Tuesday – August 6, 2024

Lecture 1: Haskell expressions and recursion

1 Language (or more broadly, cognition) is computation

What kinds of machines are human minds, such that they can acquire, represent, and use language in the way that they do?

It can be helpful to think about the different levels of computation described by Marr (1982)[1]

- *What* is the machine computing?
- *Why* is the machine performing this particular kind of computation?
- These two questions can help us develop a **computational-level** understanding of the system. This can help inform our understanding of the system at other, less abstract levels: *how* the machine is performing this particular kind of computation via particular algorithms (the **algorithmic level**), implemented in particular hardware (the **implementational level**).

In this course we will examine natural language grammars primarily at Marr's computational level. We will do so by asking: how can we characterize the properties of the abstract computations that mental grammars must perform in order to recognize or generate linguistic expressions?

- This is a separate question from the question of what specific algorithms our minds use to carry out these computations, or how those algorithms are implemented in our brains.

- Quiz from tuesdays lecture content

[1] Marr, D. (1982). Vision: A computational investigation into the human representation and processing of visual information. San Francisco, CA: W.H. Freeman.

2 Expressions in Haskell

2.1 Basics

· variables are immutable

1. An **expression** (or **term**) is a piece of code, or a program. Some examples: `3`, `3 + 4`, `"hello"`, `x * 3`. There are two questions that we could ask about an expression:

   - How are complex expressions *built up* out of smaller expressions?
   - What does an expression *evaluate* to? (The answer will be another expression.)

   We will write $e \Longrightarrow e'$ to say that an expression e evaluates *in only one step* to the expression e':

   (1) a. `3 + 4` $\Longrightarrow$ `7`
   b. `2 * (3 + 4)` $\Longrightarrow$ `2 * 7`
   c. `"un" ++ "lock"` $\Longrightarrow$ `"unlock"`

   We will write $e \Longrightarrow^* e'$ to say that an expression e evaluates *in zero or more steps*[2] to the expression e':

   (2) a. `3 + 4` $\Longrightarrow^*$ `7` – 1 step
   b. `2 * (3 + 4)` $\Longrightarrow^*$ `2 * (3 + 4)` – 0
   c. `2 * (3 + 4)` $\Longrightarrow^*$ `2 * 7` – 1
   d. `2 * (3 + 4)` $\Longrightarrow^*$ `14` – 2
   e. `("un" ++ "lock") ++ "able"` $\Longrightarrow^*$ `"unlockable"`

2. The **type** of an expression determines how it can combine with other expressions, and how it gets evaluated. For example, we can think of Booleans as defined along these lines:

   type name

   (3) `data Bool = True | False`

   This means that `True` and `False` are expressions of type `Bool`.

   The basic functions for working with Booleans are conjunction (`&&`), disjunction (`||`), and negation (`not`). These functions map expressions of type `Bool` to other expressions of type `Bool`. Their evaluation rules work as you'd expect:

   depends on it

   (4) a. `True &&` $e \Longrightarrow e$
   `False &&` $e \Longrightarrow$ `False`
   b. `True ||` $e \Longrightarrow$ `True` depends on it
   `False ||` $e \Longrightarrow e$
   c. `not True` $\Longrightarrow$ `False`
   `not False` $\Longrightarrow$ `True`

":t n" ← tells us type of n which is integer

[2] This relation is defined as the reflexive, transitive closure of $\Longrightarrow$, i.e. (i) $e \Longrightarrow^* e$ and (ii) $e \Longrightarrow^* e'$ if there is an e'' such that $e \Longrightarrow e''$ and $e'' \Longrightarrow^* e'$.

2.2 Working with this system in ghci

The evaluation relation $\Longrightarrow^*$ corresponds to what `ghci`, the Haskell interpreter, does to an expression that you type in. Here are some examples of how this works.

```
$ ghci
GHCi, version 8.10.4: https://www.haskell.org/ghc/  :? for help
Prelude> 2 * (3 + 4)
14
Prelude> ("un" ++ "lock") ++ "able"
"unlockable"
Prelude> (not False) || (True && False)
True
Prelude> :q
Leaving GHCi.
$
```

We can also give names to expressions by writing code in a file. For example, we could add this to `Variables.hs`:

```
module Variables where

    n = 4
    k = 8
```

And now we can use these definitions in `ghci` like this:

```
$ ghci Variables.hs
GHCi, version 8.10.4: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Variables        ( Variables.hs, interpreted )
Ok, one module loaded.
*Variables> 5 * n
20
*Variables> k - n
4
*Variables> (k > n) || False
True
```

In addition to asking `ghci` to evaluate an expression, we can also use it to check the type of an expression, using the `:t` command:[3]

```
*Variables> :t "hello"
"hello" :: [Char]
*Variables> :t k > n
k > n :: Bool
```

[3]Unfortunately you will sometimes see some weird stuff like `Num a => a` where you expected to see a simple integer type, like `Int`. This is because of some fancy footwork that is going on under the hood to allow you to mix integer and non-integer values, for example when you write `2 + 3.5`. We can ignore this.

all variables defined

variables with no variables

2.3 `let` expressions

`let` expressions are one of several types of expressions that allow us to build closed expressions out of open expressions, by binding the free occurrences of variables. We can use `let` expressions to give a name to the result of some computation, so that this result can be used elsewhere (perhaps multiple times).

For example:

(5) `let x = 3 in (x + 4) * x` $\Longrightarrow$ `(3 + 4) * 3`

This is also essentially what happens when we write `x = 3` in a file, and then evaluate `(x + 4) * x`.

Here are the basic ideas for building and evaluating `let` expressions:

(6) a. If e_1 and e_2 are both expressions, and v is a variable, then `let` $v = e_1$ `in` e_2 is an expression.

b. `let` $v = e_1$ `in` e_2 $\Longrightarrow [e_1/v]$ e_2

~ Syntax

Here, $[e_1/v]$ e_2 means the expression just like e_2, but with all free occurrences of the variable v replaced with e_1 (a **substitution**). We'll come back to a more careful definition of how this replacement works, but the general pattern looks like this:

`let` $v = e_1$ `in` $\ldots v \ldots v \ldots$ $\Longrightarrow$ $\ldots e_1 \ldots e_1 \ldots$

So, we can write out what's going on in example (5) above a bit more explicitly:

same thing

(7) `let x = 3 in (x + 4) * x` $\Longrightarrow [3/x]$ `(x + 4) * x` $=$ `(3 + 4) * 3`

Notice that the rule for building a `let` expression only tells you that e_1 and e_2 need to be expressions, but doesn't put any other restrictions on what they are. So there's no reason we can't have `let` expressions like these:

(8) `let x = 3 in (5 + 4)`

(9) `let x = 3 in (let y = 2 * x in (y + 4))` = 10

7

(10) `let z = (let x = 3 in (x + 4)) in (z * 2)` = 14

There's nothing special going on here: to evaluate these expressions, just follow our normal rules of evaluation, working from the inside out.

2.4 Lambda expressions

Another way to build a closed expression out of an open expression is to use the open expression as the body of a **lambda** expression. This is the same lambda (λ) that you may have seen before in semantics to introduce a function. Haskell uses a backslash (\) as its symbol for λ.

Here are definitions for building and evaluating lambda expressions:

(11) a. **Lambda expressions**: If e is an expression and v is a variable, then $\backslash v -> e$ is an expression.

b. **Function application**: If e_1 and e_2 are both expressions, then $e_1\ e_2$ and $e_1\ \$\ e_2$ are also expressions.[4]

Apply e_1 to e_2

Against this backdrop, the evaluation "recipe" for lambda expressions can be stated like this:

- substitute e_2 into v for e

(12) a. $(\backslash v -> e)\ e_2 \Longrightarrow [e_2/v]\ e$

b. $(\backslash v -> e)\ \$\ e_2 \Longrightarrow [e_2/v]\ e$

take in x and input is 3

For example:

(13) a. `(\x -> (x + 4) * x) 3` $\Longrightarrow [3/x]$ `(x + 4) * x` = `(3 + 4) * 3`

b. `(\x -> (x + 4) * x) $ 3` $\Longrightarrow [3/x]$ `(x + 4) * x` = `(3 + 4) * 3`

This evaluation looks essentially the same as the evaluation of the `let` expression `let x = 3 in (x + 4) * x`. In both cases, we're immediately providing a value for the variable x in an open expression that contains x. But unlike `let`, a lambda expression gives us the option to postpone providing that value until some later time.

It can be enlightening to think about the following comparison:

open
- `3 + x` is an expression that is in an important sense incomplete. It's an expression that will be of type `Int`, if it's given a value for x of the appropriate type (e.g., another `Int`). takes int and outputs int

closed
- `\x -> 3 + x` is a stand-alone expression whose type we can specify as (`Int` → `Int`). It can stand on its own (it is closed) in the same way that `let x = 5 in (3 + x)` can, but `3 + x` cannot. The variable x is *bound*, even though no value has been provided for it.

[4]Note that function application with a space is left-associative, and function application with a `$` is right-associative: `f a b` = `(f a) b`, but `f $ a b` = `f (a b)`.

use this

2.5 `case` expressions

2.5.1 Simple versions

We've seen expressions of type `Int`, such as `3` and `4 * 5`, expressions of type `String`, such as `"hello"`, and expressions of type `Bool`, such as `True` and `False`. But we can also define new types of our own. Let's define a new type called `Shape`, like this:

```
data Shape = Rock | Paper | Scissors deriving Show
```

This definition has the consequence that `Rock`, `Paper`, and `Scissors` are all now expressions of the type `Shape`. We have to include `deriving Show` in order to get `ghci` to print expressions of this type to the screen. For now, just treat this as boilerplate.

Every type has a corresponding `case` expressions. These are fundamental components of everything that we'll be doing with this system. Here is how to build a `case` expression for our new type `Shape`:

(14) If e is an expression of type `Shape`, and e_1, e_2, and e_3 are all expressions of the same type, then `case` e `of {Rock ->` e_1`; Paper ->` e_2`; Scissors ->` e_3`}` is an expression.

Here are the evaluation rules for these `case` expressions:

:r reload code

(15) a. `case Rock of {Rock ->` e_1`; Paper ->` e_2`; Scissors ->` e_3`}`
$\Longrightarrow$ e_1

b. `case Paper of {Rock ->` e_1`; Paper ->` e_2`; Scissors ->` e_3`}`
$\Longrightarrow$ e_2

c. `case Scissors of {Rock ->` e_1`; Paper ->` e_2`; Scissors ->` e_3`}`
$\Longrightarrow$ e_3

For example:

(16) `case Paper of {Rock -> 0; Paper -> 5; Scissors -> 2}` $\Longrightarrow$ `5`

This isn't very interesting by itself: we're just "matching" the `Shape`-type expression specified by the `case` expression, and returning another expression when we find a match. But this becomes more interesting when the `Shape`-type expression being matched is the result of other steps of evaluation.

To illustrate with some more meaningful examples:

(17) a. `let myShape = Paper in`
`(case myShape of {Rock -> 0; Paper -> 5; Scissors -> 2})`
myShape
$\Longrightarrow$ `case Paper of {Rock -> 0; Paper -> 5; Scissors -> 2}`
$\Longrightarrow$ `5`

b. `f = \x -> case x of {Rock -> 0; Paper -> 5; Scissors -> 2}`
`f Rock`
input

$\Longrightarrow$ `case Rock of {Rock -> 0; Paper -> 5; Scissors -> 2}`
$\Longrightarrow$ `0`

2.5.2 More interesting versions, with variables

The `case` expressions for more interesting types (what we might call "compound types") involve a third instance of variable substitution, in addition to `let` expressions and lambda expressions.

To see how this works, let's define a new type, `Result`, like this:

```
data Result = Draw | Win Shape deriving Show
```

Whereas `Shape` is a type with three "options", `Result` is a type with two "options". But one of those options (`Win`) comes with some extra information: a `Shape`. In other words, `Result` has two values: `Draw` and `Win` e, where e is an expression of type `Shape`.

This definition of the `Result` type has the consequence that:

(18) If e is an expression of type `Result`, and e_1 and e_2 are both expressions of the same type, and v is a variable, then `case` e `of {Draw ->` e_1`; Win` v `->` e_2`}` is an expression.

Here are the evaluation rules for `case` expressions with type `Result`, which involve variable substitution:

(19) a. `case Draw of {Draw ->` e_1`; Win` v `->` e_2`}` $\Longrightarrow$ e_1
b. `case (Win` e`) of {Draw ->` e_1`; Win` v `->` e_2`}` $\Longrightarrow$ $[e/v]$ e_2 v substituted for e

If we imagine that we have an appropriate `toString` function, then the following example illustrates how we could use variable substitution:

Rock won

(20) `case (Win Rock) of {Draw -> "No winner"; Win x -> "Congrats to " ++ toString x}`
$\Longrightarrow$ [Rock/x] `"Congrats to " ++ toString x` $=$ `"Congrats to " ++ toString Rock` $\Longrightarrow^*$
`"Congrats to " ++ "Rock"`
$\Longrightarrow$ `"Congrats to Rock"`

sub rock for x

2.6 Free and bound variables, and consequences for substitution

There are a few tricky details to watch out for when you're substituting/replacing a variable with a term in some larger term. These details concern the fiddly notion of *free* occurrences of variables.

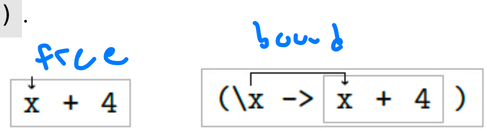
Intuitively, we should expect that the expressions `3 + 4` and `(\x -> x + 4) 3` will "behave alike" in all contexts. This means that the following expressions should behave alike as well:

(21) a. `let x = 5 in x * (3 + 4)` 35
b. `let x = 5 in x * ((\x -> x + 4) 3))`

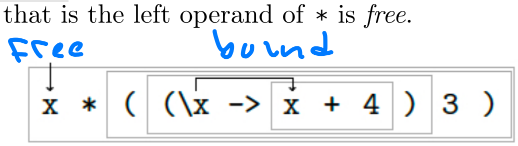
In particular, (21-b) should *not* evaluate to `5 * ((\x -> 5 + 4) 3))`! Intuitively, this is because the occurrence of `x` that is the left operand of `+` in (21-b) is none of the `let` expression's "business." Instead, it is associated with the lambda expression. But the occurrence of `x` that is the left operand of `*` *is* the `let` expression's "business."

We can understand this in terms of operator scope and variable binding. If a variable is **bound** by an operator, then that variable is in the **scope** of that operator. If we take a look at how (21-b) is built up, we can see how these relationships work:

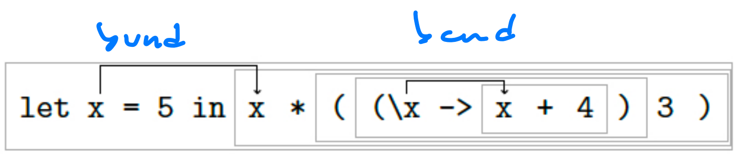
- The occurrence of `x` that is the left operand of `+` is *free* in the expression `x + 4`, but is *bound* in the expression `(\x -> x + 4)`.



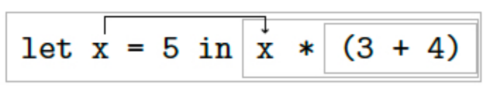
- In `x * ((\x -> x + 4) 3)`, the occurrence of `x` that is the left operand of `+` is *bound* (by the `\x`), and the occurrence of `x` that is the left operand of `*` is *free*.



- In the full expression (21-b), the occurrence of `x` that is the left operand of `+` is (still) *bound* (by the `\x`), and the occurrence of `x` that is the left operand of `*` is now also *bound* (in this case, by the `let x`).



- So `3 + 4` and `(\x -> x + 4) 3` behave alike in the two expressions in (1) because they are both *closed expressions*. The surrounding `let` does not "get inside" either of them.



The general rules for variable binding, in the expressions we've seen so far, are:

(22) a. All free occurrences of v in e are bound by the `let` in `let` $v = e'$ `in` e.[5]
b. All free occurrences of v in e are bound by the lambda in `\`v `->` e.
c. All free occurrences of v in e are bound by the `case` in `case` e' `of {...; Win` v `->` e`; ...}`.

So to ensure that things stay in sync with our intuitive expectations about what should "behave alike," we must take $[e/v]e'$ to mean the result of substituting e only for the *free* occurrences of v in e'.[6]

[5] Actually, in Haskell, free occurrences of v are also bound in e'. This is what allows recursive definitions. But we'll put this aside for the moment.

[6] There's one more catch: $[e/v]e'$ is undefined if there are binders in e' for some variable that occurs free in e. This will rarely come up in practice, however.

(23)
a. [5/x] `x * (3 + 4)` = `5 * (3 + 4)`
b. [5/x] `x * ((\x -> x + 4) 3)` = `5 * ((\x -> x + 4) 3)`
c. [5/x] `x * ((\x -> x + 4) 3)` ≠ `5 * ((\x -> 5 + 4) 3)`

not free

3 Recursive types and recursive expressions

3.1 Propositional formulas

You're probably familiar with recursive definitions of the following sort from semantics and logic textbooks.

, true and false values

(24) The set $\mathcal{F}$ of propositional formulas is defined as the smallest set such that:

a. $\mathbf{T} \in \mathcal{F}$
b. $\mathbf{F} \in \mathcal{F}$
c. if $\phi \in \mathcal{F}$, then $\neg\phi \in \mathcal{F}$
d. if $\phi \in \mathcal{F}$ and $\psi \in \mathcal{F}$, then $(\phi \overset{\text{and}}{\wedge} \psi) \in \mathcal{F}$
e. if $\phi \in \mathcal{F}$ and $\psi \in \mathcal{F}$, then $(\phi \underset{\text{or}}{\vee} \psi) \in \mathcal{F}$

The standard denotations of these formulas are as follows:

(25) a. $[\![\mathbf{T}]\!]$ is true
b. $[\![\mathbf{F}]\!]$ is false
c. $[\![\neg\phi]\!]$ is true if $[\![\phi]\!]$ is false; and is false otherwise
d. $[\![\phi \wedge \psi]\!]$ is true if both $[\![\phi]\!]$ and $[\![\psi]\!]$ are true; otherwise, $[\![\phi \wedge \psi]\!]$ is false
e. $[\![\phi \vee \psi]\!]$ is true if either $[\![\phi]\!]$ or $[\![\psi]\!]$ is true; otherwise, $[\![\phi \vee \psi]\!]$ is false

3.2 Recursive types

We can define a Haskell type to represent these formulas in a way that very closely matches this definition:

, type , name , conjunction , disjunction

(26)
```
data Form = T | F | Neg Form | Cnj Form Form | Dsj Form Form deriving Show
```
form ∧ form form ∨ form

The five **constructors** here (`T, F, Neg, Cnj` and `Dsj`) correspond to the five "ways to be a formula" given in (24). Whereas `Bool` is a type with two "options", `Form` is a type with five "options". And furthermore, three of those options, namely `Neg`, `Cnj`, and `Dsj`, come with some extra information, namely another `Form`. So this type has recursive structure: `Form` is being used inside its own definition, on both the left-hand and right-hand sides of the equals sign.

This definition has the consequence that:

(27) a. `T` and `F` are expressions of type `Form`.
b. If e is an expression of type `Form`, then `Neg` e is an expression of type `Form`.
c. If e_1 and e_2 are expressions of type `Form`, then `Conj` e_1 e_2, and `Dsj` e_1 e_2 are expressions of type `Form`.

∨ ∧

For example, we can use the expression `Dsj (Neg T) (Cnj F T)` to represent the formula $(\neg \mathbf{T} \vee (\mathbf{F} \wedge \mathbf{T}))$.

∨ ¬T F∧T

```
F :: Form    T :: Form          ← base case
      ↓
(Cnj F T) :: Form
      ↓
Dsj (Neg T) (Cnj F T) :: Form
      ↑
(Neg T) :: Form
      ↑
T :: Form          ← base case: a form that does not contain an instance of that type.
```

3.3 Recursive expressions

Function example →

Here's a function that takes an expression of type `Form` as its argument, and returns an expression of type `Bool` that is the denotation of that argument. Its structure follows the standard denotation definitions we saw before:

```
f1 = Dsj (Neg T) (Cnj F T)
```

run this on here

name of function

function

phi ∧ psi

phi ∨ psi

```
denotation = \x -> case x of
                     T -> True
                     F -> False
                     Neg phi -> not (denotation phi)
                     Cnj phi psi -> (denotation phi) && (denotation psi)
                     Dsj phi psi -> (denotation phi) || (denotation psi)
```

input output

(28)
```
denotation (T) = True
denotation (F) = False
denotation (Neg phi) = not (denotation phi)
denotation (Cnj phi psi) = (denotation phi) && (denotation psi)
denotation (Dsj phi psi) = (denotation phi) || (denotation psi)
```

denotation (f1) = False

A few properties of the `denotation` function:

- It gives us a particular mapping from `Form`s to `Bool`s. So in this system, we can write its type as `denotation :: (Form` → `Bool)`.
  input output
- It's recursively-defined, and the recursive parts of this definition relate in a systematic way to the instances of recursion within the `Form` type.
- In particular, whenever we have a `Form` "inside" of the `Form` type, we have a corresponding instance of `denotation` appearing on the right-hand side of the definition.

Let us evaluate `denotation (Dsj (Neg T) (Cnj F T))`

3.4 A recursive type for natural numbers

The `Form` type that we defined above is an example of a recursive type. For a deeper understanding of exactly how these work, let us look at another example, the type `Numb` defined as follows:

zero   S followed by numb

(29)
```
data Numb = Z | S Numb deriving Show
```

imagine haskell didn't have integers data type

```
S (S Z) :: Numb
   ↑
  (S Z) :: Numb
     ↑
     Z :: Numb
```
← *base case*: a form that does not contain an instance of that type.

It's just like the `Result` type, except that the thing "inside" it is another thing of the same type (whereas the thing "inside" a `Result` is a `Shape`). Straight away, we can write some simple functions to work with this type.

lambda

(30)
```
isZero = \n -> case n of {Z -> True; S n' -> False}
isOne = \n ->  case n of
               Z -> False
               S n' -> case n' of {Z -> True; S n" -> False}
```

peel off n   peel off another n

Here's a function `double` that takes a `Numb` and doubles it.

(31)
```
double :: Numb -> Numb
double = \n ->  case n of
                Z -> Z
                S n' -> S (S (double n'))
```

non zero

We can again imagine the following abstract shape of the evaluation for, say, `double (S (S (S Z)))`:

3

(32)
```
double (S (S (S Z)))
* S (S (double (S (S Z))))
* S (S (S (S (double (S Z)))))
* S (S (S (S (S (S (double Z))))))
* S (S (S (S (S (S Z)))))
```

6 S

It can take some time to get the hang of writing recursive functions like these. Here are some ingredients for writing recursive functions/types/expressions

- To avoid getting lost in infinite recursion in the computation, you need identify one/more base case(s) (often 0, 1, Z, (S Z), an empty list, a list with one element, empty string etc.).
- Think about how the problem can be divided into the same problem but with smaller size.
- Think about how the solutions to sub-problems can be combined into a solution to the bigger problem.
  - For example: what's the difference between doubling a number and doubling its predecessor? This tells us how to describe `double S n'` in terms of `double n'`.

```
isOdd :: Num → Bool
isOdd :: \n → case n of
              z → False
              s n' → case (isOdd n') of
                       True → False
                       False → True
```

1 less of n'

More examples

will be posted

- Write a function `isOdd :: Numb -> Bool` which returns `True` if a number is odd, and `False` otherwise.
- Write a function `add :: Numb -> (Numb -> Numb)` which computes the sum of two numbers.

3.5 A recursive type for lists

We can represent lists of, say, integers using a very similar structure to what we used for `Numb`:

name either

(33)
```
data IntList = Empty | NonEmpty Int IntList deriving Show
```

For example, the list containing `5` followed by `7` followed by `2` (and nothing else) would be represented as:

(34)
```
NonEmpty 5 (NonEmpty 7 (NonEmpty 2 Empty))
```

Using this type, we could write a function to calculate the sum of a list of integers:

function name input output

call total on rest

(35)
```
total :: IntList -> Int
total = \l -> case l of
              Empty -> 0
              NonEmpty x rest -> x + total rest
```

Haskell has a built-in type to represent lists, which uses some compact syntax. The compact syntax is convenient, but it can obscure the fact that this built-in type actually has exactly the same kind of recursive structure as this `IntList` type. Using this built-in type, we write the list containing `5` then `7` then `2` as (34) instead of (36); and we write the function for summing a list as in (37) instead of (35).

(36)
```
5: (7: (2: []))
```

(37)
```
total :: [Int] -> Int
total = \l -> case l of
              [] -> 0
              x : rest -> x + total rest
```

The differences are just that:

- the built-in type uses `[]` instead of `Empty`
- the built-in type uses the colon ("cons") instead of `NonEmpty`; and
- the colon is written *between* its two arguments, unlike `NonEmpty` and other constructors we've seen.

And, as an added convenient (but potentially misleading) bonus, we can also write `[5,7,2]` in place of `5 : (7 : (2 : []))`.