Notes for Lecture 9 (Fall 2022 week 5, part 1):
Lists

Jana Dunfield

October 2, 2022

The code for this lecture is in `lec9.hs`.

We will probably not have time to get through lists during today's lecture, but try to read that section and ask questions on Piazza, in email, or in Wednesday's lecture.

1   Lists

We have briefly worked with the type `String`, which is a type synonym for `[Char]`, the type of lists of `Character`s.

More generally, `[a]` is the type of lists of elements of type `a`. For example, a list of Booleans has type `[Bool]`.

```
Prelude> :type [True, False, False]
[True, False, False] :
```

1.1   Constructing lists

Lists are 'data' types—every list is built up from two constructors:

```
[]                      pronounced ''nil'' or ''the empty list''


:                       pronounced ''cons''
```

The first constructor, "nil", is probably what you'd expect.

The second one might be surprising, since we've been using strings a little (for the `toBinary` and `toNary` functions in assignment 1) and haven't seen it. This is because Haskell tries to be friendly by printing lists in a more readable form. The list that's displayed as

```
[True, False, False]
```
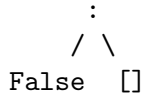
is really the expression

```
True : (False : (False : []))
```

Unlike the constructors we define in our own 'data' types, "cons" is infix, so the `:` goes between its two arguments. Let's unpack the above expression, starting from the right.

```
[]                  is the empty list


(False : [])        is a list whose head is False and whose tail is []
```

We can draw the list (`False : []`) as a parse tree:

```
     :
    / \
 False  []
```
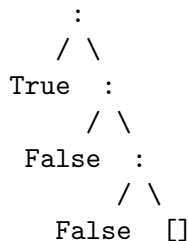
Moving towards the left in `True : (False : (False : []))`, we have

```
(False : (False : []))   is a list whose head is False
                         and whose tail is the list (False : [])
```

Continuing to the left, we have

```
True : (False : (False : []))   is a list whose head is True
                                and whose tail is the list (False : (False : []))
```

Its parse tree (or syntax tree; I use these terms interchangeably) is

```
      :
     / \
  True  :
       / \
   False  :
         / \
      False  []
```

In a sense, every list is a tree—one that leans to the right, because unlike a general tree where the left child could be a large tree, the left child of "cons" (`:`) is always a single element.

If we enter `True : (False : (False : []))` into GHC, we get

```
[True, False, False]
```

because GHC, reasonably, thinks that `[True, False, False]` is more readable than
`True : (False : (False : []))`. I agree with this, but to write Haskell code that deals with lists,
it is often very useful to think of lists as what they really are: syntax trees made up of `:` ("cons")
and `[]` ("nil"). This is because we can pattern-match on lists, similar to how we can pattern-match
on types like `ArithExpr` (lec8.hs).

1.2 Pattern matching on lists

For example, we can write a function that adds up a list of integers. (I wanted to name this function
`sum` but Haskell yelled at me because `sum` is already defined by the Haskell "Prelude" library.)

```
mysum :: [Integer] -> Integer

mysum []       = 0
mysum (x : xs) = x + (mysum xs)

Lec9> mysum [10, 3, 100]
113
```

The function `mysum` is defined using two clauses. The first says that if `mysum` is applied to the empty list `[]`, return `0`. The second says that if `mysum` is applied to a list of the form `x : xs`—that is, a list whose head is `x` and whose tail is `xs`—add `x` to `(mysum xs)`, which is the sum of the elements in the tail.

To see how this works, let's step `mysum [100]`.

Mentally converting between lists in comma-bracket form (like `[1, 2, 3]` or `[100]`) and lists built using "cons" and "nil" takes some practice; if you haven't been doing that for years, it's good to write out the conversion. Otherwise it's harder to figure out what substitution we should use when we step.

We convert `[100]`, which is a list whose head is `100` and whose tail is the empty list, to

```
(100 : [])
```

Now, instead of starting w   . (If we didn't have
parentheses around   uld be a syntax error
(because : is infix); if it w   lied to three arguments,
which wouldn't make sen

```
    mysum (100 : [])
=> ???
```

To step this, we do the usual function application with pattern matching: go through each clause of the definition of `mysum` until we find one that can be "lined up" with the argument, `(100 : [])`.

First we try to line up `(100 : [])` and `[]`.

```
mysum (100 : [])
mysum []
```

This doesn't match because `[]` and `:` (say: "nil" and "cons") are different constructors. It would be like saying that `False` matches `True`.

So we continue to the second clause of `mysum`:

```
mysum (100 : [])
mysum (x    : xs)
```

This works: `100` lines up with `x`, and `[]` with `xs`, which gives us the substitution "100 for x, [] for xs". (Pronounced: "...nil for xs".)

```
   mysum (100 : [])   by function application
=> x + (mysum xs)     with substitution 100 for x, [] for xs
 = 100 + (mysum [])
```

Now we step the new function application `mysum []`. Here, the argument `[]` lines up perfectly with the pattern `[]` in the first clause, so we replace `mysum []` with the right-hand side, `0`.

```
    100 + (mysum [])
=> 100 + 0              by function application
=> 100                  by arithmetic
```

Since the pattern `[]` in the first clause of `mysum` has no pattern variables (unlike the second clause which has two pattern variables, `x` and `xs`), there is no substitution. If you like, you can write "with the empty substitution".

■ **Exercise 1.** Step `mysum [10, 3, 100]`.
(After a few steps, you'll get an expression that includes `mysum (100 : [])`, which we already stepped to 100.)

For another example of pattern matching, let's look at a function that appends two lists of integers. For example, `myappend [6, 5, 4] [1, 2, 3]` should return `[6, 5, 4, 1, 2, 3]`.

```
myappend :: [Integer] -> [Integer] -> [Integer]
myappend [] ys = ys
myappend (h : t) ys = h : (myappend t ys)
```

This works (try it), but I' [...] structures: the tail of a list is [...] `mysum`, where we were given only one list, it's not obvious *which* of the two lists to "take apart" by pattern matching.

One (very reasonable) way to figure out which list we need to take a [...] need to take apart the second argument, try to write the code, and get s

```
myappend0 :: [Integer] -> [Integer] -> [Integer]
myappend0 xs [] = xs
myappend0 xs (h : t) =
```

This starts out promisingly: what is `xs` appended with `[]`? Well, that's `xs` with an empty list added to the end, which is just `xs`. So the first clause works out.

The second clause is where we run into problems. In most recursive functions on 'data' types (like lists), we need to make a recursive call where at least one thing gets smaller. (If nothing gets smaller, like if we tried `myappend0 xs (h : t)`, we would recurse forever. Something needs to get smaller, for the same reason that the induction hypothesis in an inductive proof has to be given something smaller.) The tail `t` is smaller than the second argument `(h : t)`, so whatever we write in the second clause probably involves a call to `myappend0 xs t`:

```
myappend0 xs (h : t) = ??? (myappend0 xs t) ???
```

Suppose that `xs` is `[1, 2]` and the second argument is `[3, 4, 5]`. Then `h` is 3 and `t` is `[4, 5]`. If calling `myappend0 xs t` did what we want it to, it would return `[1, 2, 4, 5]`—`[1, 2]` appended with `[4, 5]`.

But we want to get `xs` appended with `h : t`. So we really want to return

```
[1, 2, 3, 4, 5]
```

Having the list [1, 2, 4, 5] doesn't really help us build [1, 2, 3, 4, 5]. Since this isn't going anywhere, we give up and try to pattern-match on the first argument:

```
myappend :: [Integer] -> [Integer] -> [Integer]
myappend []      ys = ys
myappend (h : t) ys = ??? (myappend t ys) ???
```

In the second clause, from experience, I know that I probably need to call myappend on t and ys. The reason we won't get stuck now is that myappend t ys gives us something useful!

Again, suppose that the first argument is [1, 2] and the second argument is [3, 4, 5]. That means h is 1, t is [2], and ys is [3, 4, 5].

Assuming myappend t ys behaves as it's supposed to, it will return [2] appended with [3, 4, 5]:

```
[2, 3, 4, 5]
```

We want to get [1, 2, 3, 4, 5]. We have h, which is 1 and myappend t ys, which is [2, 3, 4, 5]. Therefore

```
h : (myappend t ys)
```

returns 1 : [2, 3, 4, 5]

(Repeated from abo

```
myappend :: [Integer] -> [Integer] -> [Integer]
myappend []      ys = ys
myappend (h : t) ys = h : (myappend t ys)
```

1.3 List elements must have the same type

Haskell does not directly allow "heterogeneous" lists—lists of elements of different types. So a list whose first element is 5 and whose second element is False is rejected:

```
Prelude> :type [5, False]

<interactive>:1:2:
    No instance for (Num Bool) arising from the literal '5'
    In the expression: 5
    In the expression: [5, False]
```

The error message is confusing: Haskell is trying to explain that it wanted to make a list of numbers (Haskell has other numeric types than integers), but a Boolean is not a number.

The error message we get when trying to put a Boolean and a function in the same list is somewhat more clear: "can't match expected type '*function type*' with actual type 'Bool' ".

```
Prelude> :type [False, \x -> x]

<interactive>:1:9:
    Couldn't match expected type 't0 -> t0' with actual type 'Bool'
    The lambda expression '\ x -> x' has one argument,
    but its type 'Bool' has none
    In the expression: \ x -> x
    In the expression: [False, \ x -> x]
```

We can get around the requirement that list elements have the same type by declaring our own 'data' type. If we declare a 'data' type with two constructors, one that takes an Int argument and one that takes a Bool argument, we can build lists that contain a mix of Ints and Bools:

```
data IntOrBool = AnInt Int
               | ABool Bool

mixed_list :: [IntOrBool]
mixed_list = [AnInt 3, ABool True, AnInt 10]
```

Haskell accepts mixed_list because all of its elements do have the same type, IntOrBool. i

■ **Exercise 2.** Write a   hat adds up the elements, treating AnInt x as x

For example, sum_   return $3 + 1 + 10 = 14$.

(An earlier version of this file asked you to write a function of type   Bool] -> Integer, which is possible but more work than I intended.