Ruby and F# demo

September 12th

Mark Armstrong, PhD Candidate, McMaster University

Fall, 2019

Contents

1 What do you need for homework 1? 1

2 Ruby 2
2.1 "About Ruby" 2
2.2 Comparing numbers in Ruby 3
2.3 Branching in Ruby 4
2.4 Repeating in Ruby 5
2.5 Performing standard input and output in Ruby 6
2.6 Where to go in Ruby from here? 7

3 F# 7
3.1 "About F#" 7
3.2 Comparing numbers in F# 8
3.3 Branching in F# 8
3.4 Repeating in F# 9
3.5 Performing standard input and output in F# 10
3.6 Where to go in F# from here? 11

4 Leftovers 11

1 What do you need for homework 1?

To build our number guessing game, you need a few specific capabilities.

- Compare numbers.
- Branch.

- Repeat.
- Perform standard input and output.

Let's quickly run through the ways to accomplish these in both Ruby and F#.

Along the way, we'll try to do some "sight-seeing" in these languages. This means that we'll identify how to perform each task right up front, but then attempt to explore a bit deeper into how the languages are unique.

2 Ruby

2.1 "About Ruby"

Ruby is an almost purely object-oriented language, heavily inspired by Smalltalk, Lisp and Perl.

It's creator, Yukihiro "Matz" Matsumoto, has documented these inspirations in a post on ruby-talk.

> Ruby is a language designed in the following steps:
>
> - take a simple lisp language (like one prior to CL).
> - remove macros, s-expression.
> - add simple object system (much simpler than CLOS).
> - add blocks, inspired by higher order functions.
> - add methods found in Smalltalk.
> - add functionality found in Perl (in OO way).

Ruby was a language born out of Matz's interests and love of certain language features. He's said about its creation

> Well, Ruby was born on February 24 1993. I was talking with my colleague about the possibility of an object-oriented scripting language. I knew Perl (Perl4, not Perl5), but I didn't like it really, because it had smell of toy language (it still has). The object-oriented scripting language seemed very promising.
>
> I knew Python then. But I didn't like it, because I didn't think it was a true object-oriented language OO features appeared to be add-on to the language. As a language manic and OO fan for

15 years, I really wanted a genuine object-oriented, easy-to-use scripting language. I looked for, but couldn't find one.

So, I decided to make it. It took several months to make the interpreter run. I put it the features I love to have in my language, such as iterators, exception handling, garbage collection.

Then, I reorganized the features of Perl into a class library, and implemented them. I posted Ruby 0.95 to the Japanese domestic newsgroups in Dec. 1995.

2.2 Comparing numbers in Ruby

```
x = 10
y = 5

puts x
puts y
puts x < y
puts x > y
puts x == y
```

Pretty boring so far. But something that's a bit fascinating about Ruby is that despite this syntax being reminiscent of "procedural" languages such as C, "behind the scenes" (most of) what we are doing is calling methods!

Let's make that explicit.

```
x = 10
y = 5

Kernel.puts(x)
Kernel.puts(y)
Kernel.puts(x.< y)
Kernel.puts(x.> y)
Kernel.puts(x.== y)
```

Neat! So the "regular" looking syntax is syntactic sugar for this explicit method calling syntax.

We can strip away more syntactic sugar; when we call these methods, we are really sending messages.

```
x = 10
y = 5
```

```
Kernel.send "puts", x
Kernel.send "puts", y
Kernel.send "puts", (x.send "<", y)
Kernel.send "puts", (x.send ">", y)
Kernel.send "puts", (x.send "==", y)
```

That's really cool. Ruby is (almost) purely object-oriented; by that, we mean that (almost) everything is an object, and (almost) everything is message-passing between objects.

2.3 Branching in Ruby

Ruby includes the classic branching operator, `if`. Use the `end` keyword to end the body.

```
x = 10000

if x > 1000 then
  puts "x is impossibly large!"
end
```

To make it a one-liner, use the `then` keyword or a `;`. Or if you don't like that, use braces `{}`.

```
x = 10000

if x > 1000 then puts "x is impossibly large!" end

if x < 100000; puts "But it's not all that impossibly large..." end
```

When writing a one-liner, consider using postfix form instead. Note we don't need `end` for this form!

```
x = 10000

puts "x is impossibly large!" if x > 1000

puts "But it's not all that impossibly large..." if x < 100000
```

Ruby has both `else` and `elsif`. So no need for the `else if` form.

```
has_if = true
has_elsif = true
has_else = true

if false then
  puts "Whoa, you're in trouble."
else
  if has_elsif then
    puts "Why didn't you use 'elsif' instead of 'else if' then?"
  elsif has_if and has_elsif then
    puts "Ah, I guess you had no choice then."
  end
end
```

In addition to `if`, Ruby includes `unless`.

```
lacks_unless = false

puts "No need for 'if not ...'!" unless lacks_unless
```

2.4 Repeating in Ruby

As you might expect from a language with both `if` and `unless`, Ruby has both `while` and `until` loops. As with `then` for conditionals, the `do` is optional unless you want a one-liner.

```
x = 0

while x <= 5 do
  puts "Counting up with while..." ++ x.to_s
  x += 1
end

until x == 0 do
  x -= 1
  puts "Counting down with until..." ++ x.to_s
end
```

Ruby also includes a variety of nifty iterating loops.

```
for x in 1..5 do
  puts "Counting up with for..." ++ x.to_s
```

```
end

5.times do |x| puts "Repeating a task with times..." ++ x.to_s end

[1,2,3,4,5].each do |x|
  puts "Iterating over a list with each..." ++ x.to_s
end
```

The last example above used a *block*; a unit of code that takes an argument. Also known as an *anonymous function* or a *lambda*.

```
f = lambda { |x, y|
  puts "Hey, how'd you call me? I'm not supposed to have a name!"
  puts "Well, anyway, take your stuff back."
  puts x
  puts y
}

g = f

f.("f's","stuff")
puts ""
g.call "g's", "things"
```

2.5 Performing standard input and output in Ruby

We've been using standard output all along here.

```
puts "Hello world"
puts "It's nice to meet you"
```

Note that `puts` (PUT String) places a newline after the string. `print` does not.

```
print "Whatever you do, "
print "don't squish these lines together!"
```

Standard input is just as easy.

```
puts "Guess my favourite number."
guess = gets
puts "How'd you know it was #{guess}?"
```

That last block showed off the fact that strings delimited by double quotes are formatted strings! Single quote strings are not formatted.

```
will = "will"
will_not = "will not"

puts "I #{will} be formatted."
puts 'I #{will_not} be formatted.'
puts "I %s also be formatted. Maybe you like my syntax more." % [will]
```

2.6 Where to go in Ruby from here?

My suggestion is to research **all of these topics** further. Read Musa's cheatsheet. Check out documentation and tutorials. We've barely scratched the surface here, and likely any Ruby expert would say I've missed several important things.

For this course, you don't need to become experts, and if there is a idiomatic way to do things in Ruby, then

1. it's not as emphasised as it sometimes is for Python, and
2. I'm not concerned with you learning it.

Instead, focus on

- finding ways to do things that are similar to what you do elsewhere,
- finding ways to do things that are **completely different** than what you do elsewhere,
- and most importantly, having some fun with it.

3 F#

3.1 "About F#"

F# is a member of the ML family of languages, born from a .NET implementation of OCaml .

You've likely worked with Haskell, and possibly Elm, both of which inherited several ideas from ML.

So, parts of F# will likely be familiar to you, even if you have never seen it before.

3.2 Comparing numbers in F#

```
let x, y = 5, 10

printfn "%b" (x > y)
printfn "%b" (x < y)
printfn "%b" (x = y)
```

You should notice from the application of `printfn` that it's curried.

```
let x, y = 5, 10
let printbool = (printfn "%b")

printbool (x < y)
printbool (x = y)
```

Of course `printfn` is (what we prefer to call) a procedure, rather than a function, because its primary purpose is the side effect of printing to the console. But does it have a value? We can use `%O` (capital letter O) to print any object; let us do so to investigate the value of `printfn`.

```
let x = printfn "Hello world"

printfn "%O" x
```

`()` is an empty tuple. How many different empty tuples can there be? And once you answer that, what's a good name for the type of empty tuples?

```
printfn "hello"
```

3.3 Branching in F#

F# has `if`, `elif` and `else`. Unlike Haskell and Elm, you don't have to provide an `else` clause.

```
if 2 = 5 then printfn "What reality am I in?"

if   2 = 7 then printfn "Now I'm even more concerned..."
elif 2 = 4 then printfn "This doesn't help either..."
else            printfn "Okay, maybe everything's okay."
```

More interesting than `if-then-elif-then-else` is pattern matching. It's a powerful tool when working with algebraic datatypes, which we'll see later on, especially for the assignment.

```
let r =
  match 5 with
  | 0 -> "Factorial is 1."
  | 1 -> "Factorial is 1."
  | 2 -> "Factorial is 2."
  | 3 -> "Factorial is 6."
  | 4 -> "Factorial is 24."
  | _ -> "I'm tired, go away."

printfn "%s" r
```

We can attach guards to the cases when pattern matching.

```
let r =
  match 5 with
  | 5 when false -> "It might match, but it can't satisfy this guard!"
  | 5 when true  -> "This guard's easily pleased."
  | _ -> "Uh... hello?"

printfn "%s" r
```

Of course pattern matching and guards shine most when defining functions/procedures.

```
let rec fact n =
  match n with
  | 0 -> 1
  | n when n > 0 -> n * fact (n - 1)
  | _ -> printfn "Can't calculate factorial of a negative!"
         -1

printfn "Factorial 5 is %d" (fact 5)
```

3.4 Repeating in F#

We've just defined a recursive function; you know that recursion will allow us to repeat, potentially infinitely. Note the type annotation; we omit them usually, but occasionally the type checker needs our input.

```
let rec forever (x : int) : int =
  printfn "All work and no play makes F# a dull language..."
  forever x
```

Then call, for instance, `forever 10` to run until you get tired and interrupt it.

As far as I can tell, we cannot create such an infinite loop with type `unit` (i.e., as a constant, i.e., a non-function). We can cheat a bit an make it a function on `unit` instead, (note the type annotation), so then the only possible argument is `()`

```
let rec forever' (_ : unit) : Lazy<unit> =
  printfn "I can do it, I can do it ∞ times!"
  forever'()
```

Then call `forever'()` to run until you get tired of it.

Given what you know about procedure calls and the memory stack, you might be expecting that this is not truly an infinite loop; eventually, we should run out of room on the memory stack, and the program should crash.

But that doesn't happen! The reason is *tail recursion*, which we'll discuss in detail at some point; for now, suffice to say that the F# runtime is smart, and reuses stack space.

Now, perhaps you still don't like the idea of infinite recursion. That's okay! F# also includes loops for you. If you want to name the loop, you'd better still mark it `lazy`.

```
let forever : Lazy<unit> =
  lazy while true do
    printfn "Much better..."
```

There are also `for..in` and `for..to` loops you can explore.

```
for x in [1;2;3] do
  printfn "Iterating through a list %d" x

for x = 1 to 3 do
  printfn "Counting up %d" x

for x = 3 downto 1 do
  printfn "Counting down %d" x
```

3.5 Performing standard input and output in F#

We've been printing formatted strings with `printfn`.

There's also the method `System.Console.WriteLine()` for outputting strings (not formatted).

Unfortunately, there's no "`writefn`" analagous to `printfn`. But there is `System.Console.ReadLine()`.

```
let value = System.Console.ReadLine()

System.Console.WriteLine("Is there an echo in here?");
System.Console.WriteLine(value)
```

3.6 Where to go in F# from here?

Assuming you are familiar with either Haskell or Elm, explore how F# is similar to and/or differs from them.

While they are all within the functional paradigm, F# does not encourage purity in the same way. You may find this makes starting out in the language simpler.

Explore concepts you may know from elsewhere in F#, including:

- algebraic datatypes, such as trees or lists,
- using recursive algorithms instead of iterative algorithms, and
- defining higher-order functions; functions which take functions as arguments.

If you are not already, **become comfortable with recursion**! F# has loops and mutable state, but some languages we consider later will not!

4 Leftovers

I wrote this function when starting out with repetition in F#; it doesn't fit the narrative above now, but it can live here as a leftover.

```
let rec forever : Lazy<sbyte> =
  let rec forever_helper (so_far : sbyte) : sbyte =
    match so_far with
    | 0y -> printfn "Starting out..."
            forever_helper 1y
    | n when n % 100y = 0y -> printfn "Took a hundred steps..."
                              forever_helper (n + 1y)
    | n when n = -1y -> printfn "Sorry, fell asleep and overflowed a bit!"
```

```
                           forever_helper (n + 1y)
  | n -> forever_helper (n + 1y)
in lazy forever_helper 0y
```

Run it with `forever.Force()`, if you don't mind looping for a while.