dye.
data Food = Cake Bool | Cookies Bool | CandyCane Color

Cake ... are constructors. New items are constructed by applying the constructors to elements of other data types. In this simple example only a finite number, seven, food items are constructable.

Cake True :: Food  -- cake with nuts
Cake False :: Food  -- cake without nuts
Cookies True :: Food
Cookies False :: Food
CandyCane Red :: Food  -- candy cane with red dye
CandyCane Green :: Food  -- candy cane with green dye
CandyCane Blue :: Food  -- candy cane with blue dye
Constructors are special functions. The application of constructors, like functions in general, is indicated syntactically by juxtaposition. Thus,

\texttt{Cookies True}

is an expression consisting of a function applied to a data value.

Construction of data is the same as function application.
Constructors, like all programs/functions, are data values. The data type to which they belong depends on the domain and on the range.

Cake :: Bool -> Food
Cookies :: Bool -> Food
CandyCane :: Color -> Food

where the arrow is a constructor of data types, in other words, The binary, infix arrow is an example of a type constructor. It is way of representing a data type by combining two other data types, the domain and the range of the function. Without it we are lost . . .

(->) :: * => (* => *)

Eventually we will define our own type constructors.
Our universe is now filled with an infinite number of data types. Here are just a few:

\[
\begin{align*}
\text{Bool} &: \ast \quad -- \text{Bool is a data type} \\
\text{Bool} \to \text{Food} &: \ast \quad -- \text{Bool} \to \text{Food is a data type} \\
\text{Bool} \to \text{Bool} &: \ast \quad -- \text{Bool} \to \text{Bool is a data type} \\
(\text{Bool} \to \text{Bool}) \to \text{Bool} &: \ast \\
\text{Bool} \to (\text{Bool} \to \text{Bool}) &: \ast \\
\text{Bool} \to ((\text{Bool} \to \text{Bool}) \to \text{Bool}) &: \ast \\
\text{Bool} \to ((\text{Bool} \to \text{Bool}) \to \text{Bool}) \to \text{Bool} &: \ast
\end{align*}
\]

We take \((\to)\) is be right-associative so that we may omit some parentheses.

Elements of data types constructed with \((\to)\) are said to be functions.

It is not surprising that we can construct functions by means other than data type definitions. These functions are not called constructors. Only those program/functions explicitly introduced in data type definitions are called constructors.
Only constructors (and not other functions) can be used in patterns.

All constructors (even especially user-defined constructors) can be used in patterns.
Pattern Matching

Functions can be constructed by a list of equations/rules for transforming one data value to another data value.

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool}
\]

\[
\begin{align*}
\text{not True} &= \text{False} \\
\text{not False} &= \text{True}
\end{align*}
\]
Pattern Matching

Functions can be constructed by list of equations/rules for transforming one data value to another data value.

```haskell
containsNuts :: Food -> Bool
containsNuts (Cake True) = True
containsNuts (Cookies True) = True
containsNuts (CandyCane Red) = False
```

Omitting a case is bad (may result in a error at runtime.)
Enumerating all the possibilities is tedious:

```haskell
containsNuts :: Food -> Bool
containsNuts (Cake True) = True
containsNuts (Cake False) = False
containsNuts (Cookies True) = True
containsNuts (Cookies False) = False
containsNuts (CandyCane Red) = False
containsNuts (CandyCane Green) = False
containsNuts (CandyCane Blue) = False
```

For this reason we have variables.
Enumerating all the possibilities is tedious:

```haskell
containsNuts :: Food -> Bool
containsNuts (Cake True) = True
containsNuts (Cookies True) = True
containsNuts x = False
```

The order of the equations/rules matter. The last equation/rule applies to any data value constructed other than the ones listed previously. It is an “else” or “otherwise” case — a catch-all.
Finally, because naming a variable and not referring to it, is in somewhat odd, Haskell permits anonymous variables (called the wildcard pattern).

```haskell
containsNuts :: Food -> Bool
containsNuts (Cake True) = True
containsNuts (Cookies True) = True
containsNuts _ = False
```
Assuming we have a data type \texttt{Float}, we may wish to use it as the size of circle and square shapes.

\begin{verbatim}
data Shape = Circle Float | Square Float
\end{verbatim}

\begin{align*}
\text{area (Circle } r\text{)} &= \pi r^2 \\
\text{area (Rectangle } s1\ s2\text{)} &= s1\times s2
\end{align*}
Assuming we have a (complex) data type \( J \) and \( P \) for images. We can define a data type for multiple kinds of images.

```haskell
data Image = JPEG J | PNG P
```
Suppose you want to model a room with a light and two light switches
data State = On | Off deriving Eq

data System = Switches State State

isOn (Switches x y) = if x==y then False else True

Or, simply:

isOn (Switches x y) = x/=y

Type and constructor names must be capitalized. If equality is desired, then a deriving clause is needed. Same for printing: the deriving (Eq, Show) clause is needed.
Deriving

```haskell
data Color = Red | Green | Blue
  deriving (Show, Eq)

data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Show, Eq, Ord)
```

Show allows Haskell to print data structures. Eq permits equality testings. Ord derives (the obvious) order on the data values.

If you are using these type definitions in the interactive system, one will most likely want Show at least.
You may wonder why we don’t define tuples next. Tuples seem a lot easier and more intuitive that higher-order functions. And it certainly is possible.

\((\ ,\ )\ ::\ *\Rightarrow\ *\Rightarrow\ *

But we are going to introduce higher-order functions anyway, so we might as well dispense with tuples conceptually.

This simplification was used by Frege, Schönfinkel, and Curry. The data values (programs/functions) in \(A \times B \rightarrow C\) are isomorphic to \(A \rightarrow B \rightarrow C\). So, we don’t need tuples.
data BoolPair = BoolPair Bool Bool

BoolPair :: Bool -> Bool -> BoolPair
BoolPair True :: Bool -> BoolPair
(BoolPair True) False :: BoolPair

f :: Bool -> Bool -> BoolPair
f x y = BoolPair (not x) (not y)

-- projection function
first :: BoolPair -> Bool
first (BoolPair x _) = x

Just as easy and natural as positional correspondence in traditional multi-argument functions.
Application Is Left Associative

All this

blah blah blah blah blah

does get overwhelming.

Remember application is left associative:

(((blah blah) blah) blah) blah

And skilled Haskell programs can use techniques to make code more readable.
data MixedPair = MixedPair Day Color

data MixedTriple = MixedTriple BoolPair Day Color

data MixedUnion = Pair MixedPair | Triple MixedTriple

-- 21 mixed pairs + 84 mixed triple = 105
data MixedPair = MixedPair Day Color

data MixedTriple = MixedTriple BoolPair Day Color

data MixedUnion = Pair MixedPair | Triple MixedTriple

-- 21 mixed pairs + 84 mixed triple = 105
Type constructors can take types as parameters.

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

maybe :: b -> (a -> b) -> Maybe a -> b
maybe n _ Nothing = n
maybe _ f (Just x) = f x
```

Maybe is a highly significant (and predefined) type constructor (kind *->*) in Haskell.
See the class `Optional<T>` introduced in Java 8.
Haskell Types

Type constructors can take types as parameters.

\[
data \text{Either} \ a \ b = \text{Left} \ a \mid \text{Right} \ b
\]

\[
either :: (a \to c) \to (b \to c) \to \text{Either} \ a \ b \to c
\]

\[
either \ f \ _ \ (\text{Left} \ x) = f \ x
\]

\[
either \ _ \ g \ (\text{Right} \ y) = g \ y
\]
Data types can be recursive, as in the following:

```haskell
data T = Con1 Bool | Con2 T

data Bad = Con Bad
```

Contrary to those bad examples, recursive and polymorphic types are the “bees knees,” the “cat’s pajamas,” and the “the snake’s hips.”

```haskell
data Nat = Nil | Succ Nat
data IList = Nil | Cons Integer IList
data PolyList a = Nil | Cons a (PolyList a)
```
Haskell Trees

A list is just a tree with one branch.

```
data Nat  = Zero    | Succ Nat
data List = Nil     | Cons () List
```

Natural numbers are isomorphic to unit lists. See examples of trees in Hudak, PPT, Ch7.
Haskell Trees

The simplest binary tree

data SimpleTree = SimLeaf | SimBranch SimpleTree SimpleTree

SimLeaf :: SimpleTree
SimBranch SimLeaf SimLeaf :: SimpleTree
SimBranch SimLeaf (SimBranch SimLeaf SimLeaf) :: SimpleTree
SimBranch (SimBranch SimLeaf SimLeaf) SimLeaf :: SimpleTree
Haskell Trees

data IntegerTree = IntLeaf Integer |
                        IntBranch IntegerTree IntegerTree

IntLeaf 4 :: IntegerTree
IntLeaf 8 :: IntegerTree
IntBranch (IntLeaf 23) (IntLeaf 45) :: IntegerTree

IntBranch (IntLeaf 7) (IntBranch (IntLeaf 23) (IntLeaf 45)) (IntLeaf 78)
Haskell Trees

data InternalTree a = ILeaf | IBranch a (InternalTree a) (InternalTree a)
data Tree a = Leaf a | Branch (Tree a) (Tree a)
data FancyTree a b = FLeaf a | FBranch b (FancyTree a b) (FancyTree a b)
data GTree = GTree [GTree]
data GPTree a = GPTree a [GPTree a]
data BinQuadTree = Zero | One |
  Quads BinQuadTree BinQuadTree BinQuadTree BinQuadTree BinQuadTree
Nested Types

```haskell
data List a = NilL | ConsL a (List a)
data Nest a = NilN | ConsN a (Nest (a,a))
data Bush a = NilB | ConsB a (Bush (Bush a))

data Node a = Node2 a a | Node3 a a a
data Tree a = Leaf a | Succ (Tree (Node a))
```

Hinze, Finger Trees.
Haskell classes are roughly similar to a Java interface. Like an interface declaration, a Haskell class declaration defines a protocol for using an object rather than defining an object itself. C++ and Java attach identifying information (such as a VTable) to the runtime representation of an object. In Haskell, such information is attached logically instead of physically to values, through the type system. There is no access control (such as public or private class constituents) built into the Haskell class system. Instead, the module system must be used to hide or reveal components of a class.
\[ e_1 \gg\gg e_2 \]

\[(\gg\gg) :: \text{Monad } m \Rightarrow m a \rightarrow m b \rightarrow m b\]

\[ e_1 \gg e_2 = e_1 \gg\gg (_\_ \rightarrow e_2) \]
do { e }

do { x<-e; es } e >>= \x -> do {es}

do { e; es } e >>= \x -> do {es}

do {let ds; es} let ds in do {es}
\[ (x, \text{bar}) \mid (x,y) \leftarrow \text{foos}, \]
\[ x < 2, \]
\[ \text{bar} \leftarrow \text{bars}, \]
\[ \text{bar} < y \]

\begin{verbatim}
do (x,y) <- foos
    guard (x < 2)
    bar <- bars
    guard (bar < y)
    return (x, bar)
\end{verbatim}

\begin{verbatim}
foos >>= \(x, y) ->
    guard (x < 2)
    >> bars
    >>= \bar ->
    guard (bar < y)
    >> return (x, bar)
\end{verbatim}
=  "is define to be" / "is define as"
== "is equal to" / "equals/equals"
_   whatever / wildcard pattern
@   as
:: has type
:  cons

\ lambda

++  append
!!  index
.   compose / dot

<*>  ap(PLY)
==  bind
>>  then
<=< left fish / left Kleisli composition operator
=>> right fish / right Kleisli composition operator
Two major tenets of problem solving with a computer.

1. Decompose complex problems into simple problems and compose the solutions.

2. Programs are functions of the inputs.

Both of these in clear display in a functional language.

The second one gets lost in the fog of programs as sequence of mutating actions, specifically printing.
Main Haskell

Functional programming requires that everything be a function. A Haskell main program is a function from strings to strings. We think of programs as being functions from the input to the output. Nonetheless, it is strange at first to think of the IO behavior of a programming as function from the input string to the output string.