What is Haskell?

- A typed, lazy, purely functional programming language

  Haskell = λ-calculus +
  - Better syntax
  - Types
  - Built-in features
    - Booleans, numbers, characters
    - Records (tuples)
    - Lists
    - Recursion
    - ...

Why Haskell?

- Haskell programs tend to be simple and correct

  Quicksort in Haskell

  ```haskell
  sort [] = []
  sort (x:xs) = sort ls ++ [x] ++ sort rs
  where
  ls = [ l | l <- xs, l <= x ]
  rs = [ r | r <- xs, x < r ]
  ```

- Goals for this week
  - Understand the above code
  - Understand what typed, lazy, and purely functional means (and why you care)
Haskell vs λ-calculus: Programs

- A program is an expression (not a sequence of statements)
- It evaluates to a value (it does not perform actions)
  - λ:
    \( (\lambda x \rightarrow x) \text{ apple} \quad -- \Rightarrow \text{ apple} \)
  - Haskell:
    \( (\lambda x \rightarrow x) \text{ "apple"} \quad -- \Rightarrow \text{ "apple"} \)

Haskell vs λ-calculus: Functions

- Functions are first-class values:
  - can be **passed as arguments** to other functions
  - can be **returned as results** from other functions
  - can be **partially applied** (arguments passed one at a time)
    \( (\lambda x \rightarrow (\lambda y \rightarrow x \, y)) \, (\lambda x \rightarrow z + 1) \, 0 \quad -- \Rightarrow 2 \)
- **BUT**: unlike λ-calculus, not everything is a function!

Haskell vs λ-calculus: top-level bindings

- Like in Elsa, we can name terms to use them later
- Elsa:
  ```
  let T = \ x \ y \rightarrow x
  let F = \ x \ y \rightarrow y
  
  let PAIR = \ x \ y \rightarrow \ b \rightarrow \ ITE \ b \ x \ y
  let FST = \ p \rightarrow p \ T
  let SND = \ p \rightarrow p \ F
  
  eval FST:
  FST (PAIR \text{ apple} \text{ orange})
  \Rightarrow \text{ apple}
  ```
Haskell vs λ-calculus: top-level bindings

• Like in Elsa, we can name terms to use them later
  • Haskell:
    haskellIsAwesome = True
    pair = \x y -> \b -> if b then x else y
    fst = \p -> p haskellIsAwesome
    snd = \p -> p False

    -- In GHCi:
    > fst (pair "apple" "orange") -- "apple"

• The names are called top-level variables
• Their definitions are called top-level bindings

Syntax: Equations and Patterns

• You can define function bindings using equations:

  pair x y b = if b then x else y -- pair = \x y b -> ...
  fst p = p True "apple" "orange"
  snd p = p False "apple" "orange"

• A single function binding can have multiple equations with different patterns of parameters:

  pair x y True = x -- If 3rd arg matches True,
  -- use this equation;
  pair x y False = y -- Otherwise, if 3rd arg matches
  -- False, use this equation.

• The first equation whose pattern matches the actual arguments is chosen
• For now, a pattern is:
  - a variable (matches any value)
  - or a value (matches only that value)
Syntax: Equations and Patterns

- A single function binding can have multiple equations with different patterns of parameters:

  pair x y True  = x  -- If 3rd arg matches True,  
                     -- use this equation;
  pair x y False = y  -- Otherwise, if 3rd arg matches 
                     -- False, use this equation.

- Same as:

  pair x y True  = x  -- If 3rd arg matches True,  
                     -- use this equation;
  pair x y b    = y  -- Otherwise use this equation.

Equations with guards

- An equation can have multiple guards (Boolean expressions):

```plaintext
cmpSquare x y | x > y*y  = "bigger :)
               | x == y*y = "same :|
               | x < y*y  = "smaller :(
```

- Same as:

```plaintext
cmpSquare x y | x > y*y  = "bigger :)
               | x == y*y = "same :|
               | otherwise = "smaller :("```
Recursion

- Recursion is built-in, so you can write:

```haskell
def sum(n):
    if n == 0:
        return 0
    else:
        return n + sum(n - 1)
```

- Or you can write:

```haskell
def sum(n):
    if n == 0:
        return 0
    else:
        return n + sum(n - 1)
```

Scope of variables

- Top-level variables have global scope

```haskell
message = if haskellIsAwesome
           then "I love CSE 130"
           else "I'm dropping CSE 130"

haskellIsAwesome = True
```

- Or you can write:

```haskell
-- What does f compute?
def f(n):
    if n == 0:
        return True
    else:
        return g(n - 1)  -- mutual recursion!
def g(n):
    if n == 0:
        return False
    else:
        return f(n - 1)  -- mutual recursion!
```

- Answer: f is isEven, g is isOdd

Scope of variables

- Is this allowed?

```haskell
haskellIsAwesome = True

haskellIsAwesome = False  -- changed my mind
```

- Answer: no, a variable can be defined once per scope; no mutation!
Local variables

- You can introduce a new (local) scope using a `let` expression:
  
  ```
  sum 0 = 0
  sum n = let n' = n - 1
           in n + sum n' -- the scope of n'
           -- is the term after in
  ```

- Syntactic sugar for nested `let`-expressions:
  
  ```
  sum 0 = 0
  sum n = let
           n' = n - 1
           sum' = sum n'
           in n + sum'
  ```

Local variables

- If you need a variable whose scope is an equation, use the `where` clause instead:
  
  ```
  cmpSquare x y | x > z = "bigger :)"
                 | x == z = "same :|
                 | x < z = "smaller :(

  where z = y*y
  ```

Types

- What would *Elsa* say?
  
  ```
  let FNORD = ONE ZERO
  ```

- **Answer:** Nothing. When evaluated, it will crunch to `something`, but it will be nonsensical.
  - *λ*-calculus is *untyped*. 
• What would **Python** say?
  ```python
def fnord():
    return 0(1)
```
  • **Answer:** Nothing. When evaluated will cause a run-time error.
  - Python is **dynamically typed**

• What would **Java** say?
  ```java
void fnord() {
    int zero;
    zero(1);
}
```
  • **Answer:** Java compiler will reject this.
  - Java is **statically typed**.

• In Haskell every expression either **has a type** or is ill-typed and rejected statically (at compile-time, before execution starts)
  - like in Java
  - unlike λ-calculus or Python
  ```haskell
fnord = 1 0 -- rejected by GHC
```
Type Annotations

- You can annotate your bindings with their types using ::, like so:

  -- / This is a Boolean:
  haskellIsAwesome :: Bool
  haskellIsAwesome = True

  -- / This is a string
  message :: String
  message = if haskellIsAwesome
             then "I love CMPS 112"
             else "I'm dropping CMPS 112"

- If you omit annotations, GHC will infer them for you
- Inspect types in GHCi using :t
- You should annotate all top-level bindings anyway! (Why?)

Type Annotations

- / This is a word-size integer
  rating :: Int
  rating = if haskellIsAwesome then 10 else 0

- / This is an arbitrary precision integer
  bigNumber :: Integer
  bigNumber = factorial 100

Function Types

- Functions have arrow types
  - \x -> e has type A -> B
  - If e has type B, assuming x has type A
- For example:
  > :t (\x -> if x then 'a' else 'b')
  (\x -> if x then 'a' else 'b') :: Bool -> Char
Function Types

- You should annotate your function bindings:
  
  \[
  \text{sum :: Int } \to \text{ Int} \\
  \text{sum 0 = 0} \\
  \text{sum n = n + sum (n - 1)} \\
  \]

- With multiple arguments:
  
  \[
  \text{pair :: String } \to \text{ (String } \to \text{ (Bool } \to \text{ String))} \\
  \text{pair x y b = if b then x else y} \\
  \]

- Same as:
  
  \[
  \text{pair :: String } \to \text{ String } \to \text{ Bool } \to \text{ String} \\
  \text{pair x y b = if b then x else y} \\
  \]

Lists

- A list is
  
  - either an empty list
    
    \[
    [] -- \text{ pronounced "nil"} \\
    \]
  - or a head element attached to a tail list
    
    \[
    x:xs -- \text{ pronounced "x cons xs"} \\
    \]

Terminology: constructors and values

\[
[] -- \text{ A list with zero elements} \\
1:[] -- \text{ A list with one element: 1} \\
(\_):1[] -- \text{ Same thing: for any infix } op, \\
\quad \text{ -- (op) is a regular function!} \\
1:(2:(3:(4:[]))) -- \text{ A list with four elements: } 1, 2, 3, 4 \\
1:2:3:4:[] -- \text{ Same thing (: is right associative)} \\
[1,2,3,4] -- \text{ Same thing (syntactic sugar)}
\]
Lists

- [] and (:) are called the list constructors
- We’ve seen constructors before:
  - True and False are Bool constructors
  - 0, 1, 2 are... well, it’s complicated, but you can think of them as Int constructors
  - these constructions didn’t take any parameters, so we just called them values
- In general, a value is a constructor applied to other values (e.g., list values on previous slide)

Type of a list

- A list has type [A] if each one of its elements has type A
- Examples:
  ```java
  myList :: [Int]
  myList = [1,2,3,4]
  myList' :: [Char]  -- or :: String
  myList' = ['h', 'e', 'l', 'l', 'o']  -- or = "hello"
  myList'' = [1, 'h']  -- Type error: elements have different types!
  myList''' :: [t]  -- Generic: works for any type t!
  myList''' = []
  ```

Functions on lists: range

```haskell
-- | List of integers from n upto m
upto :: Int -> Int -> [Int]
upto n m
  | n > m    = []
  | otherwise = n : (upto (n + 1) m)
```

- There is also syntactic sugar for this!
  ```haskell
  [1..7]  -- [1,2,3,4,5,6,7]
  [1,3..7]  -- [1,3,5,7]
  ```
Functions on lists: length

-- | Length of the List
length :: ???
length xs = ???

Pattern matching on lists

-- | Length of the List
length :: [Int] -> Int
length [] = 0
length (_:xs) = 1 + length xs

- A pattern is either a variable (incl. _) or a value
- A pattern is
  - either a variable (incl. _)
  - or a constructor applied to other patterns
- Pattern matching attempts to match values against patterns and, if desired, bind variables to successful matches.

Some useful library functions

-- | Is the list empty?
null :: [t] -> Bool

-- | Head of the list
head :: [t] -> t -- careful: partial function!

-- | Tail of the list
tail :: [t] -> [t] -- careful: partial function!

-- | Length of the list
length :: [t] -> Int

-- | Append two lists
(++) :: [t] -> [t] -> [t]

-- | Are two lists equal?
(==) :: [t] -> [t] -> Bool

You can search for library functions (by type!) at hoogle.haskell.org
Pairs

- \((\cdot,\cdot)\) is the pair constructor

  -- Field access using Library functions:
  whichFruit = fs myPair -- "apple"
  howMany   = snd myPair -- 3

  -- Field access using pattern matching:
  isEmpty (x, y) = y == 0
  -- same as:
  isEmpty = \((x, y) -> y == 0
  -- same as:
  isEmpty p = let (x, y) = p in y == 0

Pattern matching with pairs

- Is this pattern matching correct? What does this function do?

  \[
  \begin{align*}
  f & : \text{String} \to [(\text{String}, \text{Int})] \to \text{Int} \\
  f \_ [] & = 0 \\
  f x \ ((k,v) : ps) & \\
  & | x == k = v \\
  & | \text{otherwise} = f x ps
  \end{align*}
  \]

Tuples

- Can we implement triples like in \(\lambda\)-calculus?

- Sure! But Haskell has native support for \(n\)-tuples:

  \[
  \begin{align*}
  \text{myPair} & : (\text{String}, \text{Int}) \\
  \text{myPair} & = ("apple", 3) \\
  \text{myTriple} & : (\text{Bool}, \text{Int}, [\text{Int}]) \\
  \text{myTriple} & = (\text{True}, 1, [1,2,3]) \\
  \text{my4tuple} & : (\text{Float}, \text{Float}, \text{Float}, \text{Float}) \\
  \text{my4tuple} & = (\pi, \sin \pi, \cos \pi, \sqrt{2})
  \end{align*}
  \]

  -- And also:
  \[
  \begin{align*}
  \text{myUnit} & : () \\
  \text{myUnit} & = ()
  \end{align*}
  \]
List comprehensions

- A convenient way to construct lists from other lists:
  
  \[
  \text{[toUpper c | c <- s]} \quad -- \quad \text{Convert string s to upper case}
  \]

  \[
  [(i,j) | i <- [1..3], \quad \text{-- Multiple generators}
          j <- [1..i]]
  \]

  \[
  [(i,j) | i <- [0..5], \quad \text{-- Guards}
          j <- [0..5],
          i + j == 5]
  \]

Quicksort in Haskell

\[
\begin{align*}
\text{sort} & \; [ ] \; = \; [ ] \\
\text{sort} & \; (x:xs) \; = \; \text{sort} \; \text{ls} \; ++ \; [x] \; ++ \; \text{sort} \; \text{rs} \\
\text{where} & \\
\text{ls} & \; = \; [ \; l \; | \; l \; \leftarrow \; xs, \; l \; \leq \; x \; ] \\
\text{rs} & \; = \; [ \; r \; | \; r \; \leftarrow \; xs, \; x \; < \; r \; ]
\end{align*}
\]

What is Haskell?

- A typed, lazy, purely functional programming language
Haskell is statically typed

- Every expression either has a type, or is ill-typed and rejected at compile time
- Why is this good?
  - catches errors early
  - types are contracts (you don’t have to handle ill-typed inputs!)
  - enables compiler optimizations

Haskell is purely functional

- **Functional** = functions are *first-class values*
- **Pure** = a program is an expression that evaluates to a value
  - No side effects! unlike in Python, Java, etc:
    ```haskell
def f(x):
    calls++;
    // side effect!
    System.out.println("calling f"); // side effect!
    launchMissile();              // side effect!
    return x * 2;
```
  - in Haskell, a function of type `Int -> Int` computes a *single integer output* from a *single integer input* and does *nothing else*

Haskell is purely functional

- **Referential transparency:** The same expression always evaluates to the same value
  - More precisely: In a scope where `x1, ..., xn` are defined, all occurrences of `e` with `FV(e) = \{x1, ..., xn\}` have the same value
- Why is this good?
  - easier to reason about (remember `x++` vs `++x` in C?)
  - enables compiler optimizations
  - especially great for parallelization (`e1 + e2`: we can always compute `e1` and `e2` in parallel!)
Haskell is lazy

- An expression is evaluated only when its result is needed
- Example: take 2 [1 .. (factorial 100)]
  take 2 ( upto 1 (factorial 100))
  => take 2 ( upto 1 933262154439...)
  => take 2 (1:(upto 2 933262154439...)) -- def upto
  => 1: (take 1 ( upto 2 933262154439...)) -- def take 3
  => 1: (take 1 (2:(upto 3 933262154439...)) -- def upto
  => 1:2:(take 0 ( upto 3 933262154439...)) -- def take 3
  => 1:2:[] -- def take 1

- Haskell is lazy

- Why is this good?
  - Can implement cool stuff like infinite lists: [1..]
    -- first n pairs of co-primes:
    take n [(i,j) | i <- [1..],
             j <- [1..i],
             gcd i j == 1]
  - encourages simple, general solutions
  - but has its problems too :

That’s all folks!