More Haskell

- More on lists, lists of lists, list functions including \( \text{take}, \text{drop}, \text{elem} \) and \( \text{!!} \) (get n'th element)
- Defining variables at top level
- Backquotes and parentheses for making functions infix or prefix
- Infinite lists
- \( \text{:} \text{expr} \) to get the type of an expression.
- Use of \texttt{Num} and \texttt{Fractional} to discuss general collections of numbers.

Change ghci prompt

unix> ghci

Prelude> :set prompt "> " -- changes the prompt

Defining interactive variable

- At top level, we can define a variable good to the end of the session (unless you redefine it.)
- Note defining \( x \) doesn’t print out its value. To get the value, use \( x \) as an expression

> \( x = [1,2,3,5,7] \) -- Note no "let" or "in"

> 

> \( x \) -- use \( x \) as an expression to print its value

\[1,2,3,5,7]\]

Continue on with lists in Haskell

> \([\] \) -- empty list

\([]\]

> \([1] \) -- singleton list

\([1]\]

> \([1, 'a'] \) -- need the elements of the list to all have the same type

<interactive>:3:2: error:
- No instance for (\texttt{Num Char}) arising from the literal \texttt{‘1’}
- In the expression: \texttt{1}
  
  In the expression: \texttt{[1, ‘a’]}

  In an equation for \texttt{‘it’}: \texttt{it = [1, ‘a’]}
> [[2], [3, 5]] -- okay to have list containing sublists of different lengths
[[2], [3, 5]]

Various List Functions

> x = [1, 2, 3, 5, 7]
> length x
5
> null x
False
> null []
True
> reverse x
[7, 5, 3, 2, 1]
> reverse (reverse x) == x -- reverse is its own inverse
True
> x
[1, 2, 3, 5, 7]
> minimum x
1
> maximum x
7
> sum x
18
> product x
210

-- Use list comprehensions to run a whole bunch of tests
> [f x | f <- [length, minimum, maximum, sum, product]]
[5, 1, 7, 18, 210]

-- Saw head, tail of list last time. Didn't see init, last
> x
[1, 2, 3, 5, 7]
> init x  -- all but last element
[1, 2, 3, 5]
> last x   -- last element
7

-- Taking head, tail, init, or last of [] yield runtime errors
> tail []
*** Exception: Prelude.tail: empty list
-- take n x returns list of first n elements of list x
-- drop n x returns what remains after omitting first n elements of list x
-- If n > length of x, then take n x and drop n x return []
> x
[1,2,3,5,7]
> take 3 x
[1,2,3]
> take 4 x
[1,2,3,5]
> take 0 x
[]
> take (-1) x -- actually do need those parentheses'^
[]
> take 4 [[]]
[]
> drop 3 x -- list without first three items
[5,7]
> drop 8 x -- 8 is > length of x
[]

\textbf{More functions:} \texttt{!!}, \texttt{elem}

\begin{itemize}
  \item Use \texttt{list !!} \textit{n} to return \textit{n}'th element of a list
\end{itemize}

> x !! 0 -- return first element
1
> x !! 1
2
> x !! 8

*** Exception: Prelude.!!: index too large

-- Is a value an element of a list?
> elem 2 [1,2,3,5,7] -- is 2 in the list?
True
> elem 8 [1,2,3,5,7] -- but not 8
False
> -- You have to ask elem of lists of things that support ==
> elem sqrt [sqrt, sqrt]

^ There aren't negative constants in Haskell: \texttt{-1} is actually a function call of unary \texttt{-} on argument 1.
Saying \texttt{take -1 x} means \texttt{(take -) 1 x}; it tries to run \texttt{take} on function \texttt{-} instead of an integer.
Prefix to Infix; Infix to Prefix

> 2 `elem` [1,2,3,5,7]  -- use as infix operator (surround with backticks)
True
> 3 : x  -- usual notation: infix operator
[3,1,2,3,5,7]
> (:) 3 x  -- use as a prefix function (surround with parens)
[3,1,2,3,5,7]
> (+) 5 2
7

Fancy: List of functions; can’t print out functions

• Let’s try something fancy, a list of arithmetic functions
> fs = [(+), (-), (*), div, rem]  -- list of binary functions
Prelude>
> f = fs !! 1  -- f is second element of f1 i.e., function (-)
> f 14 8  -- Like (-) 14 8
6
> (fs !! 0) 14 8  -- Like (+) 14 8
22

-- Try running all functions in sf on 14 and 8
> [f 14 8 | f <- fs]
[22,6,112,1,6]
> [(+), 14 8, (-) 14 8, (*) 14 8, div 14 8, rem 14 8]  -- got right answer?
[22,6,112,1,6]

-- You can’t print out functions, so (+) and (fs !! 0) cause errors
> (+)

Fancy: List of functions; can’t print out functions

• Let’s try something fancy, a list of arithmetic functions
> fs = [(+), (-), (*), div, rem]  -- list of binary functions
Prelude>
> f = fs !! 1  -- f is second element of f1 i.e., function (-)
> f 14 8  -- Like (-) 14 8
6
> (fs !! 0) 14 8  -- Like (+) 14 8
22

-- Try running all functions in sf on 14 and 8
> [f 14 8 | f <- fs]
[22,6,112,1,6]
> [(+), 14 8, (-) 14 8, (*) 14 8, div 14 8, rem 14 8]  -- got right answer?
[22,6,112,1,6]

-- You can’t print out functions, so (+) and (fs !! 0) cause errors
> (+)

<interactive>:79:1: error:
• No instance for (Show (Integer -> Integer -> Integer))
  arising from a use of ‘print’
  (maybe you haven’t applied a function to enough arguments?)
• In a stmt of an interactive GHCi command: print it
> fs !! 0

<interactive>:80:1: error:
  • No instance for (Show (Integer -> Integer -> Integer))
    arising from a use of ‘print’
    (maybe you haven’t applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it

Infinite Lists

- Easy to define infinite lists. You can’t print them completely out; can only print a finite sublist.
- [1..] is [1,2,3, ...]
- [1,3..] is [1,3,5,7,...]
- cycle list equals list ++ list ++ list ++ ....
- repeat x equals cycle [x]

> r = repeat 17
   -- r = [17,17,17, etc]

> -- Just using > r here would try to print an infinite list
> take 20 r
   -- first 20 elements of r
[17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17]
> s = drop 20 [1..]
   -- using an infinite list range [1, 2, 3, etc]
> take 10 s
[21,22,23,24,25,26,27,28,29,30]

> t = cycle [1,3,5,6]
   -- [1,3,5,6, 1,3,5,6, 1,3,5,6, etc]
> take 20 t
[1,3,5,6,1,3,5,6,1,3,5,6,1,3,5,6,1,3,5,6]
> sum (take 20 s)
75
> squares = [x*x | x <- [1..]]
   -- squares 1, 4, 9, etc
> take 10 squares
[1,4,9,16,25,36,49,64,81,100]
> pairs = [(x,y) | x <- [1..], y <- [1..]]
   -- may not be what you think
> take 10 pairs
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10)]
> [(x,y) | x <- [1..5], y <- [6..10]]
   -- finite list for comparison
[(1,6),(1,7),(1,8),(1,9),(1,10),(2,6),(2,7),(2,8),(2,9),(2,10),(3,6),(3,7),
 (3,8),(3,9),(3,10),(4,6),(4,7),(4,8),(4,9),(4,10),(5,6),(5,7),(5,8),(5,9),
 (5,10)]

-- Easy to write functions that yield infinite lists
> repeat2 v = v : repeat2 v
   -- behaves like repeat
> cycle2 x = x ++ cycle2 x          -- behaves like cycle

-- Or just define a variable recursively
> ones = 1 : ones
> pow2 = 1 : [2*x | x <- pow2]
> take 10 pow2
[1,2,4,8,16,32,64,128,256,512]

> -- Calculate successive approximations of pow2
> 1 : [2*x | x <- [1]]           -- Knowing 1st element lets us calc 2nd
[1,2]
> 1 : [2*x | x <- [1,2]]         -- Knowing 1st two gets us third element
[1,2,4]
> 1 : [2*x | x <- [1,2,4]]       -- So now we can calculate 4th element
[1,2,4,8]
> 1 : [2*x | x <- [1,2,4,8]]     -- And then 5th element; you get idea
[1,2,4,8,16]

- Can use referential transparency to help reason about expressions

- Define
  - take 0 x = []
  - take n (a : b) = a : take (n-1) b
  - pow2 = 1 : [x*2 | x <- pow2]
  - body = [x*2 | x <- pow2]
- Note pow2 = 1 : body
- So take 3 pow2
  = take 3 (1 : body)
  = 1 : take 2 body
  = 1 : take 2 [x*2 | x <- 1 : body]
  = 1 : take 2 ((1*2) : [x*2 | x <- body])
  = 1 : 2 : take 1 [x*2 | x <- body]
  = 1 : 2 : take 1 [x*2 | x <- [x*2 | x <- pow2]]
  = 1 : 2 : take 1 [x*2 | x <- [x*2 | x <- 1 : body]]
  = 1 : 2 : take 1 [x*2 | x <- (1*2) : [x*2 | x <- body]]
  = 1 : 2 : take 1 ((1*2)*2 : [x*2 | x <- [x*2 | x <- body]])
  = 1 : 2 : 4 : take 0 omitted
  = 1 : 2 : 4 : []
  = [1,2,4]

- (Didn’t say referential transparency made reasoning fun, though ;-) ’

---

' Yes, that’s an emoji — it’s actually the original one. The graphics ones you use are second generation.
More List functions: Zip and Map

- zip takes two lists and returns a list of pairs: a pair with the first elements of the two lists, a pair with the second elements, etc.

```haskell
> z = zip [1,2,3] ['a','b','c']
> z
[(1,'a'),(2,'b'),(3,'c')]
> head z
(1,'a')
> z = zip [1,2,3] [6,7,9]
> z
[(1,6),(2,7),(3,9)]
```

- map applies a function to every element of a list

```haskell
> map sqrt [1..5]
[1.0,1.4142135623730951,1.7320508075688772,2.0,2.23606797749979]
> k x = 8 + x
    -- a function on one argument
> map k [1..5]
    -- add 8 everywhere
[9,10,11,12,13]
> map ((+) 8) [1..5]
    -- k is same as function result of (+) 8
[9,10,11,12,13]
> map2 f x = [f v | v <- x]
map2 ((+) 8) [1..3]
= [(+) 8 v | v <- [1..3]]
= [(+) 8 v | v <- [1,2,3]]
= [(+) 8 v | v <- [1]] ++ [(+) 8 v | v <- [2]] ++ [(+) 8 v | v <- [3]]
= [(+) 8 1] ++ [(+) 8 2] ++ [(+) 8 3]
= [9] ++ [10] ++ [11]
= [9, 10, 11]
```

Ask ghci for the Type of an Expression

-- At top level, :t expr will print the type of the expression

-- s is a list of numbers, but Haskell is vague as to what kind of number (Int, Float, ... ?)

```haskell
> :t s
s :: Num a => [a]
```
Some Kinds of Numbers

-- Fractionals are numbers that support division (i.e., /)
-- Built-in types are Float, Double
> :t (3.5 * 4.2)
(3.5 * 4.2) :: Fractional a => a

-- Binary operations like + aren't of type a x a -> a
-- where a is a number type. + is of type a -> a -> a.
-- We'll see the difference in a bit.
> :t (+)
(+) :: Num a => a -> a -> a

-- Division is a binary operation on fractional numbers
> :t (/)
(/) :: Fractional a => a -> a -> a

-- [(+), (-), (*), (/)] is a list of binary operations on numbers
-- (Postpone discussing Integral and Fractional parts that specify what kind of number)

-- A list of binary operations on numbers
> :t [(+), (-)]
[(+), (-)] :: Num a => [a -> a -> a]

Tuples; N-Argument Functions; Currying and Uncurrying

• Tuples are similar to lists but have fixed length. They aren't the same as lists.
  • In Haskell, the pair type of a and b is written (a, b), not a x b
> (1,2)
(1,2)
> :t (1,2)
(1,2) :: (Num a, Num b) => (a, b)
> [1,2] == (1,2)

<interactive>:24:10: error:
  • Couldn't match expected type '[Integer]' with actual type '(Integer, Integer)'
  • In the second argument of '(:=)', namely '(1, 2)'
In the expression: [1, 2] == (1, 2)
In an equation for 'it': it = [1, 2] == (1, 2)
> fst (1,2) -- first element of pair
1
> snd (1,2)  -- second element of pair
2

- In a typical language, a function like \(+\) is of type \(\text{Int} \times \text{Int} \rightarrow \text{Int}\). I.e., it takes a pair of values and returns a value. In Haskell, this type is written \((\text{Int, Int}) \rightarrow \text{Int}\), but Haskell actually uses a different type.
- In Haskell, the type of \((+\) is \(\text{a} \rightarrow \text{a} \rightarrow \text{a}\) (where \(\text{a}\) is a \text{Number} type).
- A function like \(+\) is a one-place function that takes the left operand and returns a function. This function is a takes the right operand of the \(+\) and returns the sum. So \(+\) takes its arguments \textit{one after another}.
- Functions like \(+\) that take their arguments one after another are said to be "\text{curried}".
- The name has nothing to do with spices, it comes from Haskell Curry, the mathematician / logician / CS person for whom the language Haskell is named.

> div 23 6  -- Apply \texttt{div} to 23, then apply the result to 6
2
> (+) 5 7  -- Apply \texttt{+} to 5, then apply the result to 5
12

> :t (+)  -- \((+\) is a function that takes a value \(\text{(of} -- \text{type } \text{a})\) and returns an \(\text{a} \rightarrow \text{a}\) function
(+) :: \text{Num a} => \text{a} \rightarrow \text{a} \rightarrow \text{a}

> :t (+) 5
(+) 5 :: \text{Num a} => \text{a} \rightarrow \text{a}  -- So \((+\) 5 is a function of type \(\text{a} \rightarrow \text{a}\)

> f = (+) 5
> f 7  -- \text{can apply it, just like any other function}
12
> f 8
13

- The binary functions we're familiar with are \textit{uncurried}; they take their two arguments \textit{at the same time}. The function \(\text{g}\) below is the uncurried version of \(+\).

> g (x,y) = x + y  -- add two elements of a pair
> :t g
g :: \text{Num a} => (\text{a, a}) \rightarrow \text{a}

- In Haskell, we pretty much always use curried functions. There are \texttt{curry} and \texttt{uncurry} functions that convert from one to the other.
  - \texttt{curry g} takes its arguments one after another: \((\text{curry g}) 2 \ 4 = 2 + 4 = 6\).
  - \texttt{uncurry (+)} takes it arguments at the same time: \((\text{uncurry (+)}) (2, 4) = 2 + 4 = 6\).

> g (x,y) = x + y  -- Uncurried addition
> :t g
> h x y = x + y  -- Curried addition \(\text{(just like +)}\)
> :t h
> :t (curry g) -- Same type as +
(curry g) :: Num c => c -> c -> c
> :t (uncurry (+)) -- Same type as g
(uncurry (+)) :: Num c => (c, c) -> c

-- Call g on pair of arguments, call h one arg after another
> g (1,6)
7
> h 1 6
7

-- The argument to g doesn't have to be a literal pair; it can be calculated
> z = head [(1,6),(2,4)]
> g z
7
Lecture 2 Activity Questions

1. What is type of value is [(+), (-), (*), div, rem]? 

Write definitions for three list-handling functions

2. firsthalf x = half of list x. E.g., firsthalf [1,2,3,4] = [1,2] 
   If the list is of odd length, ignore the middle element: firsthalf [1,2,3,4,5] = [1,2] 
   Use length x, division by 2, and take. div y 2 is division by 2 dropping remainder 

3. secondhalf x = second half of list x. E.g., secondhalf [1,2,3,4] = [3,4] 
   If the list is of odd length, ignore the middle element: secondhalf [1,2,3,4,5] = [4,5] 
   rem y 2 is remainder of integer division. 
   Possible hint: z + rem y 2 is z+0 or z+1 depending on y being even or odd 

4. pal x = Is x a palindrome? (does first half equal reverse of second half?)

Experiments to try

In Haskell, function composition is dot 
In ghci, what happens on inputs below: Any errors? Why 
> sqrt (sqrt 16) 
> (sqrt . sqrt) 16 
> sqrt sqrt 16 

In ghci, define 
> fs = [sqrt, sqrt . sqrt] 
> head fs -- gives error; which one and why? 

-- what are the values of f1, f2, and g? 
> f1 = head fs 
> f2 = last fs 
> g = tail fs 
-- what do these do and why? 
> f1 16 
> f2 16 
> g 16 
> head g 81 
> f2 16 -- what happens and why?
Activity Solution

1. \[ (+), (-), (*), \text{div}, \text{rem} :: \text{Integral}\ a \Rightarrow \ [a \rightarrow a \rightarrow a] \]

2. firsthalf \( x \) = take \( \text{div} (\text{length } x) \ 2 \) \( x \)
   Take first \( N \) items of \( x \) where \( N \) is \( \text{div} (\text{length } x) \ 2 \), i.e., half the length of 2 ignoring remainder from division of 2.

3. For secondhalf, let's look at an initial buggy version then fix it.
   (Buggy): secondhalf \( x \) = drop \( \text{div} (\text{length } x) \ 2 \) \( x \). This works okay for even-length lists but it includes the middle element for odd-length lists. i.e., secondhalf \( [1, 2, 3, 4, 5] \) = [3, 4, 5].
   If \( N = \text{div} (\text{length } x) \ 2 \), then for even-length \( x \), we want to drop the first \( N \) elements. For odd-length \( x \), we want to drop the first \( N+1 \) elements. Turns out we can drop the first \( N + \text{rem} (\text{length } x) \ 2 \) elements because the remainder is 0 for even lengths and 1 for odd lengths. So the correct solution is
   secondhalf \( x \) = drop \( \text{div} (\text{length } x) \ 2 + \text{rem} (\text{length } x) \ 2 \) \( x \)

4. The palindrome testing is straightforward. We need the parentheses below because function application is left associative.
   pal \( x \) = firsthalf \( x \) \( == \) reverse (secondhalf \( x \))

Some tests - you can verify that

- secondhalf \( [1, 2, 3, 4, 5, 6] \) = \( [4, 5, 6] \)
- buggy secondhalf \( [1, 2, 3, 4, 5, 6, 7] \) = \( [4, 5, 6, 7] \) (bug: includes middle element 4)
- fixed secondhalf \( [1, 2, 3, 4, 5, 6, 7] \) = \( [5, 6, 7] \) (omits middle element)
- Return true: pal \( [] \), pal \( [1] \), pal \( [1, 1] \), pal \( [1, 2, 1] \), pal \( [1, 2, 3, 2, 1] \)
- Also true (since strings are lists of characters): pal \"\" , pal \"1\" , pal \"11\" , pal \"121\"
- Return false: pal \"12\" , pal \"1312\"