Introducing Haskell

COS 441 Slides 3B

Agenda

- Last time: Introducing Haskell
  - It's a functional language with
    - a *sophisticated type system*, including type inference
    - *immutable data structures*
    - pure computation
    - lazy evaluation
  - Reasoning about Haskell programs occurs via
    - "*substitution of equals for equals*"
    - this law *always* applies in Haskell, rarely in C or Java
  - Good Haskell programmers use *functional abstraction* often
  - Haskell tools
    - ghci: the top-level interpreter
      - :l to load a file; :t to discover a type; :info discover more info
    - ghc: the Haskell compiler
- Today: More Haskell Basics

HASKELL BASICS: DEFINITIONS & BUILT-IN TYPES

Local Definitions

- We often want to define small helper function within the scope of other, larger functions:

```
foo1 z =
    let triple x = x*3
    in triple z
```

- Or:

```
foo2 z = triple z
    where triple x = x*3
```

Haskell Indentation

- Haskell, like Python, but unlike Java, C or math written in a notebook, has semantically meaningful indentation
- Wrong:

must indent function body

```
copies k n =
if n == 0 then []
else k : copies k (n-1)
```

```
zap z =
   let x = z
   y = z + z
   in x + y
```

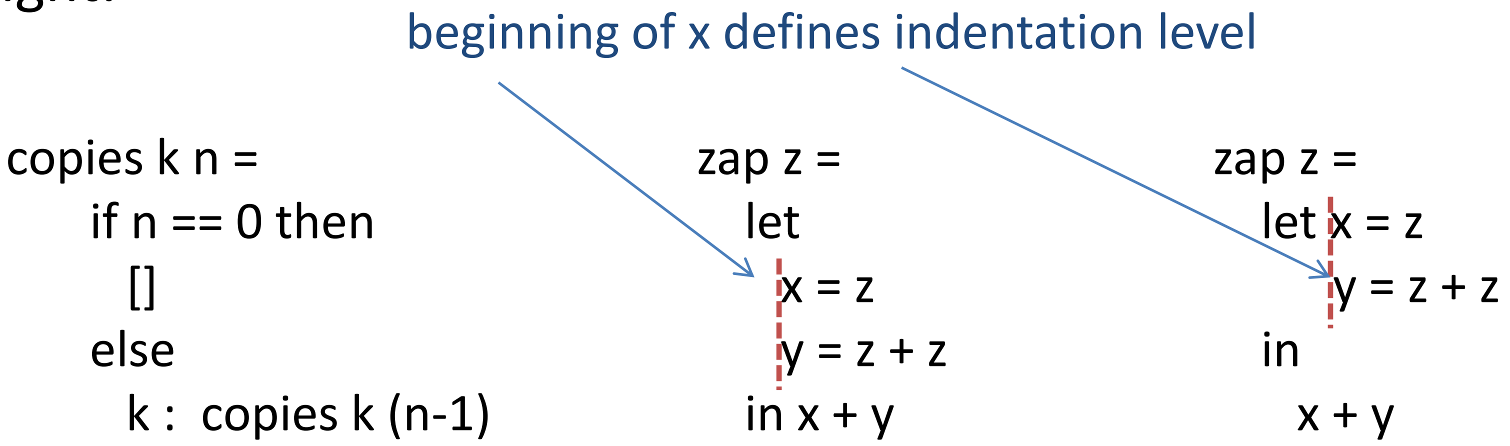
indent y = ...

indent z + z

```
zap z =
   let x = z
      y =
      z + z
   in
    x + y
```

Haskell Indentation

- Haskell, like Python, but unlike Java, C or math written in a notebook, has semantically meaningful indentation
- Right:



beginning of x defines indentation level

```
copies k n =
    if n == 0 then
      []
    else
      k :  copies k (n-1)
```

```
zap z =
  let
    x = z
    y = z + z
  in x + y
```

```
zap z =
  let x = z
      y = z + z
  in
    x + y
```

- Golden rules:
  - let (and where and do) indentation:
    - the first non-whitespace character after let defines the indentation level; subsequent definitions must start at that level
  - Code which is part of some expression should be indented further than the line containing the beginning of that expression

Tuples

- Java uses objects to collect up several different kinds of values; C uses structs
  - both, especially Java objects, are incredibly heavy weight
- Haskell uses tuples
  - constructed by enclosing a sequence of values in parens:

```
('b', 4) :: (Char, Integer)
```

  - deconstructed (used) via pattern matching:

```
easytoo :: (Integer, Integer, Integer) -> Integer
easytoo (x, y, z) = x + y * z
```

Lists

- Lists in Java look very similar to the mathematical list notation we introduced in previous lectures

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

[‘a’, ‘b’, ‘c’] :: [Char]

- [ ] is the empty list (called nil)
- String is a synonym for [Char]
- We can build lists of lists:

[[1, 2], [3], [8, 9, 10]] :: [[Integer]]

- For every type T, we can build lists of type [T]

Lists

- Lists are homogenous; all elements must be the same type

```
Prelude> [ True, 'a' ]

<interactive>:1:9:
    Couldn't match expected type `Bool' with actual type `Char’
    In the expression: 'a'
    In the expression: [True, 'a']
    In an equation for `it': it = [True, 'a']
```

expecting a boolean

actually a character

Constructing Lists

- What do you know, constructing lists in Haskell resembles the mathematical notation we used earlier!
- Building a list:

```
3 : [ 4, 5 ]
```

- Building a list inside a function:

```
add123 xs = 1 : 2 : 3 : xs
```

- Calculating:

```
add123 [ ]
= 1 : 2 : 3 : [ ]
```

```
add123 (3 : [ 4, 5 ])
= 1 : 2 : 3 : (3 : [ 4, 5 ]
= 1 : 2 : 3 : 3 : 4 : 5 : [ ]
```

Functions building lists

```
-- A list of n copies of k

copies :: Integer -> Integer -> [Integer]
copies k n =
   if n == 0 then []
   else k : copies k (n-1)
```

Functions building lists

```
-- A list of n copies of k

copies :: Integer -> Integer -> [Integer]
copies k n =
   if n == 0 then []
   else k : copies k (n-1)
```

```
copies 4 12
=> [12, 12, 12, 12]
```

instead of using “=“
I’ll sometimes use the symbol “=>” which means “evaluates to”

Functions building lists

```
-- A list of n copies of k

copies :: Integer -> Integer -> [Integer]
copies k n =
    if n == 0 then []
    else k : copies k (n-1)



copies 4 12
=> [12, 12, 12, 12]
```

```
-- A list of the numbers from m to n

fromTo :: Integer -> Integer -> [Integer]
fromTo m n =
    if n < m then []
    else m : fromTo (m+1) n
```

Functions building lists

```
-- A list of n copies of k

copies :: Integer -> Integer -> [Integer]
copies k n =
    if n == 0 then []
    else k : copies k (n-1)


copies 4 12
=> [12, 12, 12, 12]
```

```
-- A list of the numbers from m to n

fromTo :: Integer -> Integer -> [Integer]
fromTo m n =
    if n < m then []
    else m : fromTo (m+1) n


fromTo 9 13
=> [ 9, 10, 11, 12, 13 ]
```

Functions deconstructing lists

```
-- Sum the elements of a list

listSum :: [ Integer ] -> Integer

listSum [ ] = 0
listSum (x:xs) = x + listSum xs
```

Functions deconstructing lists

```
-- Sum the elements of a list

listSum :: [ Integer ] -> Integer

listSum [ ] = 0
listSum (x:xs) = x + listSum xs
```

lower case letter =
any type at all
(a type variable)

upper case letter =
a concrete type

```
length :: [a] -> Int

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

Functions deconstructing lists

```
-- Sum the elements of a list

listSum :: [ Integer ] -> Integer

listSum [ ] = 0
listSum (x:xs) = x + listSum xs
```

```
length :: [a] -> Int

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

```
cat :: [a] -> [a] -> [a]

cat [ ] xs2 = xs2
cat (x:xs) xs2 = x:(cat xs xs2)
```

Functions deconstructing lists

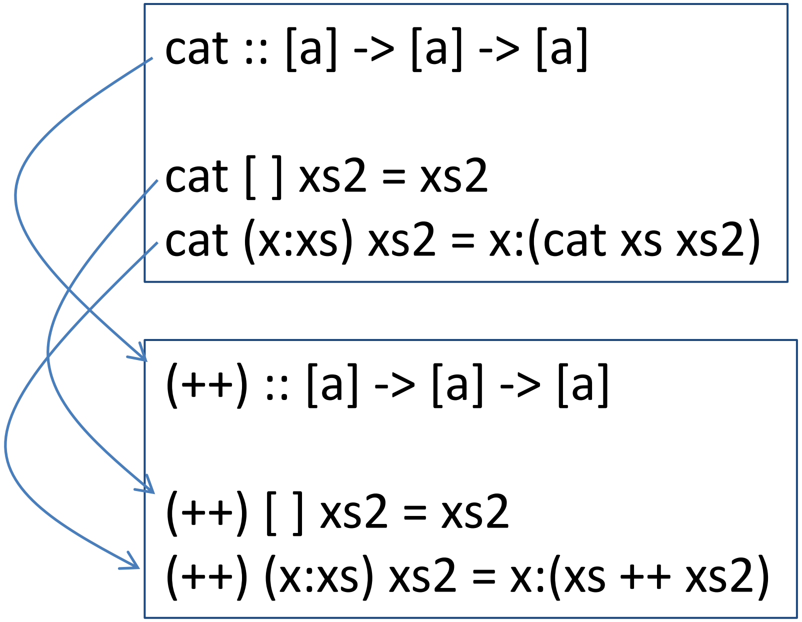
```
-- Sum the elements of a list

listSum :: [ Integer ] -> Integer

listSum [ ] = 0
listSum (x:xs) = x + listSum xs
```

```
length :: [a] -> Int

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



```
cat :: [a] -> [a] -> [a]

cat [ ] xs2 = xs2
cat (x:xs) xs2 = x:(cat xs xs2)
```

```
(++) :: [a] -> [a] -> [a]

(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

INDUCTIVE PROOFS
ABOUT HASKELL PROGRAMS

Recall: Proofs by simple calculation

- Some proofs are very easy and can be done by:
  - unfolding definitions
  - using lemmas or facts we already know
  - folding definitions back up
- Eg:

Definition:
easy x y z = x * (y + z)

given this

we do this proof

Theorem: easy a b c == easy a c b

Proof:

easy a b c

= a * (b + c) (by unfold)

= a * (c + b) (by commutativity of add)

= easy a c b (by fold)

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

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

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

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

case: xs = x:xs’

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

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

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys) (LHS of theorem equation)

case: xs = x:xs’

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

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

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)

case: xs = x:xs’

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

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

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)
= 0 + length (ys)	(simple arithmetic)

case: xs = x:xs’

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

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

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)
= 0 + length (ys)	(simple arithmetic)
= length [ ] + length (ys)	(fold length -- done, we have RHS)

case: xs = x:xs’

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

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

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)
= 0 + length (ys)	(simple arithmetic)
= length [ ] + length (ys)	(fold length)

case: xs = x:xs’

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

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

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)
= 0 + length (ys)	(simple arithmetic)
= length [ ] + length (ys)	(fold length)

case: xs = x:xs’

length ((x:xs’) ++ ys)	(LHS of theorem equation)
= length (x:(xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys))	(unfold length)

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

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

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]

length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)
= 0 + length (ys)	(simple arithmetic)
= length [ ] + length (ys)	(fold length)

case: xs = x:xs’

length ((x:xs’) ++ ys)	(LHS of theorem equation)
= length (x:(xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys))	(unfold length)

subcase xs’ = [ ]

subcase xs’ = x’:xs’’

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

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

Another Theorem

Theorem: For all finite Haskell lists xs and ys,
length(xs ++ ys) = length xs + length ys

Proof attempt:

case: xs = [ ]	
length ([ ] ++ ys)	(LHS of theorem equation)
= length (ys)	(unfold ++)
= 0 + length (ys)	(simple arithmetic)
= length [ ] + length (ys)	(fold length)
case: xs = x:xs’	
length ((x:xs’) ++ ys)	(LHS of theorem equation)
= length (x:(xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys))	(unfold length)
subcase xs’ = [ ]	
…	
subcase xs’ = x’:xs’’	
= 1 + length ((x’:xs’’) ++ ys)	(substitution)
= 1 + length (x’:(xs’’ ++ ys))	(unfold ++)
= 1 + 1 + length (xs’’ ++ ys)	(unfold length)
subsubcase xs’’ = [ ] ….	

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

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

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof strategy:

- Proof by induction on the length of xs
  - must cover both cases: [ ] and x:xs'
    - apply inductive hypothesis to smaller arguments (smaller lists)
    - In general, Haskell has lots of non-inductive data types like Integers (as opposed to Natural Numbers) so you have to be careful all series of shrinking arguments have base cases
  - use folding/unfolding of Haskell definitions
  - use lemmas/properties you know of basic operations

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = [ ]:

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

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = [ ]:
length ([ ] ++ ys) (LHS of theorem)

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

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = [ ]:

length ([ ] ++ ys)	(LHS of theorem)
= length ys	(unfold ++)
= 0 + (length ys)	(arithmetic)
= (length [ ]) + (length ys)	(fold length)

case done!

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

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs’

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

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs’

IH: length (xs’ ++ ys) = length xs’ + length ys

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

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs'

IH: length (xs' ++ ys) = length xs' + length ys

length ((x:xs') ++ ys) (LHS of theorem)

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

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs’

IH: length (xs’ ++ ys) = length xs’ ++ length ys

length ((x:xs’) ++ ys)	(LHS of theorem)
= length (x : (xs’ ++ ys))	(unfold ++)

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

(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs’

IH: length (xs’ ++ ys) = length xs’ + length ys

length ((x:xs’) ++ ys)	(LHS of theorem)
= length (x : (xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys)	(unfold length)

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

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs’

IH: length (xs’ ++ ys) = length xs’ + length ys

length ((x:xs’) ++ ys)	(LHS of theorem)
= length (x : (xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys)	(unfold length)
= 1 + (length xs’ + length ys)	(by IH)

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

(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs’

IH: length (xs’ ++ ys) = length xs’ + length ys

length ((x:xs’) ++ ys)	(LHS of theorem)
= length (x : (xs’ ++ ys))	(unfold ++)
= 1 + length (xs’ ++ ys)	(unfold length)
= 1 + (length xs’ + length ys)	(by IH)
= length (x:xs’) + length ys	(reparenthesizing and folding length)

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

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Proofs over Recursive Haskell Functions

Theorem: For all finite Haskell lists xs and ys,

length(xs ++ ys) = length xs + length ys

Proof: By induction on xs.

case xs = x:xs'

IH: length (xs' ++ ys) = length xs' + length ys

length ((x:xs') ++ ys)	(LHS of theorem)
= length (x : (xs' ++ ys))	(unfold ++)
= 1 + length (xs' ++ ys)	(unfold length)
= 1 + (length xs' + length ys)	(by IH)
= length (x:xs') + length ys	(reparenthesizing and folding length we have RHS with x:xs' for xs)

case done!

All cases covered! Proof done!

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

```
(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
```

Exercises

To test your understanding, try to prove the following:

Theorem 1: for all finite lists xs, ys. listSum(xs ++ ys) = listSum xs + listSum ys

```
drop n [ ] = [ ]
drop n (x:xs) = if n <= 0 then x:xs
                else drop (n-1) xs
```

Theorem 2: for all finite lists xs, natural numbers n and m,

drop n (drop m xs) = drop (n+m) xs

Hint: split the inductive case where xs = x:xs into 3 subcases:

case xs = x:xs:

subcase m = 0 and n = 0: ...

subcase m = 0 and n = n’ + 1 for some natural number n’ (ie: n > 0): ...

subcase m = m’+1 for some natural number m’ (ie: m > 0): ...

Summary

- Haskell is
  - a functional language emphasizing immutable data
  - where every expression has a type:
    - Char, Int, (Char, Int, Float), [Int], [[[(Char, [[Int]])]]]
    - Char -> Int, (Char, Char) -> Int -> [(Char, Int)]
    - String = [Char]
- Reasoning about Haskell programs involves
  - substitution of "equals for equals," unlike in Java or C
  - mathematical calculation:
    - unfold function abstractions
    - push symbolic names around like we do in mathematical proofs
    - reason locally using properties of operations (eg: + commutes)
    - use induction hypothesis
    - fold function abstractions back up
- Homework: Install Haskell. Read LYAHFGG Intro, Chapter 1