Notes for Lecture 5 (F. 2022 Week 3 part 1): Characters; more examples of guards and recursion

Jana Dunfield

September 18, 2022

The code for this lecture is in `lec5.hs`.

1 Char type

Like many languages you may be familiar with, Haskell has a type of single characters, written in single quotes:

```
capitalA :: Char
capitalA = 'A'
```


Strings, which consist of zero or more characters, are (also like many other languages) written in double quotes:


```
Prelude> "a string"
"a string"
Prelude> ""
""
```


Remark 1. Non-ASCII characters, such as letters not common in English and mathematical symbols, may not work very well:

```
Prelude> "é ∀ ≠"

"\233 \8704 \8800"
```

The type of a single character is `Char`. The type of a string is `String`, but `String` is a *type synonym*: `String` is a convenient name for the type `[Char]`, the type of *lists* of `Char`s.

```
Prelude> :type "AAAAAAAAAAAAAA"
"AAAAAAAAAAAAAA" :: [Char]
```

For example, the string `"abc"` is actually the list `['a', 'b', 'c']`.

```
Prelude> "abc" == ['a', 'b', 'c']
True
```

Regardless of which one you write, Haskell prints the (more compact) string version, in double quotes, not the list.

```
Prelude> ['a', 'b', 'c']
"abc"
```

If we write a type declaration that says `String`, Haskell will display the type we wrote rather than `[Char]`.

```
coherent :: [Char]
coherent = "A"        -- just one character, but in " " so [Char], not Char

withdeclaration :: String
withdeclaration = "abc"      -- try  :type withdeclaration
```

```
*Lec5> :type coherent
coherent :: [Char]
*Lec5> :type withdeclaration
withdeclaration :: String
```

2 Guards

See the examples `myAbs`, `is_upper`, `is_lower` in `lec5.hs`.

Exercise 1. Rewrite `is_lower` to use guards.

(I prefer the original version. This isn't a change I would actually make, but it can help you get used to thinking about what Haskell functions do.)

3 Recursion and stepping

The function `two_raised` computes 2 raised to the power n. For example, `two_raised 8` evaluates to 256.

```
--                  0
-- two_raised 0 = 2  = 1

--                  n
-- two_raised n = 2     (2 to the nth power)
-- assume n >= 0
--
two_raised :: Integer -> Integer
two_raised n = if n == 0 then 1 else 2 * two_raised (n - 1)
```

This function calls itself (`two_raised (n - 1)`), so it is *recursive*.[1]

The trick to writing `two_raised` is the equation

$$2^n = 2 \cdot 2^{n-1}$$

which gives us the `else`-branch `2 * two_raised (n - 1)`.

The following is *part of* the sequence of steps for the expression `two_raised 3`.

[1] Later, we will introduce the idea of *tail recursion*. "Tail-recursive" is not a fancy name for "recursive"; every tail-recursive function is recursive, but not every recursive function is tail-recursive.

```
   two_raised 3
=> (if n == 0 then 1 else 2 * two_raised (n - 1))[3/n]
=  (if 3 == 0 then 1 else 2 * two_raised (3 - 1))
=> (if False  then 1 else 2 * two_raised (3 - 1))
=> 2 * two_raised (3 - 1)
=> 2 * two_raised 2
=> ...
=> 2 * (2 * two_raised (2 - 1))
=> 2 * (2 * two_raised 1)
=> ...
=> 2 * (2 * (2 * (two_raised (1 - 1))))
=> 2 * (2 * (2 * (two_raised 0)))
=>=> 2 * (2 * (2 * 1))
=> 2 * (2 * 2)
=> 2 * 4
=> 8
```

I'll go into some detail about the first step:


```
   two_raised 3
=> (if n == 0 then 1 else 2 * two_raised (n - 1))[3/n]
=  (if 3 == 0 then 1 else 2 * two_raised (3 - 1))
```


In the first step, I'm writing $\ldots[3/n]$ to mean "$\ldots$ with 3 substituted for n". The next line, the one that begins with =, is *not* a step: the expression


```
(if n == 0 then 1 else 2 * two_raised (n - 1))[3/n]
```

is another way of writing the expression

```
(if 3 == 0 then 1 else 2 * two_raised (3 - 1))
```

not a separate step of computation. Since these are two different ways of writing the same expression, we would accept either in a stepping question on a quiz or assignment. We would also accept writing out both, as I do in `lec5.hs`.

4 Infinite recursion

```
-- Some functions that loop forever
-- (and that may need unusual interventions to interrupt them)
danger_zone :: Integer -> Integer
danger_zone n = 2 * danger_zone n

maybe_danger_zone :: Integer -> Integer
maybe_danger_zone n = maybe_danger_zone n
```

Be careful when you try applying these functions; you may need to interrupt, "Force Quit", or do whatever your OS calls the operation of stopping a process/program/application.

On my laptop, applying `maybe_danger_zone` just sits there until I interrupt it (on macOS this should be Control-C; on Windows, it's probably Control-Z):

```
*Lec5> maybe_danger_zone 1
^CInterrupted.
```

`danger_zone` is more difficult to interrupt; I was able to press Control-C several times before the system noticed.

```
*Lec5> danger_zone 1
^C^C^C^C^CInterrupted.
```

Exercise 2. Neither function ever returns a result, but one is more difficult to interrupt. Why do you think that is?

Hints:

- The `Integer` type in Haskell is *arbitrary precision*: Haskell allows very large numbers, even those that don't fit within 64 bits (the "natural" integer size on your computer's CPU, if your CPU is typical). So `danger_zone` will not stop multiplying when the number exceeds 2^{64}.
- What happens when you step `maybe_danger_zone`? What happens when you step `danger_zone`? Just write out the first few steps of each, but think about what would happen if you had to write out many steps of each.