Intro to programming in Haskell

Computational Linguistics (LING 455)

Rutgers University September 7, 2021

A little bit of Haskell

Firing up the Haskell interpreter

```
> ghci
GHCi, version 8.10.4: https://www.haskell.org/ghc/
Prelude>

Prelude> 2+3
5

Prelude> take 20 "My very educated mother just served"
"My very educated mot"

Prelude> drop 20 "My very educated mother just served"
"her just served"
```

Writing .hs scripts

```haskell
-- W2.hs (a comment!)
-- Edit this with a text editor (VS Code is great!)

myList :: [Int]   -- myList is a list of Ints
myList = [1..100] -- types can generally be left off

sumList :: [Int] -> Int
sumList []     = 0
sumList (n:ns) = n + sumList ns -- a recursive function
--                   ^^^^^^^       defined using itself!
```

Interpreting .hs scripts

```
> ghci W1.hs
GHCi, version 8.10.4
[1 of 1] Compiling Main ( W1.hs, interpreted )
Ok, one module loaded.

*Main> sumList myList
5050
```

Types in Haskell: preventing many errors

```
*Main> not True
False

*Main> not 2
<interactive>:7:5: error:
  • No instance for (Num Bool) arising from ‘2’
  • In the first argument of ‘not’, namely ‘2’
    In the expression: not 2
    In an equation for ‘it’: it = not 2
```

Not all languages work in this way. For example, Javascript:

```
>> !2
false
```

Declaring types

```
str1 :: String
str1 = "Simon" -- Strings go in double-quotes

str2 :: [Char]
str2 = str1 -- Strings are just lists of characters

char1 :: Char
char1 = 'S' -- Single characters are in single quotes
```

Generally you do not *need* to declare types. In most cases `ghci` can figure out the type on its own. Types are useful for your reader (as code **documentation**), and also to you the coder:

- A type is a program **specification**: a program's type gives a hint (sometimes a big one) about how the program should be defined.

Interlude: two kinds of equality

When I write `str2 = str1`, I am *defining* `str2` to be whatever `str1` is. This presupposes that `str1` is already defined.

This is not the same as a true-or-false assertion that `str2` and `str1` are the same, `str2 == str1`. This assertion is `True` or `False`, and presupposes that `str1` and `str2` are both already defined:

```
*Main> str1 = "Simon"
*Main> str2 = "simon"
*Main> str2 == str1 -- "S" and "s" aren't the same!
False
```

Querying types

You can also ask `ghci` to tell you an expression's type:

```
*Main> :t (:)
(:) :: a -> [a] -> [a]
```

Given any `a` and any list of `a`s, `(:)` produces another list of `a`s:

```
*Main> 'S' : "imon"
"Simon"
```

The `(++)` function is similar but glues together two lists:

```
*Main> :t (++)
(++) :: [a] -> [a] -> [a]
*Main> "Si" ++ "mon"
"Simon"
```

Interlude: infix operators

In general, any operator that is written as an infix can also be written as a prefixed function, by surrounding it with parens:

```
*Main> (+) 2 ((*) 5 6)
32
*Main> (==) ((*) 5 6) 30
True
```

Though it's much harder to read, this is probably a more "faithful" representation of these functions, as suggested by their types. The infix notation is just a nice bit of syntactic "sugar".

Qualified types

```
*Main> :t (==)
(==) :: Eq a => a -> a -> Bool
```

This says: `(==)` is the sort of thing that takes one `a`, then another `a`, and gives back a `Bool` (`True` or `False`), so long as `a` is a type ghci knows how to check `Eq` for (not always possible!).

```
*Main> :t (+)
(+) :: Num a => a -> a -> a
```

This says: `(+)` is the sort of thing that takes one `a`, then another `a`, and ultimately gives back a third `a`, so long as `a` is a `Num`.[1]

[1] Computer languages represent different kinds of numbers in different ways. The details of this aren't important for this course.

Expressions and evaluation

Evaluation of expressions

Expressions are any program or piece of code. Expressions are set in a `grey box` to distinguish from normal text. Some examples:

- `4`, `"hello"`, `x + 3`, ...

We will write $e \Longrightarrow e'$ to mean that expression e **evaluates *in one step*** to expression e'. Some examples:

- `3 + 4` $\Longrightarrow$ `7`
- `2 * (3 + 4)` $\Longrightarrow$ `2 * 7`
- `("un" ++ "lock") ++ "able"` $\Longrightarrow$ `"unlock" ++ "able"`

We may sometimes write $e \Longrightarrow^* e'$ to mean that e evaluates to e' in one or more steps. Thus, `2 * (3 + 4)` $\Longrightarrow^*$ `14`.

Expressions in Haskell

Haskell has a few important types of expressions:

- Identifiers
  - Variables start with a lowercase letter: `x`, `x1`, `myThing`, `add`
  - Data constructors start with an uppercase letter: `True`, `Cons`
- Several related ways of doing evaluation/substitution:
  - Let expressions: `let x = y in z`
  - Lambda expressions: `\x -> body`
  - Case expressions: `case x of {...}`

`let` expressions

`let` expressions allow us to associate a name with some value and use it elsewhere, maybe more than once:

- `let x = 5 in x + (3 * x)` $\Longrightarrow$ `5 + (3 * 5)`

Here's a general rule characterizing evaluation of `let` expressions:

`let v = e in e'` $\Longrightarrow$ [`e` / `v`] `e'`

[`e` / `v`] `e'` is an expression like `e'`, but with all (free) occurrences of `v` replaced by `e`. Pronounced: `e` replacing `v` in `e'`.

Lambda expressions

Another sort of expression, closely related to `let` expressions, are lambda expressions, which are used to define **functions**:

- `\x -> x + (3 * x)`, `\f84 -> f84 'S'`, ...

Lambda expressions have a familiar evaluation behavior:

- `(\x -> x + (3 * x)) 5` $\Longrightarrow$ `5 + (3 * 5)`

This suggests the following familiar evaluation rule:[2]

`(\v -> e') e` $\Longrightarrow$ [`e` / `v`] `e'`

We will often call `e` the **argument** of the function.

[2] As before, the replacement is restricted to **free** occurrences of `v` in `e'`. Thus, we see that `let x = 5 in x * ((\x -> x + 4) 3)` $\Longrightarrow$ `5 * ((\x -> x + 4) 3)`: the innermost `x` is intended to be associated with `\x`, not `let x`.

Matters of taste

You can replace a `let` with a lambda:

- `let v = e in e'` ≡ `(\v -> e') e`
- Both of these evaluate to [`e` / `v`] `e'`

When you define a function, you can use an explicit lambda or not. The latter style is more concise and idiomatic.[3]

```
addTwoLam = \n -> n + 2
addTwo n = n + 2
```

[3] For those who have studied lambda calculus before, this is just η-equivalence.

Pattern matching

When some value has a known shape, it is convenient to expoit that:

```haskell
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x) -- equivalent to
                     -- swap p = (snd p, fst p)
-- *Main> swap (True, 3.14)
-- (3.14, True)

-- These both work the same:
swap' pair = let (x, y) = pair in (y, x)
swap'' = \(x, y) -> (y, x)
```

`case` expressions

A `case` expression allows a function to behave in different ways on different inputs: (It's important the possible values are left-aligned!)

```
is3or4 x = case x of 3 -> True
                     4 -> True
                     _ -> False
```

We can also define a new type and write `case` expressions for it:

```
data Hand = Rock | Paper | Scissors
  deriving Eq

playRock p2 = case p2 of Scissors -> "Rock wins!"
                         _        -> "Rock doesn't win!"
```

When defining a function, a `case` expression can be left out:

```
firstOdd :: [Int] -> Bool
firstOdd []     = False
firstOdd (x:xs) = odd x
-- *Main> firstOdd [5..100]
-- True

-- an equivalent definition:
firstOdd' xs = case xs of []     -> False
                          (x:xs) -> odd x
```

Note that we are also pattern-matching on the `head` of the list `x` in the second `case`-branch. We can also pattern-match on the tail `xs`.

Recursion and inductive types

Numbers

```haskell
-- A Natural number is either Zero,
-- or the Successor of a Natural number

data Nat = Z | S Nat
  deriving (Eq, Show) -- don't worry about this

-- With this data type, we can roll our own numbers:
one   = S Z
two   = S one -- S (S Z)
three = S two -- S (S (S Z))

-- We can also perform sanity checks:
-- *Main> two == S (S Z)
-- True
```

```haskell
addOne :: Nat -> Nat
addOne n = S n

subOne :: Nat -> Nat
subOne Z     = Z
subOne (S n) = n
```

We didn't *have* to decide that `subOne Z` $\Longrightarrow$ `Z`. If we left off this line, trying to evaluate `subOne Z` would lead to an error.

Our first recursive function turns a `Nat` into a regular `Int`:

```haskell
toInt :: Nat -> Int
toInt Z     = 0           -- Rule TI0
toInt (S n) = 1 + toInt n -- Rule TIR
--                ^^^^^ recursion!
```

`toInt (S (S Z))`

Our first recursive function turns a `Nat` into a regular `Int`:

```
toInt :: Nat -> Int
toInt Z     = 0            -- Rule TI0
toInt (S n) = 1 + toInt n -- Rule TIR
--                ^^^^^ recursion!
```

```
toInt (S (S Z))
⟹ 1 + toInt (S Z)        by Rule TIR
```

Our first recursive function turns a `Nat` into a regular `Int`:

```
toInt :: Nat -> Int
toInt Z     = 0           -- Rule TI0
toInt (S n) = 1 + toInt n -- Rule TIR
--                ^^^^^ recursion!
```

```
toInt (S (S Z))
⟹ 1 + toInt (S Z)        by Rule TIR
⟹ 1 + (1 + toInt Z)      by Rule TIR
```

Our first recursive function turns a `Nat` into a regular `Int`:

```
toInt :: Nat -> Int
toInt Z     = 0            -- Rule TI0
toInt (S n) = 1 + toInt n -- Rule TIR
--                ^^^^^ recursion!
```

```
toInt (S (S Z))
⟹ 1 + toInt (S Z)          by Rule TIR
⟹ 1 + (1 + toInt Z)        by Rule TIR
⟹ 1 + (1 + 0)              by Rule TI0
```

Our first recursive function turns a `Nat` into a regular `Int`:

```
toInt :: Nat -> Int
toInt Z     = 0            -- Rule TI0
toInt (S n) = 1 + toInt n -- Rule TIR
--                ^^^^^ recursion!
```

```
toInt (S (S Z))
⟹ 1 + toInt (S Z)        by Rule TIR
⟹ 1 + (1 + toInt Z)      by Rule TIR
⟹ 1 + (1 + 0)            by Rule TI0
⟹ 1 + 1
```

Our first recursive function turns a `Nat` into a regular `Int`:

```
toInt :: Nat -> Int
toInt Z     = 0            -- Rule TI0
toInt (S n) = 1 + toInt n -- Rule TIR
--                ^^^^^ recursion!
```

```
toInt (S (S Z))
⟹ 1 + toInt (S Z)          by Rule TIR
⟹ 1 + (1 + toInt Z)        by Rule TIR
⟹ 1 + (1 + 0)              by Rule TI0
⟹ 1 + 1
⟹ 2
```

On recursion

To write a recursive function, ask yourself these questions, in order:

1. What is your **base case**? Where does the recursion bottom out?
2. Assume that you know how do a calculation for some 'step' of the problem. Given this assumption, what do you require to complete the **next step**?

```
toInt :: Nat -> Int
toInt Z     = 0           -- base case
toInt (S n) = 1 + toInt n
-- assume you can compute `toInt n`. how can
-- you use this to compute toInt (S n)?
```

If your recursion never bottoms out, your program will loop. Don't try this at home: `let badRec n = badRec n in badRec (S Z)`.

Write a `double` function

Write a `double` function

```
double :: Nat -> Nat
double Z     = Z                 -- call this Rule 0
double (S n) = S (S (double n)) -- call this Rule R
--                  ^^^^^^ recursion!
```

`double (S (S Z))`

Write a `double` function

```haskell
double :: Nat -> Nat
double Z     = Z                -- call this Rule 0
double (S n) = S (S (double n)) -- call this Rule R
--                  ^^^^^^ recursion!
```

`double (S (S Z))`

$\Longrightarrow$ `S (S (double (S Z)))` by Rule R

Write a `double` function

```haskell
double :: Nat -> Nat
double Z     = Z                -- call this Rule 0
double (S n) = S (S (double n)) -- call this Rule R
--                  ^^^^^^ recursion!
```

`double (S (S Z))`

$\Longrightarrow$ `S (S (double (S Z)))` by Rule R

$\Longrightarrow$ `S (S (S (S (double Z))))` by Rule R

Write a `double` function

```haskell
double :: Nat -> Nat
double Z     = Z                   -- call this Rule 0
double (S n) = S (S (double n)) -- call this Rule R
--                 ^^^^^^ recursion!
```

	`double (S (S Z))`	
$\Longrightarrow$	`S (S (double (S Z)))`	by Rule R
$\Longrightarrow$	`S (S (S (S (double Z))))`	by Rule R
$\Longrightarrow$	`S (S (S (S Z)))`	by Rule 0

Write `isEven` and `isOdd`

Write `isEven` and `isOdd`

```
isEven :: Nat -> Bool
isEven Z     = True     -- Rule E0
isEven (S n) = isOdd n  -- Rule ER
--             ^^^^^ mutual

isOdd :: Nat -> Bool
isOdd Z     = False     -- Rule O0
isOdd (S n) = isEven n  -- Rule OR
--            ^^^^^^ recursion
```

`isEven (S (S Z))`

Write `isEven` and `isOdd`

```
isEven :: Nat -> Bool
isEven Z     = True      -- Rule E0
isEven (S n) = isOdd n   -- Rule ER
--             ^^^^^ mutual

isOdd :: Nat -> Bool
isOdd Z     = False      -- Rule O0
isOdd (S n) = isEven n   -- Rule OR
--            ^^^^^^ recursion
```

`isEven (S (S Z))`
$\Longrightarrow$ `isOdd (S Z)` by Rule ER

Write `isEven` and `isOdd`

```
isEven :: Nat -> Bool
isEven Z     = True      -- Rule E0
isEven (S n) = isOdd n   -- Rule ER
--             ^^^^^ mutual

isOdd :: Nat -> Bool
isOdd Z     = False      -- Rule O0
isOdd (S n) = isEven n   -- Rule OR
--            ^^^^^^ recursion
```

`isEven (S (S Z))`

$\Longrightarrow$ `isOdd (S Z)` by Rule ER

$\Longrightarrow$ `isEven Z` by Rule OR

Write `isEven` and `isOdd`

```
isEven :: Nat -> Bool
isEven Z     = True     -- Rule E0
isEven (S n) = isOdd n  -- Rule ER
--             ^^^^^ mutual

isOdd :: Nat -> Bool
isOdd Z     = False     -- Rule O0
isOdd (S n) = isEven n  -- Rule OR
--            ^^^^^^ recursion
```

`isEven (S (S Z))`
$\Longrightarrow$ `isOdd (S Z)` by Rule ER
$\Longrightarrow$ `isEven Z` by Rule OR
$\Longrightarrow$ `True` by Rule E0

Constructing lists

Here is another way to roll by hand a type that Haskell gives us:

```
data IntList = Empty | Cons Int IntList
--                     ^^^^ 'Cons' for 'Construct'

myIntList = Cons 2 (Cons 5 (Cons 3 Empty))

sumIntList :: IntList -> Int
sumIntList Empty       = 0
sumIntList (Cons n ns) = n + sumIntList ns

-- *Main> sumIntList myIntList
-- 10
```

Haskell's native lists

In fact, this is how Haskell's lists already work!

```
*Main> 2 : (5 : (3 : [])) == [2,5,3]
True
```

And recalling that `x : xs` is just infix sugar for `(:) x xs`...

```
*Main> [2,5,3] == (:) 2 ((:) 5 ((:) 3 []))
True
```

A `Cons 2 (Cons 5 (Cons 3 Empty))` by any other name...

Recursive functions on lists

Since Haskell lists are iteratively constructed, they can be iteratively *destructed* by recursive functions, just like `Nat` s.

```
sumList []     = 0
sumList (n:ns) = n + sumList ns
```

`sumlist [2,5,3] -- 2:(5:(3:[]))`

Recursive functions on lists

Since Haskell lists are iteratively constructed, they can be iteratively *destructed* by recursive functions, just like `Nat` s.

```
sumList []     = 0
sumList (n:ns) = n + sumList ns
```

`sumlist [2,5,3] -- 2:(5:(3:[]))`

$\Longrightarrow$ `2 + sumlist [5,3]`

Recursive functions on lists

Since Haskell lists are iteratively constructed, they can be iteratively *destructed* by recursive functions, just like `Nat` s.

```
sumList []     = 0
sumList (n:ns) = n + sumList ns
```

```
sumlist [2,5,3] -- 2:(5:(3:[]))
⟹ 2 + sumlist [5,3]
⟹ 2 + (5 + sumList [3])
```

Recursive functions on lists

Since Haskell lists are iteratively constructed, they can be iteratively *destructed* by recursive functions, just like `Nat` s.

```
sumList []     = 0
sumList (n:ns) = n + sumList ns
```

```
sumlist [2,5,3] -- 2:(5:(3:[]))
⟹ 2 + sumlist [5,3]
⟹ 2 + (5 + sumList [3])
⟹ 2 + (5 + (3 + sumList []))
```

Recursive functions on lists

Since Haskell lists are iteratively constructed, they can be iteratively *destructed* by recursive functions, just like `Nat`s.

```
sumList []     = 0
sumList (n:ns) = n + sumList ns
```

```
sumlist [2,5,3] -- 2:(5:(3:[]))
⟹ 2 + sumlist [5,3]
⟹ 2 + (5 + sumList [3])
⟹ 2 + (5 + (3 + sumList []))
⟹ 2 + (5 + (3 + 0))
```

Recursive functions on lists

Since Haskell lists are iteratively constructed, they can be iteratively *destructed* by recursive functions, just like `Nat` s.

```
sumList []     = 0
sumList (n:ns) = n + sumList ns
```

```
sumlist [2,5,3] -- 2:(5:(3:[]))
⟹ 2 + sumlist [5,3]
⟹ 2 + (5 + sumList [3])
⟹ 2 + (5 + (3 + sumList []))
⟹ 2 + (5 + (3 + 0))
⟹ 2 + (5 + 3)
```

Recursive functions on lists

Since Haskell lists are iteratively constructed, they can be iteratively *destructed* by recursive functions, just like `Nat`s.

```
sumList []     = 0
sumList (n:ns) = n + sumList ns
```

```
sumlist [2,5,3] -- 2:(5:(3:[]))
⟹ 2 + sumlist [5,3]
⟹ 2 + (5 + sumList [3])
⟹ 2 + (5 + (3 + sumList []))
⟹ 2 + (5 + (3 + 0))
⟹ 2 + (5 + 3)
⟹ 2 + 8
```

Recursive functions on lists

Since Haskell lists are iteratively constructed, they can be iteratively *destructed* by recursive functions, just like `Nat` s.

```
sumList []     = 0
sumList (n:ns) = n + sumList ns
```

```
sumlist [2,5,3] -- 2:(5:(3:[]))
⟹ 2 + sumlist [5,3]
⟹ 2 + (5 + sumList [3])
⟹ 2 + (5 + (3 + sumList []))
⟹ 2 + (5 + (3 + 0))
⟹ 2 + (5 + 3)
⟹ 2 + 8
⟹ 10
```

Math and logic

The `sumList` function may be familiar from math classes:

- `sumList` is the big summation operation $\sum$
- `prodList`, defined with `*` replacing `+`, is the big product $\prod$

As it happens, `Prelude` already defines `sum` and `product` for you.

The `Prelude` also defines `or` and `and`. What do they do?

```
*Main> :t or
or :: [Bool] -> Bool -- same type for `and`
```

Math and logic

The `sumList` function may be familiar from math classes:

- `sumList` is the big summation operation $\sum$
- `prodList`, defined with `*` replacing `+`, is the big product $\prod$

As it happens, `Prelude` already defines `sum` and `product` for you. The `Prelude` also defines `or` and `and`. What do they do?

```
*Main> :t or
or :: [Bool] -> Bool -- same type for `and`
```

They operate on lists of `Bool`s: `or` requires that *at least one* of them is `True`, and `and` requires that *all* of them are `True`.

- `or` is the big disjunction $\bigvee$
- `and` is the big conjunction $\bigwedge$

List comprehensions

Lists are in some ways central to Haskell, and it makes it very easy to do amazing stuff with them, like 'graph' functions:

```
x3 = [ (x,y) | x <- [1..10], y <- [1..1000], y == x^3 ]
-- *Main> x3
-- [(1,1),(2,8),(3,27),(4,64),(5,125),(6,216),(7,343),
--  (8,512),(9,729),(10,1000)]
```

The synax of a list comprehension is: `[ vals | conditions ]`. This is intended to be just like set comprehensions:

$$\{(x, y) \mid x \in \{1 \ldots 10\}, y \in \{1 \ldots 1000\}, y = x^3\}$$

Ranges

Ranges are easy to pick out with lists:

- `[1..10]` $\Longrightarrow^*$ `[1,2,3,4,5,6,7,8,9,10]`
- `['d'..'m']` $\Longrightarrow^*$ `"defghijklm"`

`[1..]` is an *infinite list*, which you can actually use!

```
*Main> take 10 (drop 36 [1..])
[37,38,39,40,41,42,43,44,45,46]
```

What do you suppose the `length` of `myListEvens` is?

```
myListEvens = [n | n <- [1..100], even n]
```

How would you define `myListOdds`?

What do you suppose the `length` of `myListEvens` is?

```
myListEvens = [n | n <- [1..100], even n]
```

How would you define `myListOdds`?

```
myListOdds = [n | n <- [1..100], odd n]
```

What's `sumList myListEvens - sumList myListOdds`?

What do you suppose the `length` of `myListEvens` is?

```
myListEvens = [n | n <- [1..100], even n]
```

How would you define `myListOdds`?

```
myListOdds = [n | n <- [1..100], odd n]
```

What's `sumList myListEvens - sumList myListOdds`?

```
*Main> sumList myListEvens - sumList myListOdds
50
```

List comprehensions

```
muls17 = [ n | n <- [1..], n `mod` 17 == 0 ]
```

```
*Main> take 15 muls17
[17,34,51,68,85,102,119,136,153,170,187,204,221,238,255]
```

```
pyTriples = [ (a,b,c) | c <- [1..], b <- [1..c],
                        a <- [1..b], a*a + b*b == c*c ]
```

```
*Main> take 4 (drop 123 pyTriples)
[(48,189,195),(28,195,197),(120,160,200),(56,192,200)]
```

On `pyTriples`: we know the hypotenuse `c` is longer than both legs `a` and `b`, and we don't care about returning both `(3,4,5)` and `(4,3,5)`. Much more efficient!

Strings

A `String` is just a nice sugared representation for a list or sequence of `Char` s, as we discussed earlier.

```
*Main> 'S':('i':('m':('o':('n':[])))) == "Simon"
True
```

Lists and `String` s can both be concatenated via `++`:

```
*Main> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
*Main> "Si" ++ "mon"
"Simon"
```

How would you define `++` on your own, if you had to?