CS 115
Functional Programming

Lecture 3:

Lists

Today

- More Haskell basics
- Polymorphic types
- Lists
- Functions on lists

More Haskell basics

Equality operators

- Equality is tested using the **==** operator

```
Prelude> 1 == 2
False
```

- Inequality is tested using the **/=** operator

```
Prelude> 1 /= 2
True
```

Equality operators

- Equality/inequality works on values of the same type only for a given application

```
Prelude> 1 /= 'c'
[type error]
```

- Equality can work on many different kinds of types

```
Prelude> 1 /= 2
True
Prelude> 'a' == 'a'
True
```

Equality operators

- Type of equality operators:

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

- The `a -> a -> Bool` part is an example of a *polymorphic type*
- A polymorphic type is parameterized on one or more *type variables* (in this case `a`)
- Recall that (concrete) type names need to be capitalized
- Type variable names start with lower-case letters

Equality operators

- The `a -> a -> Bool` type states that any equality function must take two arguments of the same type (`a`) and return a `Bool`
- The `Eq a =>` part refers to a *type class*
  - Later lecture! Super cool! ☺

Boolean operators/functions

- Some boolean operators include:

```
(||) :: Bool -> Bool -> Bool
(&&) :: Bool -> Bool -> Bool
```

- `||` is logical *or*, `&&` is logical *and*

```
Prelude> True || False
True
Prelude> True && False
False
```

- `not` function works as expected

Negative numbers

- Literal negative numbers (or unary minus operator) sometimes need to be surrounded by parentheses

```
Prelude> -10
-10
Prelude> 3 + -10
[precedence parsing error]
Prelude> 3 + (-10)
-7
```

Negative numbers

- Try:

```
Prelude> abs 10
10
Prelude> abs -10
[nasty error message!]
Prelude> abs (-10)
```

- Haskell thinks that `abs -10` is `10` subtracted from the `abs` function!
- Need parentheses to disambiguate

Pattern guards

- Pattern matching matches on structural features of function arguments
- Sometimes need to test for non-structural features (arbitrary predicates)

```
abs :: Int -> Int
abs x | x < 0 = -x
abs x = x
```

- The `| x < 0` is a *pattern guard*

Pattern guards

```
abs x | x < 0 = -x
```

- This says: "match the argument with `x`, as long as `x` is less than zero"
- With other equation, equivalent to:

```
abs x = if x < 0 then -x else x
```

- Basic structure:

```
<pattern> | <boolean expression> = ...
```

- Where the `<boolean expression>` can depend on names bound in the pattern

Multiple guards

- Can have more than one guard for one pattern:

```
abs x | x < 0  = -x
      | x > 0  = x
      | x == 0 = 0
```

- Guards tried one after another until one works
- Equivalent to:

```
abs x | x < 0  = -x
abs x | x > 0  = x
abs x | x == 0 = 0
```

Pitfall

- This definition:

```
abs x | x < 0  = -x
      | x > 0  = x
      | x == 0 = 0
```

- Causes `ghci` to complain:

```
Warning: Pattern match(es) are non-exhaustive
In an equation for `abs': Patterns not matched: _
```

- What's the problem?

Pitfall

```
abs x | x < 0  = -x
      | x > 0  = x
      | x == 0 = 0
```

- Mathematically, these cases *are* exhaustive
- But Haskell doesn't "know" enough math to check for exhaustive conditions in guards
  - Only checks for exhaustive *patterns*
- Haskell can't *prove* that there is no `x` that doesn't match any of the three guard clauses!

Pitfall

- Fix:

```
abs x | x < 0  = -x
      | x > 0  = x
      | otherwise = 0
```

- `otherwise` is just another name for `True`

```
Prelude> otherwise
True
```

- Haskell knows that this will always match, so cases are *provably* exhaustive

Pattern wildcards

- When you need to match something, but you don't care what the matched value is, use a wildcard pattern (_):

```
three :: a -> Int
three _ = 3
```

- (Note the polymorphic type!)

```
Prelude> three 10
3
Prelude> three 'a'
3
```

Note

- **`ghc(i)`** doesn't check for exhaustive pattern matches by default
- To enable this, use the **`-W`** command-line option, which enables warnings

```
$ ghci -W
```

[now warnings are enabled]

`error`

- Some functions are conceptually *partial*
  - only defined over a subset of the type
- Example: factorial not defined for negative numbers:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

What if we did this?

```
Prelude> factorial (-1)
```

error

- One way to handle this is to make partial functions total (handling all values in the input types) by using the `error` function for missing values:

```
factorial 0 = 1
factorial n | n < 0 = error "bad input"
factorial n = n * factorial (n - 1)
```

- This is a very crude form of error handling
  - alternatives exist (see later in course)

Lists

Lists

- Lists (singly-linked lists) are a pervasive data type in most functional programming languages
- They are the default sequence type in many applications
- Haskell lists:
  - are immutable (cannot change values)
  - have only values of one type
- As a result, lists can share structure without any problems resulting
  - affects design of list processing procedures

Lists

- There are a vast number of list processing functions in the Haskell libraries
- Many are so common that they are included in the Prelude
- Others are generally found in the `Data.List` module

Data.List

- To load the `Data.List` module into a `ghci` session:

```
Prelude> :module +Data.List
Prelude Data.List>
```

- Prompt changes to indicate new module loaded
- To unload:

```
Prelude Data.List> :module -Data.List
Prelude>
```

- (Can use `:m` as abbreviation for `:module`)

Data.List

- To import the **Data.List** module into another Haskell module:

```
import Data.List
```

- This brings all the names in **Data.List** into the local namespace
- Can qualify imported names with either

```
import qualified Data.List
import qualified Data.List as L
```

- First way adds `Data.List.` prefix, second adds `L.` prefix (my preference)

`Data.List`

- Another way to load the `Data.List` module into a `ghci` session:

```
Prelude> import Data.List
Prelude Data.List>
```

- But no `unimport` keyword, so to unimport this module must still do this:

```
Prelude Data.List> :m -Data.List
```

Lists

- Lists have two fundamental operations:
  - construction
  - deconstruction (pattern matching)
- and one fundamental datum:
  - the empty list (`[]`)
- With these, all list operations can be derived

List construction

- Lists are constructed using the `:` (cons) operator, a value, and a list of the same type as the value
- `[]` is a list
- `1 : []` is a list of `Int`s (say)
- `2 : (1 : [])` is a list of `Int`s
  - can write it as `2 : 1 : []` since `:` operator associates to the right
- `3 : 2 : 1 : []` is a list of `Int`s
- Syntactic sugar: write as `[3, 2, 1]`

List construction

- Let's ask **ghci** about the **:** operator:

```
Prelude> :type (:)
(:) :: a -> [a] -> [a]
```

- Note that **:** has a polymorphic type!

```
Prelude> :info :
data [] a = [] | a : [a]
infixr 5 :
```

- Precedence 5, right-associative
- We'll talk about **data** declaration next time

List construction

- So `:` takes a value (of type `a`) and a list of type `a`, and creates a longer list with the value at the front of the list

```
Prelude> 1 : [2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

- Can even write as

```
Prelude> (:) 1 [2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

- though not obvious why you'd want to

List construction

- Haskell lists are not heterogeneous!
  - This isn't Python!

```
Prelude> 'a' : [1, 2, 3]
[type error]
Prelude> [1, 2, 3] : [4, 5, 6]
[type error]
```

- However, can define new data types to give the effect of heterogeneous lists if you need to (next lecture)

List deconstruction

- In order to use lists, we have to be able to "undo" the `:` operator to get the contents of the list
- In some languages, use **`head`**/**`tail`** functions
  - can do this in Haskell too
- Normally, we use pattern matching to deconstruct a list
- What are the "natural" components of a list to pattern-match against?

List deconstruction

- Answer: head of list, and tail of list
- Example function: length of a list

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

- Note the `(x:xs)` on the left-hand side
- The list argument is broken into the head (`x`) and the tail (`xs`), both of which are in scope in the right-hand side of the equation
- Note the polymorphic type!

List deconstruction

- Evaluate `length [1, 2, 3]`:

```
length [1, 2, 3]
length (1 : [2, 3])    -- def'n of :
-- match x with 1, xs with [2, 3]
1 + (length [2, 3])       -- eqn 2
1 + (length (2 : [3]))    -- def'n of :
1 + (1 + (length [3]))    -- eqn 2
1 + (1 + (length (3 : [])))
1 + (1 + (1 + (length [])))
1 + (1 + (1 + (0)))       -- eqn 1
3
```

List deconstruction

- Alternate definition of `length`:

```
length :: [a] -> Int
length [] = 0
length x = 1 + length (tail x)
```

- This is valid, but generally poor style to use `head` or `tail` when you can use pattern matching instead

List deconstruction

- Recall original definition of `length`:

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

- Note that `x` on LHS is never used
  - gives warnings from `ghc` with `-W`
- Better to use _ (wildcard):

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Pattern matching again

- More ways to pattern match over lists:

```
foo :: [a] -> Int
foo [x, y, z] = 3   -- match 3-elem list
foo [1, 2] = 5      -- match only list [1, 2]
foo (x:y:z:rest) = 7
foo w = 9
```

- Pattern matching on lists is very flexible!
- (Would normally use `_` for unused variables on left-hand side)

Some useful list functions

`head` and **`tail`**

```
Prelude> head [1, 2, 3]
1
Prelude> tail [1, 2, 3]
[2, 3]
Prelude> head []
*** Exception: Prelude.head: empty list
```

- Again: do not use **`head`** or **`tail`** where pattern matching is more natural!
- Guess: types of **`head`** and **`tail`**

++ operator

- List concatenation uses the **++** operator:

```
Prelude> [1, 2, 3] ++ [4, 5, 6]
[1,2,3,4,5,6]
Prelude> [] ++ [4, 5, 6]
[4,5,6]
Prelude> [1, 2, 3] ++ []
[1,2,3]
Prelude> :info (++)
(++) :: [a] -> [a] -> [a]
infixr 5 ++
```

++ operator

- Pitfall: What's wrong with this definition?

```
last :: [a] -> a
last (_ ++ [x]) = x
```

- Pattern matching will not work on arbitrary operators! (Haskell isn't Prolog!)
- Will only work on "data constructors" and some built-in special cases, like `:` operator, tuples, and literals like numbers, chars, and strings

++ operator

- Definition of `++` operator:

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

- Could also write like this:

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

- Why not like this?

```
[] ++ ys = ys
(x:xs) ++ ys = [x] ++ (xs ++ ys)
```

concat

- The `concat` function concatenates a list of lists
- One definition:

```
concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ concat xss
```

- We'll see a more elegant definition later

reverse

- The **reverse** function reverses a list
- One definition:

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

- Any problems with this definition?
- Efficiency of this function?

reverse

- Alternate definition:

```
reverse :: [a] -> [a]
reverse xs = iter xs []
  where
    iter :: [a] -> [a] -> [a]
    iter [] ys = ys
    iter (x:xs) ys = iter xs (x:ys)
```

- Efficiency?

reverse

```
iter :: [a] -> [a] -> [a]
iter [] ys = ys
iter (x:xs) ys = iter xs (x:ys)
```

- `iter` is a *tail-recursive* function
  - "linear iterative" in CS 4 terminology
  - means: recursive call (tail call) has no pending operations
  - can be more space efficient in *some* circumstances
    - though lazy evaluation can have counterintuitive effects!
    - see this in assignment

`take` and `drop`

- `take` takes a certain number of elements from the front of a list:

```
Prelude> take 3 [1, 2, 3, 4, 5]
[1, 2, 3]
```

- `drop` "drops" a certain number of elements from the front of a list:

```
Prelude> drop 3 [1, 2, 3, 4, 5]
[4, 5]
```

- *Note*: neither one changes the input list
  - Haskell doesn't allow this (no mutation)

The `..` syntax

- The `..` syntax (not an operator!) constructs enumerations:

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

- Equivalent to `enumFromTo 1 10`

```
Prelude> [1,3..10]
[1,3,5,7,9]
```

- Equivalent to `enumFromThenTo 1 3 10`

Infinite lists

- Haskell, being lazy, has the notion of infinite lists:

```
Prelude> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
```

- `[1..]` is the list of all the positive integers!
  - equivalent to `enumFrom 1`
- Haskell generates `1` (the head) and knows how to generate the rest as needed
- What is the value of this?

```
Prelude> take 10 (drop 10 [1..])
```

Indexing: the `!!` operator

- Indexing on lists is done with the `!!` operator:

```
Prelude> [1,2,3,4,5] !! 0
1
Prelude> [1,2,3,4,5] !! 4
5
```

- This is rarely a good way to use lists

Indexing: the `!!` operator

- Let's work through a definition:

```
(!!) :: [a] -> Int -> a
[] !! _          = error "invalid index"
_ !! n | n < 0 = error "invalid index"
(x:_)  !! 0      = x
(_:xs) !! n      = xs !! (n - 1)
```

- Efficiency?
- A list is not an array; don't use it like one

`takeWhile` and `dropWhile`

- `takeWhile` takes elements from the front of a list as long as some criterion is met
- `dropWhile` drops elements from the front of a list as long as some criterion is met

```
Prelude> takeWhile (> 0) [1, 2, 3, -1, -2]
[1, 2, 3]
Prelude> dropWhile (== 0) [0, 0, 0, 1, 2]
[1, 2]
```

`zip` and `unzip`

- **`zip`** takes two lists and "zips" them together into a list of two-tuples
- **`unzip`** takes a list of two-tuples and "unzips" them into two lists (two-tuple of two lists)

```
Prelude> zip [1, 2, 3] [4, 5, 6]
[(1, 4), (2, 5), (3, 6)]
Prelude> unzip [(0, 1), (2, 3), (4, 5)]
([0,2,4], [1,3,5])
```

`zipWith`

- **`zipWith`** is like **`zip`**, except that it applies a two-argument function to the two-tuples

```
Prelude> zip [1, 2, 3] [4, 5, 6]
[(1, 4), (2, 5), (3, 6)]
Prelude> zipWith (+) [1, 2, 3] [4, 5, 6]
[5,7,9]
```

- Note: **`zipWith`** is a *higher-order function*
  - takes a function (operator) as its argument
  - the **`(op)`** syntax converting an operator to a function is essential here

`and` and `or`

- `and` and `or` are multi-argument generalizations of the `&&` and `||` operators

```
Prelude> :t and
[Bool] -> Bool
Prelude> and [True, True, False]
False
Prelude> :t or
[Bool] -> Bool
Prelude> or [False, False, True]
True
```

Next time

- More on list functions
- Higher-order list functions: `map`, `filter`, `foldr` and friends
- List comprehensions