2 Haskell

2.1 Hugs 3
2.2 Values & Expressions 4
2.3 Lists 7
2.4 Functions 13
2.5 Higher Order Functions 20
2.6 Types & Polymorphism 24
2.7 Data Types & Pattern Matching 31
2.8 Type Classes 42

What is Haskell?

• Strongly typed (lazy) functional programming language
• Any computation or program is a function
  \( f :: S \rightarrow T \) (\( S, T \) are types)
• Running a program, computing some value:
  applying a function to value(s)
• Each value has a type
• Functions are values, too \( \Rightarrow \) can be
  arguments and results of functions
  (higher order functions)
• haskell.org
2.1 Hugs

hugs [<option> ...] [<file> ...]

<expr> evaluate expression

?: help on interpreter commands
:set help on command line options
:set +t print type after evaluation
:l <file> load module from specified file
:t <expr> print type of expression
:names [pat] list names currently in scope
:q quit

2.2 Values & Expressions

The basic entities of Haskell are **values**. Expressions are obtained by applying functions to values (expressions will be evaluated until a value is obtained).

> 3
3 :: Integer

> 'c'
'c' :: Char

> 4.0 + 7
11.0 :: Double

> not True
False :: Bool

> 5 < 6
True :: Bool

simple values
Conditional

The conditional is an expression, not a statement!

```haskell
> if 2>3 then 4 else 5
5 :: Integer

> 1 + if True then 1 else 2
2 :: Integer
```

Exercise: What is the value of the following expression?

```haskell
> if if 1<2 then 3<2 else 4<5 then 'a' else 'b'
... ?
```

Complex Values

Complex values: use constructors

Types of complex values are written like values

Tuples:

```haskell
> (3,True)
(3,True) :: (Integer,Bool)

> (3.1,(False,0))
(3.1,(False,0)) :: (Double,(Bool,Integer))
```

Constructors can also be applied to expressions

```haskell
> (17-8,4>5)
(9,True) :: (Integer,Bool)
```
2.3 Lists

The most important complex values are lists. Lists are written using square brackets.

> [2,3,4]
[2,3,4] :: [Integer]

> [2,3,4.0]
[2.0,3.0,4.0] :: [Double]

Elements of a list can be arbitrarily complex values, e.g., tuples or other lists.

> [(1,'a'),(0,'b')]
[(1,'a'),(0,'b')] :: [(Integer,Char)]

> [[1,2],[3],[]]
[[1,2],[3],[]] :: [[Integer]]

Lists are homogeneous

Unlike tuples, all elements in a list must have the same type.

> ['a',True]
ERROR - Type error in list
*** Expression   : [a',True]
*** Term        : True
*** Type        : Bool
*** Does not match : Char

> ['b',[]]
ERROR - Type error in list
*** Expression   : [b',[]]
*** Term        : []
*** Type        : [a]
*** Does not match : Char
Expressions in Lists

As with tuples (and all other constructors), we can put expressions into lists.

```haskell
> [not False]
[True] :: [Bool]
> [7-3, if True then 1 else 2]
[4,1] :: [Integer]
```

Note: String = [Char]

```haskell
> ['O', 'S', 'U']
"OSU" :: [Char]
```

List Expressions

```haskell
> [1..5]
[1,2,3,4,5] :: [Integer]
> ['a'..'e']
"abcde" :: [Char]
> [1,3..8]
[1,3,5,7] :: [Integer]
> [7,6..4]
[7,6,5,4] :: [Integer]
> [0,0,3..1]
[0,0,3,0,6,0,9] :: [Double]
```

Exercise: What are the values of the following expressions?

```haskell
> [2,3..2]
[2]
> [2,2..3]
[2,2,2,2,2,2,2]```

```haskell
> [2,2..2]
[2,2,2,2,2,2,2]```
**List Comprehensions**

> \([n \* n \mid n \leftarrow [1..9]]\)

\([1,9,25,49,81] :: [\text{Integer}]\)

> \([(x,y) \mid x \leftarrow [1..3], y \leftarrow [\text{`a'..'c'}]]\)

\([(1,\text{`a'}),(1,\text{`b'}),(1,\text{`c'}),(2,\text{`a'}),..., (3,\text{`c'})] :: [(\text{Integer,Char})]\)

> \([(x,y) \mid x \leftarrow [1..4], y \leftarrow [1..4], x<y]\)

\([(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)] :: [(\text{Integer, Integer})]\)

**Exercise:** Write a list comprehension denoting the black squares of a chessboard.

(Hint: use the functions `ord` and `even`)

- `ord 'A' = 65`
- `even 2 = True`

---

**List Operations**

> `head [1..4]`

`1 :: \text{Integer}`

> `tail [1..4]`

`[2,3,4] :: [\text{Integer}]`

> "abcde"!!3

---

> "index"

> 'd' :: \text{Char}

> `99:[4,5,6]`  

---

> "cons"

`[99,4,5,6] :: [\text{Integer}]`

> `[4,5]++[8,9]`  

---

> "concat"

`[4,5,8,9] :: [\text{Integer}]`

---

**Exercise:** Evaluate the following expressions

> `head "bull":tail "cat"

---

... ?

> `[null (tail [1]),head [null []]]`

---

... ?
2.4 Functions

- Functions are values (like numbers or lists)
  ⇒ functions can be used in expressions, and, in particular, functions can be applied
- Anonymous functions are written using lambda-notation

Exercise: Write a function to produce the list of odd numbers between \(x\) and \(y\)

Definitions cannot be entered directly in Hugs; they must be written in a file which can be loaded into Hugs
Function Definitions

Example: Define predicate "divides"
(3 divides 9, but 3 does not divide 10)

C Programmer L. Imp

\[
divides :: (\text{Integer, Integer}) \rightarrow \text{Bool} \\
divides (i,j) = \text{if } j \mod i = 0 \text{ then True else False}
\]

backquotes allow infix notation

equality predicate

Haskell Programmer B. Fun

\[
divides :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Bool} \\
divides i j = j \mod i = 0
\]

Functions: Currying

Advantages of 2\textsuperscript{nd} definition:
\begin{itemize}
  \item can be used in infix notation
  \item can be partially applied!
\end{itemize}

\[
\begin{align*}
\text{even} :: & \text{Integer} \rightarrow \text{Bool} \\
\text{even} &= \text{divides 2}
\end{align*}
\]

useful in generating functions "on the fly",
\text{e.g.,} filtering even numbers:

\[
\text{filter (divides 2) [1..19]}
\]
Functions: Sectioning

Most binary functions have curried definitions
⇒ can be partially applied

(1+) successor
(-1) predecessor
(==0) is-zero
(1/) reciprocal
(/2) halving

again, useful in generating function arguments:

map (*2) l

Exercise: What does the following function do?

f x = (++[x])

Functions: Recursion

Example: Adding numbers of a list

\[
\begin{align*}
\text{sum} & : [\text{Integer}] \rightarrow \text{Integer} \\
\text{sum} \ [\ ] & = 0 \\
\text{sum} \ (x:xs) & = x + \text{sum} \ xs
\end{align*}
\]

1. scrutinize data structure
2. apply selector functions

Preview: Pattern Matching accomplishes both tasks

Exercise: Define a function

\[
\begin{align*}
\text{maxl} & : [\text{Integer}] \rightarrow \text{Integer} \\
\text{fac} & : \text{Integer} \rightarrow \text{Integer} \\
\text{fac} \ 0 & = 1 \\
\text{fac} \ n & = n \times \text{fac} \ (n-1)
\end{align*}
\]
### Evaluation

**Eager evaluation** (call-by-value)

```haskell
sqr (2+3)
= sqr 5  { def. + }
= 5*5    { def. sqr }
= 25    { def. * }
```

**Normal order evaluation** (call-by-name)

```haskell
sqr (2+3)
= (2+3)*(2+3)  { def. sqr }
= 5*(2+3)      { def. + }
= 5*5          { def. + }
= 25           { def. * }
```

**Ignore (1/0)**

= *** ERROR ***

If **eager** evaluation terminates with a result, it is the same as under normal order evaluation.

**Normal order** terminates with a result whenever possible.

**Lazy** evaluation implements normal order with equal or less reduction steps than eager evaluation.

### 2.5 Higher Order Functions

**Functions just as a result**

**plus** :: Integer -> (Integer -> Integer)

```
plus x y = x+y  [≡ plus x = \y->x+y]
```

```
plus 1 = \y->1+y  { def. plus }
```

### Functions as a parameter and as a result

```
twice f x = f (f x)  [≡ twice f = \x->f (f x)]
twice (plus 1)
= \x->(plus 1) ((plus 1) x)  { def. twice }
= \x->(plus 1) (\y->1+y) x)  { def. plus }
= \x->(plus 1) (1+x)  { application }
= \x->(\y->1+y) (1+x)  { def. plus }
= \x->(1+(1*x))  { application }
= \x->2+x  { algebra }
```
Function Composition

\[ f \cdot g = \lambda x \rightarrow f(g(x)) \]

- **twice** \( f = f \cdot f \)
- **last** = head . reverse
- **odd** = not . even
- **pow4** = \( \text{sqr} \cdot \text{sqr} \)

**Point-free definitions**
(no object variables)
Rightarrow can often simplify proofs/transformations

**Exercise:** Define a function 
\( \cdot \cdot \) that composes a unary with a binary function. Give point-free definitions for 
"nand" and "mean" using \( \cdot \cdot \).

\[ f : g = \lambda x \rightarrow f(g(x)) \]

- **nand** = not . (\&\&) / 2 : (+)
- **mean** = (/2) : (+)

---

Higher Order List Functions

- **map** is like a for-loop

\[
\begin{align*}
\text{map } f \; [] & = [] \\
\text{map } f \; (x:xs) & = f \; x : \text{map } f \; xs \\
\text{filter } p \; [] & = [] \\
\text{filter } p \; (x:xs) \; | \; p \; x & = x : \text{filter } p \; xs \\
& \text{ otherwise } = \text{filter } p \; xs
\end{align*}
\]

**Guard**

map and filter as list comprehensions:

- **map** \( f \; xs = [f \; x \; | \; x \leftarrow xs] \)
- **filter** \( p \; xs = [x \; | \; x \leftarrow xs, p \; x] \)
More List Functions

```
zip (x:xs) (y:ys) = (x,y):zip xs ys
zip xs     ys     = []
zipWith f (x:xs) (y:ys) = f x y:zipWith f xs ys
zipWith f xs     ys     = []
```

Exercise: Define "zip" as an instance of "zipWith".

---

2.6 Types & Polymorphism

**Static Typing:** all types are determined at compile time $\Rightarrow$ No runtime type errors

**Strong Typing:** each value has a unique type

**Generic Functions/Reuse:**
parametric polymorphism & systematic/well-defined overloading

type class  type variable
Type Examples

Monomorphic types

2 :: Integer
(True,1.0) :: (Bool,Double)
"abc" :: [Char]
\( \text{x} \rightarrow \text{x} + 1 :: \text{Integer} \rightarrow \text{Integer} \)
[not,(\&\& True)] :: [Bool \rightarrow \text{Bool}]

Polymorphic types

Parametric polymorphism

\( \text{x} \rightarrow \text{x} :: \text{a} \rightarrow \text{a} \)
\( \text{x} \rightarrow \text{x} + 1 :: \text{Num a} := \text{a} \rightarrow \text{a} \)

Overloading

All type names begin with a capital letter

Type Synonyms

just abbreviations,
not abstract types
(e.g. \text{f} :: \text{String} \rightarrow ... accepts
\text{String} as well as \text{[Char]} )

\begin{verbatim}
\text{type String} = \text{[Char]}
\text{type Point} = \text{(Float,Float)}
\text{type Line} = \text{[Point]}
\end{verbatim}

helpful for documentation:

\begin{verbatim}
\text{intersect} :: \text{[\text{(Float,Float)}]} \rightarrow \text{[\text{(Float,Float)}]} \rightarrow \text{[\text{(Float,Float)}]}
\text{intersect} = ...

\text{intersect} :: \text{Line} \rightarrow \text{Line} \rightarrow \text{[Point]}
\text{intersect} = ...
\end{verbatim}

Type synonyms provide some support for program updates!
Polymorphic Types

- `length [] = 0`
- `length (x:xs) = 1 + length xs`

- `length :: _ -> _` { has 1 parameter }
- `length :: _ -> Integer` { 1st result is 0 :: Integer }
- `length :: [a] -> Integer` { 1st parameter is a list }
- `length :: [a] -> Integer` { element type not constrained }

  ⇒ function works for any type

  type variable ranges over all types

Types can tell a lot about functions:

- `f :: (a,b) -> a`  
- `g :: (a,b) -> (b,a)`  
- `h :: [a] -> [b] -> [(a,b)]`  
- `i :: (a -> b) -> a -> b`

Parametric polymorphism

Type-directed programming:
First, write down the type; then code often follows naturally

Inferring Types

- `map f [] = []`
- `map f (x:xs) = f x : map f xs`

- `map :: _ -> _ -> _` { has 2 parameters }
- `map :: _ -> [a] -> [b]` { 2nd par. & 1st result is [] :: [] & elem. types might be different }
- `map :: (_ -> _) -> [a] -> [b]` { 1st parameter is a function }
- `map :: (a -> b) -> [a] -> [b]` { f is applied to elem. of par. list & result of f is put into result list }

Exercise: Infer the type of `twice`:

```
twice f x = f (f x)
```
More Polymorphic Types

head :: [a] -> a
reverse :: [a] -> [a]
(++) :: [a] -> [a] -> [a]
filter :: (a -> Bool) -> ([a] -> [a])
map :: (a -> b) -> ([a] -> [b])
(,) :: (b -> c) -> (a -> b) -> (a -> c)

Exercise: Infer the types of "f" and "zip":

\[
\begin{align*}
f \, xs &= [ \ x \ | \ x \leftarrow xs, \ \text{head} \ x &= 'a' \ ] \\
\text{zip} \ (x:xs) \ (y:ys) &= (x,y):\text{zip} \ xs \ ys
\end{align*}
\]

Type Constructors

\begin{align*}
\text{type} \ Queue \ a &= [a] \\
\text{type} \ Assoc \ a \ b &= [(a,b)] \\
\text{type} \ Monoid \ a &= (a,a \rightarrow a \rightarrow a)
\end{align*}

parameterized types are called type constructors

parameterized types are still just abbreviations, not abstract types ...

... but again helpful for documentation and update support:

\begin{align*}
enqueue :: a \rightarrow Queue \ a \rightarrow Queue a \\
enqueue \ x &= (++[x]) \\
lookup :: a \rightarrow Assoc a b \rightarrow b \\
lookup &= ...
\end{align*}

(abstract data types can be realized by using data types and modules)
2.7 Data Types & Pattern Matching

Data Types are general; they implement:
- Enumeration types
- Union types
- Recursive data structures
- A form of "lightweight" encapsulation

Data Types are versatile; they can:
- have their own printable representation
- be made instances of type classes
  ⇒ reuse overloaded functions

Enumeration Types

```
data Grade = A | B | C | D | F
=data Color = Red | Green | Blue
=data Bool = True | False
```

Constructors, must start with capital letter

Constructors are values:

```
> A
ERROR: Cannot find "show" function for:
*** Expression : A
*** Of type : Grade

> A==B
ERROR: Illegal Haskell 98 class constraint in inferred type
*** Expression : A == B
*** Type : Eq Grade => Bool
```
**Enumeration Types**

```haskell
data Grade = A | B | C | D | F
 deriving (Eq, Show, Ord, Enum)
```

Define equality (==) and printing (show) of values based on the term representation.

Constructors can be used as patterns:

```haskell
> A==B      -- Eq
False :: Bool

> A         -- Show
A :: Grade

> [A .. F]  -- Enum
[A,B,C,D,F] :: [Grade]
```

**SADTs: Somewhat Abstract Data Types**

```haskell
data Age = Years Integer

isOld :: Age -> Bool
isOld (Years y) = y>60
```

Prevents, e.g., multiplication of ages (which would be possible when using type synonyms).

```haskell
type Queue a = [a]

data Queue a = Queue [a]

enqueue :: a -> Queue a -> Queue a
enqueue x (Queue xs) = Queue (xs++[x])
```

Unwrap and wrap
Union Types

type Point = (Float, Float)
data Geo = Point | Circle Float | Rect Point Point
   deriving (Eq, Show)

Constructors with argument types are functions:

> :i Circle
Circle :: Point -> Float -> Geo -- data constructor

> Circle (0,1) 3
Circle (0.0,1.0) 3.0 :: Geo

Pattern Matching

area :: Geo -> Float
area (Point _) = 0
area (Circle _ r) = 3.1415*r*r
area (Rect _ (x,y) (x',y')) = abs ((x-x')*(y-y'))

"as"-pattern
check r@(Rect p (x,y) (x',y')) =
   if x>x' & & y<y' then r else error "illegal rectangle!"

check g = g

patterns must be linear
find x (y:xs) = x==y || find x xs
find x (x:xs) = True
   find x (y:xs) = find x xs
Patterns in Definitions

p = (0,0) :: Point
picture = [Circle p, Point p]

Note: Data types enable heterogeneous lists!

Get radius of circle:
c = head picture
r = (\Circle \_ r->r) c

Patterns can be used in lambda-abstractions

Patterns can also be used in definitions:

\[ \text{Circle}_\_ r\_ := \text{picture} \]
\[ \text{Circle}_\_ r\_ := \text{picture} \]

Bindings:
s→"abcd", c1→'a', c2→'b', ...
p→(3,4), x→3, y→4

Recursive Data Types

data Tree = Node Integer Tree Tree
| Leaf

deriving (Eq, Show)

Values are terms!

> Node 3 Leaf (Node 5 Leaf Leaf)
Node 3 Leaf (Node 5 Leaf Leaf) :: Tree

Recursive (or inductive) data types lead to recursive function definitions:

find :: Integer -> Tree -> Bool
find x Leaf = False
find x (Node y l r) | x==y = True
| x < y = find x l
| True = find x r
Polymorphic Data Types

data Tree a = Node a (Tree a) (Tree a)
  | Leaf
  deriving (Eq, Show)

Exercise: What type does the following expression have?

Node Leaf Leaf Leaf

data List a = Cons a (List a)
  | Nil

data [a] = a:[a] | []

Note: Lists are just an example of a polymorphic data type

The Maybe Data Type

data Maybe a = Just a | Nothing
  deriving (Eq, Ord, Show, Read)

saveDiv :: Integer -> Integer -> Maybe Integer
saveDiv m n | n /= 0 = Just (m `div` n)
  | True   = Nothing

saveHead :: [a] -> Maybe a
saveHead (x:xs) = Just x
saveHead [] = Nothing

Exceptional cases are mapped to 'Nothing'

Maybe can be used for dealing with exceptions and errors

How to adjust the rest of a program?

mmmap :: (a -> b) -> Maybe a -> Maybe b
mmmap f (Just x) = Just (f x)
mmmap f Nothing = Nothing
More on Binary Trees

**Exercise:** Define the following functions for the polymorphic Tree data type:

```haskell
data Tree a = Node a (Tree a) (Tree a)
  | Leaf

tmap :: (a -> b) -> Tree a -> Tree b
inorder :: Tree a -> [a]
```

```haskell
tmap :: (a -> b) -> Tree a -> Tree b
    tmap f (Node x l r) = Node (f x) (tmap f l) (tmap f r)
    tmap f Leaf = Leaf

inorder :: Tree a -> [a]
inorder (Node x l r) = inorder l ++ [x] ++ inorder r
inorder Leaf = []
```

2.8 Type Classes

**Type class** = set of types having a set of functions in common

**Defining a type class:** define names and types of required functions (member functions)

**Make a type T an instance of a class C** (i.e., inserting T into C): give implementations for the member functions

**For some classes, instances can be derived automatically**

**A type class can have (multiple) superclasses**

**More:** multi-parameter type classes, constructor classes
The Eq Class

Read as: "Type a is an instance of the Eq class if it has a function (==) or (/=) defined with the shown typing."

class Eq a where

(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool

x == y = not (x /= y)
x /= y = not (x == y)

} member functions

default => defining either definitions (==) or (/=) suffices

instance Eq Color where

Red == Red = True
Blue == Blue = True
Green == Green = True
_ ==_ = False

Read as: "Color is an instance of the Eq class where the definition of (==) is as follows ..."

Eq Class Constraints

type of elem?

elem :: (y:ys) = x==y || elem x ys
elem _ [] = False

has 2 parameters

2nd par. is [] :: [ ] &
result is False :: Bool

1st par. matches list args

(type Assoc a b = [(a,b)]

lookup :: Eq a => a -> Assoc a b -> Maybe b
llookup x ((y,z):ys) | x/=y = lookup x ys
| True = Just z
lookup _ [] = Nothing

Read as: 

forall Eq: a -> ...
The Ord Class

Ord is a subclass of Eq (because default defs. use (==)). To make a type T an instance of Ord, T must already be an instance of Eq.

```haskell
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
  ...
  -- default definitions
```

### Data Ordering

```haskell
data Ordering = LT | EQ | GT
```

- define either (≤) or compare
- (≤) on a
- (≤) on [a]
- (recursive def.)

### Instance Ord a => Ord [a]

```haskell
instance Ord a => Ord [a] where
  [] < (_:__)  = True
  (x:xs) < (y:ys) = x<y && xs<ys
  _ < _ = False
```

- (≤) on [a]
- (recursive def.)

### Instance Ord a => Ord (Tree a)

```haskell
instance Ord a => Ord (Tree a) where
  t < t' = inorder t < inorder t'
```

Ord Class Constraints

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [y | y<xs, y=x] ++ [x] ++
  qsort [y | y<xs, y>x]
```

- Why isn't Eq a needed as a class constraint?

```haskell
find :: Ord a => a -> Tree a -> Bool
find x Leaf = False
find x (Node y l r) |
  x==y = True
  | x < y = find x l
  | True = find x r
```
The Show Class

Types are made instances of Show to provide customized printable representations.

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

instance Show Bool where
  show True = "T"
  show False = "F"

instance Show a => Show (Maybe a) where
  show (Just a) = show a
  show Nothing = "?"

map saveHead [[3],[[],[4,5],[]]] = [3,2,4,?]

Exercise: Define show for trees:
Node 5 (Node 2 Leaf Leaf) Leaf → 5 < 2

Haskell vs. Java Classes

<table>
<thead>
<tr>
<th>Haskell</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>value (no state)</td>
<td>object</td>
</tr>
<tr>
<td>function</td>
<td>method</td>
</tr>
<tr>
<td>type</td>
<td>class</td>
</tr>
<tr>
<td>type class</td>
<td>interface</td>
</tr>
<tr>
<td>defining class instance</td>
<td>implementing interface</td>
</tr>
<tr>
<td>defining subclass</td>
<td>extending interface</td>
</tr>
<tr>
<td>derived instances</td>
<td></td>
</tr>
<tr>
<td>default methods</td>
<td></td>
</tr>
<tr>
<td>multi-parameter type classes</td>
<td></td>
</tr>
<tr>
<td>constructor classes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nested classes/ interfaces</td>
</tr>
</tbody>
</table>