What is Haskell?

- Last week:
  - built-in data types
    - base types, tuples, lists (and strings)
  - writing functions using pattern matching and recursion
- This week:
  - user-defined data types
    - and how to manipulate them using pattern matching and recursion
  - more details about recursion

Representing complex data

- We’ve seen:
  - base types: `Bool`, `Int`, `Integer`, `Float`
  - some ways to build up types: given types `T1`, `T2`
    - functions: `T1 -> T2`
    - tuples: `(T1, T2)`
    - lists: `[T1]`
- Algebraic Data Types: a single, powerful technique for building up types to represent complex data
  - lets you define your own data types
  - subsumes tuples and lists!
Product types

• Tuples can do the job but there are two problems...

deadlineDate :: (Int, Int, Int)
deadlineDate = (2, 4, 2019)

deadlineTime :: (Int, Int, Int)
deadlineTime = (11, 59, 59)

-- | Deadline date extended by one day
extension :: (Int, Int, Int) -> (Int, Int, Int)
extension = ...

• Can you spot them?

1. Verbose and unreadable

type Date = (Int, Int, Int)
type Time = (Int, Int, Int)

deadlineDate :: Date
deadlineDate = (2, 4, 2019)

deadlineTime :: Time
deadlineTime = (11, 59, 59)

-- | Deadline date extended by one day
extension :: Date -> Date
extension = ...

2. Unsafe

• We want this to fail at compile time!!!
  extension deadlineTime

• Solution: construct two different datatypes
  data Date = Date Int Int Int
data Time = Time Int Int Int
  -- constructor^    ^parameter types

  deadlineDate :: Date
deadlineDate = (2, 4, 2019)

  deadlineTime :: Time
deadlineTime = (11, 59, 59)
Record Syntax

- Haskell’s record syntax allows you to name the constructor parameters:

  Instead of
  ```haskell
data Date = Date Int Int Int
```

  You can write:
  ```haskell
data Date = Date { month :: Int, day :: Int, year :: Int }
```

  Use the field name as a function to access part of the data

  deadlineDate = Date { month = 7 } 2019
  deadlineMonth = month deadlineDate

Building data types

- Three key ways to build complex types/values:
  1. Product types (each-of): a value of $T$ contains a value of $T_1$ and a value of $T_2$ [done]
  2. Sum types (one-of): a value of $T$ contains a value of $T_1$ or a value of $T_2$
  3. Recursive types: a value of $T$ contains a sub-value of the same type $T_s$

Example: NanoMD

- Suppose I want to represent a text document with simple markup. Each paragraph is either:
  - plain text (String)
  - heading: level and text (Int and String)
  - list: ordered? and items (Bool and [String])

- I want to store all paragraphs in a list

  ```haskell
doc = [ (1, "Notes from 138") -- Lvl 1 heading
  , "There are two types of languages:" -- Plain text
  , (True, ["purely functional", "purely evil"])
  --^^ Ordered List
  ] -- But this doesn't type check!!
```
Sum Types

• Solution: construct a new type for paragraphs that is a sum (one-of) the three options!
  - plain text (String)
  - heading: level and text (Int and String)
  - list: ordered? and items (Bool and [String])

• I want to store all paragraphs in a list

```haskell
data Paragraph =
  Text String  -- 3 constructors,
  | Heading Int String  -- each with different
  | List Bool [String]  -- parameters
```

QUIZ

What would GHCi say? *
```
data Paragraph =
  Text String  |  Heading Int String  |  List Bool [String]
```
What would GHCi say to
```
>:t Text "Hey there!"
```

- A. Syntax error
- B. Type error
- C. Paragraph
- D. [Paragraph]
- E. [String]

http://tiny.cc/cmps112-para-ind

QUIZ

What would GHCi say? *
```
data Paragraph =
  Text String  |  Heading Int String  |  List Bool [String]
```
What would GHCi say to
```
>:t Text "Hey there!"
```

- A. Syntax error
- B. Type error
- C. Paragraph
- D. [Paragraph]
- E. [String]

http://tiny.cc/cmps112-para-grp
Constructing datatypes

**data** T =
  C1 T11 .. T1k
  | C2 T21 .. T2l
  | ..
  | Cn Tn1 .. Tnm

T is the new datatype

C1 .. Cn are the constructors of T

A value of type T is

- either C1 v1 .. vk with vi :: T1i
- or C2 v1 .. vl with vi :: T2i
- or ...
- or Cn v1 .. vm with vi :: Tni

Constructing datatypes

You can think of a T value as a box:

- either a box labeled C1 with values of types T11 .. T1k inside
- or a box labeled C2 with values of types T21 .. T2l inside
- or ...
- or a box labeled Cn with values of types Tn1 .. Tnm inside

Apply a constructor = pack some values into a box (and label it)

- Text "Hey there!"
  - put "Hey there!" in a box labeled Text
- Heading 1 "Introduction"
  - put 1 and "Introduction" in a box labeled Heading
- Boxes have different labels but same type (Paragraph)

QUIZ

**data** Paragraph =
  Text String | Heading Int String | List Bool [String]

What would GHCi say to

\>`:t [Heading 1 "Introduction", Text "Hey there!"]

- A. Syntax error
- B. Type error
- C. Paragraph
- D. [Paragraph]
- E. [String]

http://tiny.cc/cmps112-adt-ind
QUIZ

What is the type of:

```haskell
data Paragraph =
  Text String | Heading Int String | List Bool [String]
```

What would GHC say to:

```haskell
> :t [Heading 1 "Introduction", Text "Hey there!"]
```

- A. Syntax error
- B. Type error
- C. Paragraph
- D. [Paragraph]
- E. [String]

http://tiny.cc/cmps112-adt-grp

Example: NanoMD

```haskell
data Paragraph =
  Text String | Heading Int String | List Bool [String]
```

Now I can create a document like so:

```haskell
doc :: [Paragraph]
doc = [
  Heading 1 "Notes from 130",
  Text "There are two types of languages:"
  , List True ["purely functional", "purely evil"]
]
```

Example: NanoMD

Now I want convert documents in to HTML.

I need to write a function:

```haskell
html :: Paragraph -> String
html p = ??? -- depends on the kind of paragraph!
```

How to tell what’s in the box?

- Look at the label!
Pattern Matching

Pattern matching = looking at the label and extracting values from the box
  • we've seen it before
  • but now for arbitrary datatypes

```haskell
html :: Paragraph -> String
html (Text str) = ...
  -- It's a plain text! Get string
html (Heading lvl str) = ...
  -- It's a heading! Get level and string
html (List ord items) = ...
  -- It's a list! Get ordered and items
```

Dangers of pattern matching (1)

```haskell
html :: Paragraph -> String
html (Text str) = ...
html (List ord items) = ...
```

What would GHCi say to:
```haskell
html (Heading 1 "Introduction")
```

Answer: Runtime error (no matching pattern)

Dangers of pattern matching (1)

Beware of missing and overlapped patterns
  • GHC warns you about overlapped patterns
  • GHC warns you about missing patterns when called with `-W` (use :set `-W` in GHCi)
Pattern matching expression

We've seen: pattern matching in *equations*

You can also pattern-match *inside your program* using the *case* expression:

```plaintext
html :: Paragraph -> String
html p =
case p of
    Text str -> unlines [open "p", str, close "p"]
    Heading lvl1 str -> ...
    List ord items -> ...
```

---

**QUIZ**

What is the type of *

```plaintext
let p = Text "Hey there!"
in case p of
    Text str -> str
    Heading lvl1 _ -> lvl1
    List ord _ -> ord
```

- A. Syntax error
- B. Type error
- C. String
- D. Paragraph
- E. Paragraph -> String

[http://tiny.cc/cmps112-case-ind](http://tiny.cc/cmps112-case-ind)

---

**QUIZ**

What is the type of *

```plaintext
let p = Text "Hey there!"
in case p of
    Text str -> str
    Heading lvl1 _ -> lvl1
    List ord _ -> ord
```

- A. Syntax error
- B. Type error
- C. String
- D. Paragraph
- E. Paragraph -> String

[http://tiny.cc/cmps112-case-grp](http://tiny.cc/cmps112-case-grp)
Pattern matching expression: typing

The case expression

```plaintext
case e of
  pattern1 -> e1
  pattern2 -> e2
  ...
  patternN -> eN
has type T if
  • each e1...eN has type T
  • e has some type D
  • each pattern1...patternN is a valid pattern for D
    i.e. a variable or a constructor of D applied to other patterns
```

The expression e is called the match scrutinee.

---

QUIZ

What is the type of *

```plaintext
let p = Text "Hey there!"
in case p of
  Text _ -> 1
  Heading _ _ -> 2
  List _ _ -> 3
```

- A. Syntax error
- B. Type error
- C. Paragraph
- D. Int
- E. Paragraph -> Int

http://tiny.cc/cmps112-case2-ind

---

QUIZ

What is the type of *

```plaintext
let p = Text "Hey there!"
in case p of
  Text _ -> 1
  Heading _ _ -> 2
  List _ _ -> 3
```

- A. Syntax error
- B. Type error
- C. Paragraph
- D. Int
- E. Paragraph -> Int

http://tiny.cc/cmps112-case2-grp
Building data types

- Three key ways to build complex types/values:
  1. **Product types** (each-of): a value of $T$ contains a value of $T_1$ and a value of $T_2$ [done]
  2. **Sum types** (one-of): a value of $T$ contains a value of $T_1$ or a value of $T_2$ [done]
  3. **Recursive types**: a value of $T$ contains a sub-value of the same type $T$

Recursive types

Let’s define natural numbers from scratch:

data Nat = ???

data Nat = Zero | Succ Nat

A Nat value is:
- either an empty box labeled Zero
- or a box labeled Succ with another Nat in it!

Some Nat values:

- Zero -- 0
- Succ Zero -- 1
- Succ (Succ Zero) -- 2
- Succ (Succ (Succ Zero)) -- 3
- ...
Functions on recursive types

Principle: Recursive code mirrors recursive data

1. Recursive type as a parameter

data Nat = Zero -- base constructor | Succ Nat -- inductive constructor

Step 1: add a pattern per constructor

toInt :: Nat -> Int
toInt Zero = ... -- base case
toInt (Succ n) = ... -- inductive case
    -- (recursive call goes here)

1. Recursive type as a parameter

data Nat = Zero -- base constructor | Succ Nat -- inductive constructor

Step 2: fill in base case

toInt :: Nat -> Int
toInt Zero = 0 -- base case
toInt (Succ n) = ... -- inductive case
    -- (recursive call goes here)
1. Recursive type as a parameter

```haskell
data Nat = Zero -- base constructor |
          Succ Nat -- inductive constructor
```

Step 3: fill in inductive case using a recursive call:

```haskell
toInt :: Nat -> Int
toint Zero = 0 -- base case
toint (Succ n) = 1 + toInt n -- inductive case
```

---

**QUIZ**

What does this evaluate to? *

```haskell
let foo i = if i <= 0 then Zero else Succ (foo (i - 1))
in foo 2
```

- A. Syntax error
- B. Type error
- C. 2
- D. Succ Zero
- E. Succ (Succ Zero)

http://tiny.cc/cmps112-rectype-ind

---

**QUIZ**

What does this evaluate to? *

```haskell
let foo i = if i <= 0 then Zero else Succ (foo (i - 1))
in foo 2
```

- A. Syntax error
- B. Type error
- C. 2
- D. Succ Zero
- E. Succ (Succ Zero)

http://tiny.cc/cmps112-rectype-grp
2. Recursive type as a result

data Nat = Zero -- base constructor
  | Succ Nat -- inductive constructor

fromInt :: Int -> Nat
fromInt n
| n <= 0 = Zero -- base case
| otherwise = Succ (fromInt (n - 1)) -- inductive -- case

2. Putting the two together

data Nat = Zero -- base constructor
  | Succ Nat -- inductive constructor

add :: Nat -> Nat -> Nat
add Zero m = m -- base case
add (Succ n) m = Succ (add n m) -- inductive case

sub :: Nat -> Nat -> Nat
sub n Zero = n -- base case 1
sub Zero _ = Zero -- base case 2
sub (Succ n) (Succ m) = sub n m -- inductive case

Lessons learned:
- Recursive code mirrors recursive data
- With multiple arguments of a recursive type, which one should I recurse on?
- The name of the game is to pick the right inductive strategy!
Lists

Lists aren’t built-in! They are an algebraic data type like any other:

```haskell
data List = Nil  -- base constructor
  | Cons Int List  -- inductive constructor
```

- List [1, 2, 3] is represented as `Cons 1 (Cons 2 (Cons 3 Nil))`
- Built-in list constructors [] and (:) are just fancy syntax for `Nil` and `Cons`

Functions on lists follow the same general strategy:

```haskell
length :: List -> Int
length Nil = 0  -- base case
length (Cons _ xs) = 1 + length xs  -- inductive case
```

What is the right inductive strategy for appending two lists?

```haskell
append :: List -> List -> List
append Nil ys = ys
append xs ? = ??
```
Lists

What is the right inductive strategy for appending two lists?

\[
\text{append} :: \text{List} \rightarrow \text{List} \rightarrow \text{List}
\]

\[
\text{append \hspace{1em} Nil \hspace{1em} ys = ys}
\]

\[
\text{append \hspace{1em} (Cons \hspace{1em} x \hspace{1em} xs) \hspace{1em} ys = Cons \hspace{1em} x \hspace{1em} (append \hspace{1em} xs \hspace{1em} ys)}
\]

Trees

Lists are unary trees with elements stored in the nodes:

\[
1 \rightarrow 2 \rightarrow 3 \rightarrow ()
\]

\[
data \hspace{1em} \text{List} = \hspace{1em} \text{Nil} \mid \text{Cons} \hspace{1em} \text{Int} \hspace{1em} \text{List}
\]

How do we represent binary trees with elements stored in the nodes?

\[
1 \rightarrow 2 \rightarrow 3 \rightarrow ()
\]

\[
\hspace{1em} | \hspace{1em} | \hspace{1em} \hspace{1em} \hspace{1em} ()
\]

\[
\hspace{1em} | \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} ()
\]

\[
\hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} 4 \rightarrow ()
\]

\[
\hspace{1em} \hspace{1em} \hspace{1em} ()
\]

QUIZ

What is a Haskell datatype for binary trees with elements stored in the nodes?*

\[
1 \rightarrow 2 \rightarrow 3 \rightarrow ()
\]

\[
\hspace{1em} | \hspace{1em} | \hspace{1em} \hspace{1em} ()
\]

\[
\hspace{1em} | \hspace{1em} \hspace{1em} \hspace{1em} ()
\]

\[
\hspace{1em} \hspace{1em} \hspace{1em} 4 \rightarrow ()
\]

\[
\hspace{1em} \hspace{1em} ()
\]

(A) \[
data \hspace{1em} \text{Tree} = \hspace{1em} \text{Leaf} \mid \hspace{1em} \text{Node} \hspace{1em} \text{Int} \hspace{1em} \text{Tree}
\]

(B) \[
data \hspace{1em} \text{Tree} = \hspace{1em} \text{Leaf} \mid \hspace{1em} \text{Node} \hspace{1em} \text{Tree} \hspace{1em} \text{Tree}
\]

(C) \[
data \hspace{1em} \text{Tree} = \hspace{1em} \text{Leaf} \mid \hspace{1em} \text{Node} \hspace{1em} \text{Int} \hspace{1em} \text{Tree} \hspace{1em} \text{Tree}
\]

(D) \[
data \hspace{1em} \text{Tree} = \hspace{1em} \text{Leaf} \hspace{1em} \text{Int} \mid \hspace{1em} \text{Node} \hspace{1em} \text{Tree} \hspace{1em} \text{Tree}
\]

(E) \[
data \hspace{1em} \text{Tree} = \hspace{1em} \text{Leaf} \hspace{1em} \text{Int} \mid \hspace{1em} \text{Node} \hspace{1em} \text{Int} \hspace{1em} \text{Tree} \hspace{1em} \text{Tree}
\]

http://tiny.cc/cmps112-tree-ind
QUIZ

What is a Haskell datatype for binary trees with elements stored in the nodes? *

1 - 2 - 3 - ()
   |   | \ ()
   | \ ()
\ 4 - ()
   \ ()

(A) data Tree = Leaf | Node Int Tree
(B) data Tree = Leaf | Node Tree Tree
(C) data Tree = Leaf | Node Int Tree Tree
(D) data Tree = Leaf Int | Node Tree Tree
(E) data Tree = Leaf Int | Node Int Tree Tree

Trees

1 - 2 - 3 - ()
   |   | \ ()
   | \ ()
\ 4 - ()
   \ ()

data Tree = Leaf | Node Int Tree Tree
t1234 = Node 1
         (Node 2 (Node 3 Leaf Leaf) Leaf)
         (Node 4 Leaf Leaf)

Functions on trees

depth :: Tree -> Int
depth Leaf = 0
depth (Node _ 1 r) = 1 + max (depth l) (depth r)
Binary trees

() - () - () - 1
|   |   \ 2
|   \ 3
\ () - 4
\ 5
data Tree = Leaf | Node Int Tree

t12345 = Node
  (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))
  (Node (Leaf 4) (Leaf 5))
I want to implement an arithmetic calculator to evaluate expressions like:

- $4.0 + 2.9$
- $3.78 - 5.92$
- $(4.0 + 2.9) \times (3.78 - 5.92)$

What is a Haskell datatype to represent these expressions?

data Expr = ???

data Expr = Num Float | Add Expr Expr | Sub Expr Expr | Mul Expr Expr

How do we write a function to evaluate an expression?

eval :: Expr -> Float

eval (Num f) = f
Example: Calculator

data Expr = Num Float
  | Add Expr Expr
  | Sub Expr Expr
  | Mul Expr Expr

How do we write a function to evaluate an expression?

eval :: Expr -> Float
eval (Num f) = f
eval (Add e1 e2) = eval e1 + eval e2

Recursion is...

Building solutions for *big problems* from solutions for *sub-problems*

- **Base case**: what is the *simplest version* of this problem and how do I solve it?
- **Inductive strategy**: how do I *break down* this problem into sub-problems?
- **Inductive case**: how do I solve the problem *given* the solutions for subproblems?

Why use Recursion?

1. Often far simpler and cleaner than loops
   - But not always...
2. Structure often forced by recursive data
3. Forces you to factor code into reusable units (recursive functions)

Why *not* use Recursion?

1. Slow
2. Can cause stack overflow
Example: factorial

\[ \text{fac} :: \text{Int} \to \text{Int} \]
\[ \text{fac } n \]
\[ n = 1 \Rightarrow 1 \]
\[ \text{otherwise } \Rightarrow n \times \text{fac } (n - 1) \]

\[
\text{fac } 4 \\
\Rightarrow 4 \times \text{fac } 3 \\
\Rightarrow 4 \times 3 \times \text{fac } 2 \\
\Rightarrow 4 \times 3 \times 2 \times \text{fac } 1 \\
\Rightarrow 4 \times 3 \times 2 \times 1 \\
\Rightarrow 4 \times 3 \times 2 \\
\Rightarrow 4 \times 6 \\
\Rightarrow 24
\]

Each function call << allocates a frame on the call stack
- expensive
- the stack has a finite size
Can we do recursion without allocating stack frames?

Tail recursion

Recursive call is the top-most sub-expression in the function body
- i.e. no computations allowed on recursively returned value
- i.e. value returned by the recursive call == value returned by function
Tail recursive factorial

Let's write a tail-recursive factorial!

```
facTR :: Int -> Int
facTR n = loop 1 n
  where
    loop :: Int -> Int -> Int
    loop acc n
      | n <= 1    = acc
      | otherwise = loop (acc * n) (n - 1)
```
Tail recursive factorial

loop acc n
| n <= 1   = acc
| otherwise = loop (acc * n) (n - 1)

<facTR 4>
==>
<<loop 1 4>> -- call loop 1 4
==>
<<loop 4 3>> -- rec call loop 4 3
==>
<<<<loop 12 2>>> -- rec call loop 12 2
==>
<<<<loop 24 1>>>> -- rec call loop 24 1
==>
24 -- return result 24!

Each recursive call directly returns the result
• without further computation
• no need to remember what to do next!
• no need to store the “empty” stack frames!

Tail recursive factorial

Because the compiler can transform it into a fast loop

function facTR(n){
    var acc = 1;
    while (true) {
        if (n <= 1) { return acc ; } else {
            acc = acc * n; n = n - 1; }
    }
}

Tail recursive factorial

function facTR(n){
    var acc = 1;
    while (true) {
        if (n <= 1) { return acc ; } else {
            acc = acc * n; n = n - 1; }
    }

• Tail recursive calls can be optimized as a loop
  • no stack frames needed!
• Part of the language specification of most functional languages
  • compiler guarantees to optimize tail calls
That’s all folks!