Elixir 1.6 Exercises

Chapter 2: Pattern Matching

Exercise: Pattern Matching-1 (Page 18)

- Which of the following would match?
  - `a = [1, 2, 3]`
  - `a = 4`
  - `4 = a`
  - `[a, b] = [ 1, 2, 3 ]`
  - `a = [ [ 1, 2, 3 ] ]`
  - `[ a..5 ] = [ 1..5 ]`
  - `[a] = [ [ 1, 2, 3 ] ]`
  - `[[a]] = [ [ 1, 2, 3 ] ]`

A Possible Solution</summary>

```
a = [1, 2, 3]       #=> a → [1, 2, 3]
a = 4               #=> a → 4
4 = a               # assuming prior assignment

[a, b] = [ 1, 2, 3 ]
# ** (MatchError) no match of right hand side value: [1, 2, 3]
#    :erl_eval.expr/3

a = [ [ 1, 2, 3 ] ]   #=> a → [[1, 2, 3]]
[ a..5 ] = [ 1..5 ]   #=> a → 1
[a] = [ [ 1, 2, 3 ] ] #=> a → [1,2,3]

[[a]] = [ [ 1, 2, 3 ] ]
# ** (MatchError) no match of right hand side value: [[1, 2, 3]]
#    :erl_eval.expr/3
```

</details>

Exercise: Pattern Matching-2 (Page 19)

- Which of the following will match?
    - `[ a, b, a ] = [ 1, 2, 3 ]`
    - `[ a, b, a ] = [ 1, 1, 2 ]`
    - `[ a, b, a ] = [ 1, 2, 1 ]`

A Possible Solution</summary>

```
[ a, b, a ] = [ 1, 2, 3 ]
# ** (MatchError) no match of right hand side value: [1, 2, 3]

[ a, b, a ] = [ 1, 1, 2 ]
# ** (MatchError) no match of right hand side value: [1, 1, 2]


[ a, b, a ] = [ 1, 2, 1 ] #=> a → 1, b → 2

[ a, b, a ] = [ 1, 1, 1 ] #=> a → 1, b → 1
```

</details>

Exercise: Pattern Matching-3 (Page 20)

- If you assume the variable `a` initially contains the value `2`, which of the following will match?
    - `[ a, b, a ] = [ 1, 2, 3 ]`
    - `[ a, b, a ] = [ 1, 1, 2 ]`
    - `a = 1`
    - `^a = 2`
    - `^a = 1`
    - `^a = 2 - a`

A Possible Solution</summary>

```
a = 2         #=> a → 2

[ a, b, a ] = [ 1, 2, 3 ]
# ** (MatchError) no match of right hand side value: [1, 2, 3]
#     :erl_eval.expr/3

[ a, b, a ] = [ 1, 1, 2 ]
# ** (MatchError) no match of right hand side value: [1, 1, 2]
#     :erl_eval.expr/3

a = 1         #=> a → 1

^a = 2
# ** (MatchError) no match of right hand side value: 2
#    :erl_eval.expr/3

^a = 1        #=> matches. a still 1

^a = 2 - a    #=> matches. a still 1
```

</details>

Chapter 5: Anonymous Functions

Exercise: Functions-1 (Page 43)

- Go into iex. Create and run the functions that do the following
    - `list_concat.([1,2,3], [4,5,6]) #=> [1,2,3,4,5,6]`
    - `sum.(1, 2, 3) #=> 6`
    - `pair_tuple_to_list.( { 8, 7 } ) #=> [ 8, 7 ]`

A Possible Solution</summary>

```
iex(1)> list_concat = fn a, b -> a ++ b end
#Function<erl_eval.12.17052888>
iex(2)> list_concat.([1,2,3], [4,5,6])
[1, 2, 3, 4, 5, 6]

iex(3)> sum = fn a, b, c -> a + b + c end
#Function<erl_eval.18.17052888>
iex(4)> sum.(1,2,3)
6
```

```
iex(5)> pair_tuple_to_list = fn {a, b} -> [ a, b ] end
#Function<erl_eval.6.17052888>
iex(6)> pair_tuple_to_list.({ 8, 7 })
[8, 7]
```

</details>

Exercise: Functions-2 (Page 45)

- **Write a function that takes three arguments. If the first two are zero, return "FizzBuzz". If the first is zero, return "Fizz". If the second is zero return "Buzz". Otherwise return the third argument. Do not use any language features that we haven't yet covered in this book.**

A Possible Solution</summary>

```
iex(1)> fizz_word = fn
...(1)>   0, 0, _ -> "FizzBuzz"
...(1)>   0, _, _ -> "Fizz"
...(1)>   _, 0, _ -> "Buzz"
...(1)>   _, _, n -> n
...(1)> end
#Function<erl_eval.18.17052888>

iex(2)> fizz_word.(0, 0, 1)
"FizzBuzz"

iex(3)> fizz_word.(0, 1, 1)
"Fizz"

iex(4)> fizz_word.(1, 0, 1)
"Buzz"

iex(5)> fizz_word.(1, 1, 1)
1
```

</details>

Exercise: Functions-3 (Page 45)

- The operator `rem(a, b)` returns the remainder after dividing `a` by `b`. Write a function that takes a single integer (`n`) calls the function in the previous exercise, passing it `rem(n,3)`, `rem(n,5)`, and `n`. Call it 7 times with the arguments 10, 11, 12, etc. You should get “Buzz, 11, Fizz, 13, 14, FizzBuzz”, 16”.
- (Yes, it’s a FizzBuzz[1] solution with no conditional logic).

A Possible Solution</summary>

```
iex(3)> fb = fn n ->
...(3)>   fizz_word.(rem(n, 3), rem(n, 5), n)
...(3)>   end
#Function<erl_eval.6.17052888>

iex(4)> [ fb.(10), fb.(11), fb.(12), fb.(13), fb.(14), fb.(15), fb.(16) ]
["Buzz", 11, "Fizz", 13, 14, "FizzBuzz", 16]
```

</details>

1. http://c2.com/cgi/wiki?FizzBuzzTest
2. ↩

Exercise: Functions-4 (Page 47)

- Write a function `prefix` that takes a string. It should return a new function that takes a second string. When that second function is called, it will return a string containing the first string, a space, and the second string.
-
```
iex> mrs = prefix.("Mrs")
#Function<erl_eval.6.82930912>
iex> mrs.("Smith")
"Mrs Smith"
iex> prefix.("Elixir").("Rocks")
```

```
"Elixir Rocks"
```

A Possible Solution</summary>

```
iex> prefix = fn prefix -> fn str -> "#{prefix} #{str}" end end
#Function<erl_eval.6.17052888>

iex> mrs = prefix.("Mrs")
#Function<erl_eval.6.17052888>

iex> mrs.("Smith")
"Mrs Smith"

iex> prefix.("Elixir").("Rocks")
"Elixir Rocks"
```

</details>

Exercise: Functions-5 (Page 50)

- Use the `&1`,... notation to rewrite the following.
    - `Enum.map [1,2,3,4], fn x -> x + 2 end`
    - `Enum.each [1,2,3,4], fn x -> IO.puts x end`

A Possible Solution</summary>

```
iex(1)> Enum.map [1,2,3,4], &1 + 2
[3, 4, 5, 6]

iex(2)> Enum.each [1,2,3,4], IO.puts(&1)
1
2
3
4
:ok
```

</details>

Chapter 6: Modules and Named Functions

Exercise: Modules and Functions-1 (Page 55)

- Extend the `Times` module with a `triple` function, that multiplies its parameter by three.

A Possible Solution</summary>

```
defmodule Times do
 def double(n), do: n * 2
 def triple(n), do: n * 3
end
```

</details>

Exercise: Modules and Functions-2 (Page 55)

- Run the result in iex. Use both techniques to compile the file.

A Possible Solution</summary>

```
# Load our module into iex as it starts
#

$ iex times.exs
iex(1)> Times.triple 4
12
iex(2)>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
^C

# Load it in after it starts
#

$ iex
iex(1)> c "times.exs"
[Times]
iex(2)> Times.triple 7
21
```

```
iex(3)>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
^C
```

</details>

Exercise: Modules and Functions-3 (Page 55)

- Add a `quadruple` function. (Maybe it could call the `double` function….)

```
defmodule Times do
  def double(n),    do: n * 2
  def triple(n),    do: n * 3
  def quadruple(n), do: double(double(n))
end

IO.puts Times.quadruple(6)   #=> 24
```

Exercise: Modules and Functions-4 (Page 57)

- Implement and run a function `sum(n)` that uses recursion to calculate the sum of the integers from 1 to *n*. You'll need to write this function inside a module in a separate file. Then load up iex, compile that file, and try your function.

A Possible Solution</summary>

```
defmodule Recursive do
  def sum(0), do: 0
  def sum(n), do: n + sum(n-1)
end
```

```
$ iex recursive.exs
iex(1)> Recursive.sum(4)
10
iex(2)> Recursive.sum(5)
15
```

</details>

Exercise: Modules and Functions-5 (Page 57)

- **Write a function `gcd(x,y)` that finds the greatest common divisor between two nonnegative integers. Algebraically, *gcd(x,y)* is *x* if *y* is zero, *gcd(y, rem(x,y))*otherwise.**

A Possible Solution</summary>

```
defmodule MyMath do

  def gcd(x, 0), do: x
  def gcd(x, y), do: gcd(y, rem(x, y))

end


IO.puts MyMath.gcd(20, 15)   #=> 5
IO.puts MyMath.gcd(20, 16)   #=> 4
IO.puts MyMath.gcd(23, 17)   #=> 1
```

</details>

Exercise: Modules and Functions-6 (Page 62)

- *I'm thinking of a number between 1 and 1000…*
- The most efficient way to find the number is to guess halfway between the low and high numbers of the range. If our guess is too big, then the answer lies between the bottom of the range and one less than our guess. If it is too small, then the answer lies between one more than our guess and the end of the range.
- Code this up. Your API will be `guess(actual, range)`, where `range` is an Elixir range.
- Your output should look similar to:
- `iex> Chop.guess(273, 1..1000)`

  `Is it 500`

```
Is it 250
Is it 375
Is it 312
Is it 281
Is it 265
Is it 273
273
```

- Hints:
    - You may need to implement helper functions with an additional parameter (the currently guessed number).
    - the `div(a,b)` function performs integer division
    - guard clauses are your friends
    - patterns can match the low and high parts of a range (`a..b=4..8`) === {::comment} defmodule Chop do def guess(actual, range = low..high) do guess = div(low+high, 2) IO.puts “Is it #{guess}” guess(actual, guess, range) end
- def guess(actual, actual, _), do: actual def guess(actual, guess, _low..high) when guess < actual, do: guess(actual, guess+1..high) def guess(actual, guess, low.._high) when guess > actual, do: guess(actual, low..guess-1) end {:/comment}

</yourturn>

A Possible Solution</summary>

```
defmodule Chop do
  def guess(actual, range = low..high) do
    guess = div(low+high, 2)
    IO.puts "Is it #{guess}?"
    _guess(actual, guess, range)
  end

  defp _guess(actual, actual, _),
    do: IO.puts "Yes, it's #{actual}"

  defp _guess(actual, guess,  _low..high)
```

```
  when guess < actual,
  do: guess(actual, guess+1..high)

defp _guess(actual, guess,  low.._high)
  when guess > actual,
  do: guess(actual, low..guess-1)
end

Chop.guess(273, 1..1000)
```

</details>

Exercise: Modules and Functions-7 (Page 70)

- Find the library functions to do the following, and then use each in iex. (If there's the word Elixir or Erlang at the end of the challenge, then you'll find the answer in that set of libraries.)
    - **Convert a float to a string with 2 decimal digits. (Erlang)**
    - **Get the value of an operating system environment variable. (Elixir)**
    - **Return the extension component of a file name (so return `.exs` if given `"dave/test.exs"` (Elixir)**
    - **Return the current working directory of the process. (Elixir)**
    - **Convert a string containing JSON into Elixir data structures. (Just find, don't install)**
    - **Execute an command in your operating system's shell**

<details>
<summary>A Possible Solution</summary>

```
# Convert a float to a string with 2 decimal digits.
iex> :io.format("~.2f~n", [2.0/3.0])
0.67
:ok

# Get the value of an operating system environment variable.
iex> System.get_env("HOME")
"/Users/dave"

# Return the extension component of a file name
```

```
iex> Path.extname("dave/test.exs")
".exs"

# Return the current working directory of the process
iex> System.cwd
"/Users/dave/BS2/titles/elixir/Book/yourturn/ModulesAndFunctions"

# Convert a string containing JSON into Elixir data structures
# There are many options. Some, such as https://github.com/guedes/exjson,
# are specifically for Elixir. Others, such as https://github.com/hio/erlang-json
# are Elnag libraries that are usable from Elixir.

# Execute an command in your operating system's shell
iex> System.cmd("date")
"Sun Jul 14 15:04:06 CDT 2013\n"
```

</details>

Chapter 7: Lists and Recursion

Exercise: Lists and Recursion-1 (Page 77)

- Write a function `mapsum` that takes a list and a function. It applies the function to each element of the list, and then sums the result, so
- `iex> MyList.mapsum [1, 2, 3], &1 * &1`
  `14`

A Possible Solution</summary>

```
defmodule MyList do

  def mapsum([], _fun),          do: 0
  def mapsum([ head | tail ], fun), do: fun.(head) + mapsum(tail, fun)

end

IO.puts MyList.mapsum([1, 2, 3], &1 * &1)  #=> 14
```

</details>

Exercise: Lists and Recursion-2 (Page 77)

- Write `max(list)` that returns the element with the maximum value in the list. (This is slightly trickier than it sounds.)

A Possible Solution</summary>

```
# Our solution uses the built-in max/2 function, which
# returns the larger of its two numeric arguments.
# Although it isn't necessary, we call it as
# `Kernel.max` to avoid confusion

defmodule MyList do

 # max([]) is undefined...

 # max of a single element list is that element
 def max([x]), do: x

 # else recurse
 def max([ head | tail ]), do: Kernel.max(head, max(tail))

end

IO.puts MyList.max([4])      #=> 4
IO.puts MyList.max([5, 4, 3]) #=> 5
IO.puts MyList.max([4, 5, 3]) #=> 5
IO.puts MyList.max([3, 4, 5]) #=> 5
```

</details>

Exercise: Lists and Recursion-3 (Page 78)

- An Elixir single quoted string is actually a list of individual character codes. Write a function `caesar(list, n)` that adds `n` to each element of the list, but wrapping if the addition results in a character greater than *z*.

- `iex> MyList.caesar('ryvkve', 13)`

  `?????? :)`

A Possible Solution</summary>

```
defmodule MyList do

  def caesar([], _n), do: []

  def caesar([ head | tail ], n)
    when head+n <= ?z,
    do: [ head+n | caesar(tail, n) ]

  def caesar([ head | tail ], n),
    do: [ head+n-26 | caesar(tail, n) ]

end

IO.puts MyList.caesar('ryvkve', 13)  #=> elixir
```

</details>

Exercise: Lists and Recursion-4 (Page 81)

- **Write a function MyList.span(from, to) that returns a list of the numbers from `from` up to `to`.**

A Possible Solution</summary>

```
defmodule MyList do

  def span(from, to) when from > to, do: []

  def span(from, to) do
    [ from | span(from+1, to) ]
  end

end

IO.inspect MyList.span(5, 10)
```

</details>

Chapter 10: Processing Collections-Enum and Stream

Exercise: Lists and Recursion-5 (Page 102)

- Implement the following Enum functions using no library functions or list comprehensions: `all?`, `each`, `filter`, `split`, and `take`

A Possible Solution</summary>

```
defmodule MyList do

 def all?(list),     do: all?(list, fn x -> !!x end) # !! converts truthy to `true`
 def all?([], _fun), do: true
 def all?([ head | tail ], fun) do
   if fun.(head) do
     all?(tail, fun)
   else
     false
   end
 end

 def each([], _fun), do: []
 def each([ head | tail ], fun) do
   [ fun.(head) | each(tail, fun) ]
 end

 def filter([], _fun), do: []
 def filter([ head | tail ], fun) do
   if fun.(head) do
     [ head, filter(tail, fun) ]
   else
     [ filter(tail, fun) ]
   end
 end

 def split(list, count),      do: _split(list, [], count)
 defp _split([], front, _),   do: [ Enum.reverse(front), [] ]
 defp _split(tail, front, 0), do: [ Enum.reverse(front), tail ]
 defp _split([ head | tail ], front, count)  do
   _split(tail, [head|front], count-1)
 end

 def take(list, n), do: hd(split(list, n))
```

```
end


IO.inspect MyList.all?([])              #=> true
IO.inspect MyList.all?([true, true])    #=> true
IO.inspect MyList.all?([true, false])   #=> false
IO.inspect MyList.all?([4, 5, 6], &1 > 3)  #=> true

MyList.each([1,2,3], IO.puts(&1))       #=> 1/2/3

IO.inspect MyList.split([1,2,3,4,5,6], 3)  #=> [[1, 2, 3], [4, 5, 6]]

IO.inspect MyList.take('pragmatic', 6)    #=> 'pragma'
```

</details>

Exercise: Lists and Recursion-6 (Page 102)

- (Harder) Write a function `flatten(list)` that takes a list that may contain any number of sublists, and those sublists may contain sublists, to any depth. It returns the elements of these lists as a flat list.
- `iex> MyList.flatten([ 1, [ 2, 3, [4] ], 5, [[[6]]]])`
  `[1,2,3,4,5,6]`

- *Hint: You may have to use* ***`Enum.reverse`*** *to get your result in the correct order.*

<details>
<summary>A Possible Solution</summary>

```
# The simplest version is probably to use list concatenation. However,
# this version ends up rebuilding the list at each step
defmodule UsingConcat do
  def flatten([]), do: []
  def flatten([ head | tail ]), do: flatten(head) ++ flatten(tail)
  def flatten(head), do: [ head ]
end

# This version is more efficient, as it picks successive head values
# from a list, adding them to `result`. It is also tail recursive.
# The trick is that we have to unnest the head if the head itself is a
# list.
```

```elixir
defmodule MyList do

  def flatten(list), do: _flatten(list, [])

  defp _flatten([], result), do: Enum.reverse(result)

  # The following two function heads deal with the head
  # being a list
  defp _flatten([ [ h | [] ] | tail], result) do
    _flatten([ h | tail], result)
  end

  defp _flatten([ [ h | t ] | tail], result) do
    _flatten([ h, t | tail], result)
  end

  # Otherwise the head is not, and we can collect it
  defp _flatten([ head | tail ], result) do
    _flatten(tail, [ head | result ])
  end

end

IO.inspect MyList.flatten([ 1, [ 2, 3, [4] ], 5, [[[6]]]])

# José Valim came up with a different implementation. It is interesting
# to spend some time with this, because it breaks down the problem
# a little differently. Rather that extract individual elements
# to built the result list, it treats the original list more like
# a tree, flattening subtrees on the right and merging the results
# into a tree that itself gets flattened. It is tider, and I prefer
# it to my solution.

defmodule JVList do
  def flatten(list), do: do_flatten(list, [])

  def do_flatten([h|t], tail) when is_list(h) do
    do_flatten(h, do_flatten(t, tail))
  end

  def do_flatten([h|t], tail) do
    [h|do_flatten(t, tail)]
  end

  def do_flatten([], tail) do
    tail
  end
end
```

</details>

Exercise: Lists and Recursion-7 (Page 114)

- **Use your span function and list comprehensions to return a list of the prime numbers from 2 to n.**

A Possible Solution</summary>

```
defmodule MyList do

 def span(from, to) when from > to, do: []

 def span(from, to), do: [ from | span(from+1, to) ]

 def primes_up_to(n) do
   range = span(2, n)
   range -- (lc a inlist range, b inlist range, a <= b, a*b <= n, do: a*b)
 end

end

IO.inspect MyList.primes_up_to(40) #=> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

</details>

Exercise: Lists and Recursion-8 (114)

- Pragmatic Bookshelf has offices in Texas (TX) and North Carolina (NC), so we have to charge sales tax on orders shipped to these states. The rates can be expressed as a keyword list[1]
- `tax_rates = [ NC: 0.075, TX: 0.08 ]`

- Here's a list of orders:
- ```
  orders = [
    [ id: 123, ship_to: :NC, net_amount: 100.00 ],
  ```
  ```

```
  [ id: 124, ship_to: :OK, net_amount:  35.50 ],
  [ id: 125, ship_to: :TX, net_amount:  24.00 ],
  [ id: 126, ship_to: :TX, net_amount:  44.80 ],
  [ id: 127, ship_to: :NC, net_amount:  25.00 ],
  [ id: 128, ship_to: :MA, net_amount:  10.00 ],
  [ id: 129, ship_to: :CA, net_amount: 102.00 ],
  [ id: 120, ship_to: :NC, net_amount:  50.00 ] ]
```

- Write a function that takes both lists and returns a copy of the orders, but with an extra field, `total_amount` which is the net plus sales tax. If a shipment is not to NC or TX, there's no tax applied.

A Possible Solution</summary>

```
defmodule Tax do

 def orders_with_total(orders, tax_rates) do
   orders |> Enum.map(add_total_to(&1, tax_rates))
 end

 def add_total_to(order = [id: _, ship_to: state, net_amount: net], tax_rates) do
   tax_rate = Keyword.get(tax_rates, state, 0)
   tax      = net*tax_rate
   total    = net+tax
   Keyword.put(order, :total_amount, total)
 end

end


tax_rates =  [ NC: 0.075, TX: 0.08 ]

orders = [
  [ id: 123, ship_to: :NC, net_amount: 100.00 ],
  [ id: 124, ship_to: :OK, net_amount:  35.50 ],
  [ id: 125, ship_to: :TX, net_amount:  24.00 ],
  [ id: 126, ship_to: :TX, net_amount:  44.80 ],
  [ id: 127, ship_to: :NC, net_amount:  25.00 ],
  [ id: 128, ship_to: :MA, net_amount:  10.00 ],
  [ id: 129, ship_to: :CA, net_amount: 102.00 ],
  [ id: 120, ship_to: :NC, net_amount:  50.00 ]
]

IO.inspect Tax.orders_with_total(orders, tax_rates)
```

</details>

1. I wish it were that simple.... ↩

Chapter 11: Strings and Binaries

Exercise: Strings and Binaries-1 (Page 123)

- **Write a function that returns true if a single-quoted string contains only printable ASCII characters (space through tilde).**

A Possible Solution</summary>

```
defmodule MyString do

  def printable_ascii(sqs), do: Enum.all?(sqs, fn ch -> ch in ? ..?~ end)

end

IO.inspect MyString.printable_ascii('cat!')   #=> true
IO.inspect MyString.printable_ascii('∂x/∂y')  #=> false
```

</details>

Exercise: Strings and Binaries-2 (Page 123)

- **Write `anagram?(word1, word2)` that returns true if its parameters are anagrams.**

A Possible Solution</summary>

```
defmodule MyString do

  def anagram(sqs1, sqs2), do: Enum.sort(sqs1) == Enum.sort(sqs2)

end

IO.inspect MyString.anagram('cat', 'act')          #=> true
IO.inspect MyString.anagram('cat', 'actor')        #=> false
```

```
IO.inspect MyString.anagram('incorporates', 'procreations')  #=> true
```

</details>

Exercise: Strings and Binaries-3 (Page 123)

- Try the following in iex:
- `iex> [ 'cat' | 'dog' ]`
  `['cat',100,111,103]`

- Why does iex print ‘cat’ as a string, but ‘dog’ as individual numbers?

A Possible Solution</summary>

```
# Because the head of the new list is actually the list [?c, ?a, ?t].
# This means the overall list consists of a list and three ASCII
# characters:
#
#   [ 'cat' | 'dog' ] = [ [ ?c, ?a, ?t ], ?d, ?o, ?g ]
#
# Because the overall list contains something other than ASCII
# characters, it is displayed as a list of values. But the first value is
# the list 'cat', which _is_ just ASCII characters.
```

</details>

Exercise: Strings and Binaries-4 (Page 123)

- (Harder) Write a function that takes a single-quoted string of the form *number* [+-*/] *number* and returns the result of the calculation. The individual numbers do not have leading plus or minus signs.
- `calculate('123 + 27') # => 150`

A Possible Solution</summary>

```
defmodule Parse do

 def calculate(expression) do
   { n1, rest } = parse_number(expression)
   rest         = skip_spaces(rest)
   { op, rest } = parse_operator(rest)
   rest         = skip_spaces(rest)
   { n2, [] }   = parse_number(rest)
   op.(n1, n2)
 end

 defp parse_number(expression), do: _parse_number({ 0, expression })

 defp _parse_number({value, [ digit | rest ] }) when digit in ?0..?9 do
   _parse_number({ value*10 + digit - ?0, rest})
 end

 defp _parse_number(result), do: result


 defp skip_spaces([ ?  | rest ]), do: skip_spaces(rest)
 defp skip_spaces(rest),          do: rest

 defp parse_operator([ ?+ | rest ]), do: { &1+&2, rest }
 defp parse_operator([ ?- | rest ]), do: { &1-&2, rest }
 defp parse_operator([ ?* | rest ]), do: { &1*&2, rest }
 defp parse_operator([ ?/ | rest ]), do: { div(&1, &2), rest }

end

IO.inspect Parse.calculate('23+45')    #=> 68
IO.inspect Parse.calculate('34  - 56')  #=> -22
IO.inspect Parse.calculate('12 * 23')   #=> 276
IO.inspect Parse.calculate('123 / 8')   #=> 15
```

</details>

Exercise: Strings and Binaries-5 (Page 130)

- Write a function that takes a list of dqs and prints each on a separate line, centered in a column which is the width of the longest. Make sure it works with UTF characters.

- iex> center(["cat", "zebra", "elephant"])

  cat

  zebra

  elephant

A Possible Solution</summary>

```
defmodule MyString do

 def center(strings) do
   strings
   |> Enum.map_reduce(0, accumulate_max_length(&1, &2))
   |> center_strings_in_field
   |> Enum.each(IO.puts(&1))
 end

 # We jump through this hoop to avoid calculating the string length twice.
 # Here, we build a tuple containing the length and the string, and
 # nest it in a tuple containing the maximum length seen so far
 defp accumulate_max_length(string, max_length_so_far) do
   l = String.length(string)
   { {string, l}, max(l, max_length_so_far) }
 end

 defp center_strings_in_field({strings, field_width}) do
   strings |> Enum.map(center_one_string(field_width, &1))
 end

 defp center_one_string(field_width, {string, length}) do
   %b[#{String.duplicate(" ", div(field_width - length, 2))}#{string}]
 end
end

MyString.center %w{ cat zebra elephant }
```

</details>

Exercise: Strings and Binaries-6 (Page 131)

- Write a function to capitalize the sentences in a string. Each sentence is terminated by a period and a space. Right now, the case of the characters in the string is random.
- `iex> capitalize_sentences("oh. a DOG. woof. ")`
  `"Oh. A dog. Woof. "`

A Possible Solution</summary>

```
defmodule MyString do

 def capitalize_sentences(string) do
   string
   |> String.split(%r{\.\s+})
   |> Enum.map(String.capitalize(&1))
   |> Enum.join(". ")
 end

end

IO.inspect MyString.capitalize_sentences("oh. a DOG. woof. ")
```

</details>

Exercise: Strings and Binaries-7 (Page 131-132)

- The Lists chapter had an exercise about calculating sales tax. We now have the sales information in a file of comma-separated id, ship_to, and amount values. The file looks like this:
- `id,ship_to,net_amount`
  `123,:NC,100.00`
  `124,:OK,35.50`
  `125,:TX,24.00`
  `126,:TX,44.80`
  `127,:NC,25.00`
  `128,:MA,10.00`
  `129,:CA,102.00`

`120,:NC,50.00`

- Write a function that reads and parses this file, and then passes the result to the sales tax function. Remember that the data should be formatted into a keyword list, and that the fields need to be the correct types (so the id field is an integer, and so on).
- You'll need the library functions **`File.open`**, **`IO.read(file, :line)`**, and **`IO.stream(file)`**.

A Possible Solution</summary>

```
defmodule SimpleCSV do
  def read(filename) do
    file = File.open!(filename)
    headers = read_headers(IO.read(file, :line))
    Enum.map(IO.stream(file), create_one_row(headers, &1))
  end

  defp read_headers(hdr_line) do
    from_csv_and_map(hdr_line, binary_to_atom(&1))
  end

  defp create_one_row(headers, row_csv) do
    row = from_csv_and_map(row_csv, maybe_convert_numbers(&1))
    Enum.zip(headers, row)
  end

  defp from_csv_and_map(row_csv, mapper) do
    row_csv
    |> String.strip
    |> String.split(%r{,\s*})
    |> Enum.map(mapper)
  end

  defp maybe_convert_numbers(value) do
    cond do
      Regex.match?(%r{^\d+$}, value)          -> binary_to_integer(value)
      Regex.match?(%r{^\d+\.\d+$}, value)     -> binary_to_float(value)
      << ?: :: utf8, name :: binary >> = value -> binary_to_atom(name)
      true -> value
    end
  end
end

defmodule Tax do
```

```
  def orders_with_total(orders, tax_rates) do
    orders |> Enum.map(add_total_to(&1, tax_rates))
  end

  def add_total_to(order = [id: _, ship_to: state, net_amount: net], tax_rates) do
    tax_rate = Keyword.get(tax_rates, state, 0)
    tax      = net*tax_rate
    total    = net+tax
    Keyword.put(order, :total_amount, total)
  end

end


tax_rates =  [ NC: 0.075, TX: 0.08 ]

orders = SimpleCSV.read("sales_data.csv")

IO.inspect Tax.orders_with_total(orders, tax_rates)
```

</details>

Chapter 12: Control Flow

Exercise: Control Flow-1 (Page 140)

- **Rewrite the FizzBuzz example using `case`.**

<details>
<summary>A Possible Solution</summary>

```
defmodule FizzBuzz do

  def upto(n) when n > 0 do
    1..n |> Enum.map(fizzbuzz(&1))
  end

  defp fizzbuzz(n) do
    case { rem(n, 3), rem(n, 5), n } do
      { 0, 0, _ } -> "FizzBuzz"
      { 0, _, _ } -> "Fizz"
      { _, 0, _ } -> "Buzz"
```

```
    { _, _, n } -> n
  end
 end
end
```

</details>

Exercise: Control Flow-2 (Page 140-141)

- We now have three different implementations of FizzBuzz. One uses `cond`, one uses `case`, and one uses separate functions with guard clauses.
- Take a minute to look at all three. Which do you feel best expresses the problem. Which will be easiest to maintain?
- The `case` style and the one using guard clauses are somewhat different to control structures in most other lnaguages. If you feel that one of these was the best implementation, can you think of ways of reminding yourself to investigate these options as you write more Elixir code in the future?

Exercise: Control Flow-3 (Page 141)

- Many built-in functions have two forms. The *xxx* form returns the tuple `{:ok, data}` and the *xxx!* form returns data on success but raises an exception otherwise. However, there are some functions that don't have the *xxx!* form.
- Write a function `ok!` takes an arbitary parameter. If the parameter is the tuple `{:ok, data}` return the data. Otherwise raise an exception containing information from the parameter.
- You could use your function like this:
- `file = ok! File.open("somefile")`

A Possible Solution</summary>

```
defmodule MustBe do

 def ok!({:ok, data}), do: data
 def ok!({error_type, error_msg}), do: raise("#{error_type}: #{error_msg}")

end

stream = MustBe.ok!(File.open("/etc/passwd"))
IO.puts(IO.stream(stream) |> Enum.take(5))

try do
 MustBe.ok!(File.open("not-a-file"))
rescue x ->
 IO.puts "ERROR"
 IO.puts x.message
end
```

</details>

Chapter 13: Organizing a Project

Exercise: Organizing a Project-1 (Page 149)

- Do what I did. Honest. Create the project, and write and test the option parser. It's one thing to read about it, but you'll be doing this a lot, so you may as well start now.

Exercise: Organizing a Project-2 (Page 153)

- Add the dependency to your project and install it.

Exercise: Organizing a Project-3 (Page 160)

- Bring your version of this project in line with the code here.

Exercise: Organizing a Project-4 (Page 160)

- (Tricky) Before reading the next section, see if you can write the code to format the data into columns, like the sample output at the start of the chapter. This is

probably the longest piece of Elixir code you'll have written. Try to do it without using `if` or `cond`.

Exercise: Organizing a Project-6 (Page 168)

- In the US, NOAA provides hourly XML feeds[1] of conditions at 1,800 locations. For example, the feed for a small airport close to where I'm writing this is at http://w1.weather.gov/xml/current_obs/KDTO.xml
- .
- Write an application that fetches this data, parses it, and displays it in a nice format.
- (Hint: you might not have to download a library to handle XML parsing)

1. http://w1.weather.gov/xml/current_obs
2. ↩

Chapter 15: Working with Multiple Processes

Exercise: Working with Multiple Processes-1 (Page 206)

- Run this code on your machine. See if you get comparable results.

Exercise: Working with Multiple Processes-2 (Page 206)

- Write a program that spawns two processes, and then passes each a unique token (for example "fred" and "betty"). Have them send the tokens back.
    - Is the order that the replies are received deterministic in theory? In practice?
    - If either answer is no, how could you make it so?

Exercise: Working with Multiple Processes-3 (Page 210)

The Erlang function `timer.sleep(time_in_ms)` suspends the current process for a given time. You might want to use it to force some scenarios in the following:

The key with these exercises is to get used to the different reports that you'll see when you're developing code.

- Use `spawn_link` to start a process, and have that process send a message to the parent and then exit immediately. Meanwhile, sleep for 500ms in the parent, then receive as many messages as there are waiting. Trace what you receive. Does it matter that you weren't waiting for the notification from the child at the time it exited?

Exercise: Working with Multiple Processes-4 (Page 210)

- Do the same, but have the child raise an exception. What difference do you see in the tracing.

Exercise: Working with Multiple Processes-5 (Page 210)

- Repeat the two, changing `spawn_link` to `spawn_monitor`.

Exercise: Working with Multiple Processes-6 (Page 211)

- In the `pmap` code, I assigned the value of `self` to the variable `me` at the top of the method, and then used `me` as the target of the message returned by the spawned processes. Why use a separate variable here?

Exercise: Working with Multiple Processes-7 (Page 211)

- Change the `^pid` in `pmap` to `_pid`. This means that the receive block will take responses in the order the processes send them. Now run the code again. Do

you see any difference in the output? If you're like me, you don't, but the program clearly contains a bug. Are you scared by this? Can you find a way to reveal the problem (perhaps by passing in a different function, or by sleeping, or increasing the number of propcesses)? Change it back to `^pid` and make sure the order is now correct.

Exercise: Working with Multiple Processes-8 (Page 215)

- **Run the Fibonacci code on your machine. Do you get comparable timings. If your machine has multiple cores and/or processors, do you see improvements in the timing as we increase the concurrency of the application?**

Exercise: Working with Multiple Processes-9 (Page 215)

- **Use the same server code, but instead run a function that finds the number of times the word "cat" appears in each file in a given directory. Run one server process per file. The function `File.ls!` returns the names of files in a directory, and `File.read!` reads the contents of a file as a binary.**
- **Run your code on a directory with a reasonable number of files (maybe 100 or so) so you can experiment with the effects of concurrency.**

Chapter 16: Nodes-The Key to Distributing Services

Exercise: Nodes-1 (Page 222)

- Set up two terminal windows, and go to a different directory in each. Then start up a named node in each. Then, in one window, write a function that lists the contents of the current directory.
- `fun = fn -> IO.puts(Enum.join(File.ls!, ",")) end`

- Run it twice, once on each node.

Exercise: Nodes-2 (Page 226)

- **When I introduced the interval server, I said it sent a tick “about every 2 seconds”. But in the receive loop, it has an explicit timeout of 2000mS. Why did I say “about” when it looks as if the time should be pretty accurate?**

Exercise: Nodes-3 (Page 226)

- **Alter the code so that successive ticks are sent to each registered client (so the first goes to the first client, the second the next client, and so on). Once the last client receives a tick, it starts back at the first. The solution should deal with new clients being added at any time.**

Exercise: Nodes-4 (Page 228)

- The ticker process in this chapter is a central server that sends events to registered clients. Reimplement this as a ring of clients. A client sends a tick to the next client in the ring. After 2 seconds, that next client sends a tick to its next client.
- When thinking about how to add clients to the ring, remember to deal with the case where a client's receive loop times out just as you're adding a new process. What does this say about who has to be responsible for updating the links?

Chapter 17: OTP:Servers

Exercise: OTP-Servers-1 (Page 234)

- You're going to start creating a server that implements a stack. The call that initializes your stack will pass in a list that is the initial stack contents.
- For now, only implement the `pop` interface. It's acceptable for your server to crash if someone tries to pop from an empty stack.
- For example, if initialized with `[5,"cat",9]`, successive calls to `pop` will return `5`, `"cat"`, and `9`.

Exercise: OTP-Servers-2 (Page 237)

- Extend your stack server with a `push` interface which adds a single value to the top of the stack. This will be implemented as a cast.

- Experiment in iex with pushing and popping values.

Exercise: OTP-Servers-3 (Page 244)

- Give your stack server process a name, and make sure it is accessible by that name in iex.

Exercise: OTP-Servers-4 (Page 244)

- Add the API to your stack module (the functions that wrap the `gen_server`calls).

Exercise: OTP-Servers-5 (Page 244-245)

- Implement the `terminate` callback in your stack handler. Use `IO.puts` to report the arguments it receives.
- Try various ways of terminating your server. For example, popping an empty stack will raise an exception. You might add code that calls `System.halt(n)` if the `push` handler receives a number less than 10. (This will let you generate different return codes). Use your imagination to try different scenarios.

Chapter 18: OTP:Supervisors

Exercise: OTP-Supervisors-1 (Page 250)

- **Add a supervisor to your stack application. Use iex to make sure it starts the server correctly. Use the server normally, and then crash it (try popping from an empty stack). Did it restart? What was the stack contents after the restart?**

Exercise: OTP-Supervisors-2 (Page 254)

- **Rework your stack server to use a supervision tree with a separate stash process to hold the state. Verify it works, and that when you crash the server the state is retained across a restart.**

Chapter 20: OTP:Applications

Exercise: OTP-Applications-1 (Page 282)

- **Turn your stack server into an OTP application.**

Exercise: OTP-Applications-2 (Page 282)

- **So far, we haven't written any tests for the application. Is there anything you can test? See what you can do.**

Exercise: OTP-Applications-3 (Page 292)

Our boss notices that after we applied our version-0-to-version-1 code change, the delta indeed works as specified. However, she also notices that if the server crashes,

the delta is forgotten—only the current number is retained. Create a new release that stashes both values.

Chapter 22: Macros and Code Evaluation

Exercise: Macros and Code Evaluation-1 (Page 311)

- Write a macro called `myunless` that implements the standard *unless*functionality. You're allowed to use the regular `if` expresssion in it.

Exercise: Macros and Code Evaluation-2 (Page 311)

- Write a macro called `times_n` that takes a single numeric argument. It should define a function in the module of the caller that itself takes a single argument, and which multiplies that argument by *n*. The new function should be called `times_`*n*. So, calling `times_n(3)` should create a function called `times_3`, and calling `times_3(4)` should return 12. Here's an example of it in use:
- ```
  defmodule Test do
    require Times
    Times.times_n(3)
    Times.times_n(4)
  end

  IO.puts Test.times_3(4)   #=> 12
  IO.puts Test.times_4(5)   #=> 20
  ```
  ```

Exercise: Macros and Code Evaluation-3 (Page 317)

- The Elixir test framework, ExUnit, uses some clever code quoting tricks. For example, if you assert
- `assert 5 < 4`

- You'll get the error "expected 5 to be less than 4."
- The Elixir source code is on Github (at
- **https://github.com/elixir-lang/elixir**
-). The implementation of this is in the file **`/lib/ex_unit/lib/assertions.ex`**. Spend some time reading this file, and work out how it implements this trick.
- (Hard) Once you're done that, see if you can use the same technique to implement a function that takes an arbitrary arithmetic expression and returns a natural language version.
- ```
  explain do: 2 + 3*4
  #=> multiply 3 and 4, then add 2
  ```

Chapter 23: Linking Modules: Behavio(u)rs and use

Exercise: Linking Modules-Behaviours and Use-1 (Page 326)

- In the body of the **`def`** macro, there's a quote block that defines the actual method. It contains:
- ```
  IO.puts "==> call:   #{Tracer.dump_definition(unquote(name), unquote(args))}"
  result = unquote(content)
  IO.puts "<== result: #{result}"
  ```

- Why does the first call to `puts` have to unquote the values in its interpolation, but the second call does not?

Exercise: Linking Modules-Behaviours and Use-2 (Page 326-327)

- The built-in function `IO.ANSI.escape` will insert ANSI escape sequences in a string. If you put the resulting strings to a terminal, you can add colors and bold or underlined text. Explore the library, and then use it to colorize the output of our tracing.

Exercise: Linking Modules-Behaviours and Use-3 (Page 327)

- (Hard). Try adding a method definition with a guard clause to the `Test` module. You'll find that the tracing now longer works.
    - Find out why
    - See if you can fix it
- (You may need to explore `Kernel.def/4`)

Chapter 24: Protocols-Polymorphic Functions

Exercise: Protocols-1 (Page 332)

- A basic Caesar cypher consists of shifting the letters is a message by a fixed offset. For an offset of 1, for example, a will become b, b will become c, and z will become a. If the offset is 13, we have the ROT13 algorithm.

- Lists and binaries can both be *string-like*. Write a `Caesar` protocol that applies to both. It would include two functions: `encrypt(string, shift)` and `rot13(string)`.

Exercise: Protocols-2 (Page 332)

- **Use a list of words in your language to look for words where `rot13(word)` is also a word in the list. For various types of English word list, have a look at http://wordlist.sourceforge.net/**
- **. The SCOWL collection looks promising, as it already has words divided by size.**

Exercise: Protocols-3 (Page 345)

- Collections that implement the `Enumerable` protocol define `count`,`member?`, and `reduce` functions. The `Enum` module uses these to implement methods such as `each`, `filter`, and `map`.
- Implement your own versions of `each`, `filter`, and `map` in terms of `reduce`.

Exercise: Protocols-4 (Page 345)

- In many cases, `inspect` will return a valid Elixir literal for the value being inspected. Update the `inspect` function for records so that it returns valid Elixir code to contruct a new record equal to the value being inspected.

Chapter 25: More Cool Stuff

Exercise: More Cool Stuff-1 (Page 350)

- Write a sigil `%s` that parses multiple lines of comma-separated data, returning a list where each element is a row of data, and each row is a list of values. Don't worry about quoting—just assume that each field is separated by a comma. So
- ```
  csv = %c"""
  1,2,3
  cat,dog
  """
  ```

- Would generate `[["1","2","3"], ["cat","dog"]]`

Exercise: More Cool Stuff-2 (Page 350)

- The function `String.to_float` converts a string to either a float or an integer, returning `:error` if the string was not a valid number.
  ```

- Update your CSV sigil so that numbers are automatically converted:
- ```
  csv = %c"""
  1,2,3.14
  cat,dog
  """
  ```
- Would generate `[[1,2,3.14], ["cat","dog"]]`

Exercise: More Cool Stuff-3 (Page 350-351)

- (Harder) Sometimes the first line of a CSV file is a list of the column names. Update your code to support this, and return the values in each row as a keyword list using the column names as the keys.
- ```
  csv = %c"""
  Item,Qty,Price
  Teddy bear,4,34.95
  Milk,1,2.99
  Battery,6,8.00
  """
  ```
- Would generate:
- ```
  [
    [Item: "Teddy bear", Qty: 4, Price: 34.95],
    [Item: "Milk", Qty: 1, Price: 2.99],
    [Item: "Battery", Qty: 6, Price: 8.00]
  ]
  ```
  ```