CSCI 1103 Computer Science 1 Honors

Fall 2020

Robert Muller - Boston College

DRAFT Lecture Notes

Week 3

Topics:

1. Sum Types & Branching with Match Expressions
2. Lists & Repetive Functions
3. Work

---

1. Sum Types & Branching with Match Expressions

Last week we introduced the first of several built-in structured types in OCaml, the product type. This week we're going to focus on the dual of product types -- *sum types*.

> Heads up! Sum types are known by several (too many) different names, including *union types*, *disjoint unions* and *tagged unions*. They are sometimes referred to as *variant types*, *either/or types* or even *enum* or *enumeration types*.
>
> In universal algebra they are known as *co-products* because they are dual to *product types*. That's another story for another class.

Here is a simple example.

```
# type coin = Heads | Tails;;
type coin = Heads | Tails;;

# Heads;;
- : coin = Heads
```

The notation `type coin = Heads | Tails` defines a new type *coin*. There are exactly two sorts of coins, those that are `Heads` and those that are `Tails`. There are no other coins, we've explicitly enumerated them all. The vertical bar `|` is pronounce "or" (hence the name either/or). The symbols `Heads` and `Tails` are often referred to as *variants* or *constructors*. In OCaml, user-defined variants must begin with a capital letter.

A value of sum type can be used in a *match expression* to dispatch to different pieces of code.

```
# let myCoin : coin = ... an expression yielding one of Heads or Tails ...

# match myCoin with | Heads -> 0 | Tails -> 1;;
- : int = ... 0 or 1 depending on the value of myCoin ...
```

Like `let` and `in`, the symbols `match` and `with` are keywords in OCaml. We'll call the expression occurring between the `match` and `with` keywords the *dispatch expression*, in this case the dispatch expression is the simple variable `myCoin`. In this class, dispatch expressions will always be of sum type. The phrases `| Heads -> 0` and `| Tails -> 1` are clauses for the various possible values of the dispatch expression. If the value of `myCoin` is `Heads` then the value of the entire match expression is `0`. If the value of `myCoin` is `Tails` then the value of the entire match expression is `1`.

The Special built-in Sum Type `bool`

OCaml has 3 important built-in sum types. We'll see 2 of them this week and the 3rd in a few weeks. The first is the built-in type `bool`.

```
type bool = true | false
```

The variants `true` and `false` are the only values of type `bool` .

> History: The name "bool" is a contraction of the surname of [George Boole](), a 19th century British mathematician. Boole helped establish formal logic, explaining his system in an immodestly titled book [The Laws of Thought]().

> Heads up!
>
> 1. The boolean variants `true` and `false` are the only variants in OCaml that start with a lower case letter.
> 2. You may be familiar with booleans in other programming languages. In Java and JavaScript the name of the type is `boolean` and the variants are as in OCaml. In Python `True` and `False` are values of type `bool`. Don't tell the Java, JavaScript or Python programmers that booleans are sum types, it's a secret!

The if-expression

The special case of the `bool` sum type can be used to dispatch in an *if-expression*. The form is

```
if boolExpr then trueExpr else falseExpr
```

The symbols `if`, `then` and `else` are all OCaml keywords. The `boolExpr` is some expression of type `bool`, i.e., one that yields one or the other of the two variants `true` or `false`. If the value `boolExpr` is `true` then the value of `trueExpr` is the value of the whole if-expression. If the value of `boolExpr` is `false` then the value of `falseExpr` is the value of the whole if-expression. Note that `trueExpr` and `falseExpr` must be of the same type.

```
# let myBool : bool = ... some expression yielding one of true or false ...
# if myBool then 0 else 1
```

Pervasive Logical Operators Involving the sum type `bool`

In OCaml there are a variety of pervasive comparison operators yielding boolean values.

```
# 2 < 3;;
- : bool = true

# 2 = 3;;
- : bool = false

# 2 != 3;;
- : bool = true

# 2 <= 3;;
- : bool = true

# let isEven n = (n mod 2) = 0;;
val isEven : int -> bool
```

Pervasive Special Forms Involving the sum type `bool`

OCaml provides the usual complement of "special forms" for working with booleans including *logical and* `&&`, *logical or* `||` and *logical not* `not`.

```
# true && false;;
- : bool = false

# true || false;;
- : bool = true

# not true;;
- : bool = false

# not false;;
- : bool = true
```

The forms `&&` and `||` are "special", i.e., they are not run-of-the-mill binary operators, because they use so-called *short-circuit evaluation*. That is, these forms don't always evaluate both operands. In particular, given

```
# expr1 && expr2;;
```

If the value of `expr1` is `false` then there is no need to evaluate `expr2`, the value of the whole and expression is `false`. Dually, given

```
# expr1 || expr2;;
```

If the value of `expr1` is `true` then there is no need to evaluate `expr2`, the value of the whole or expression is `true`. Short circuit evaluation is pretty much universal across all programming languages. It is used idiomatically in situations like this

```
# let divisor = 0;;
val divisor : int = 0

# (divisor != 0) && (4 / divisor = 6);;
```

In the above, short-circuit evaluations allows us to avoid the run-time error of attempting to divide by 0.

Match Expressions

The if-expression is a special case of the more general and more powerful *match-expression*. The latter is more widely used. When the dispatch expression is of the particular type `bool`, the `if` expression can be more concise than the `match` expression

```
# if true then 0 else 1;;
- : int = 0

# match true with | true -> 0 | false -> 1;;
- : int = 0

# if 2 = 3 then Heads else Tails;;
- : coin = Tails

# match 2 = 3 with | true -> Heads | false -> Tails;;
- : coin = Tails
```

Simplification of Match Expressions

Match expressions are simplified by first simplifying the dispatch expression. The result should be a value of some sum type. The case clauses should handle all of the variants of the given sum type. For example, using the Standard Library `Random.int` function to generate a random integer 0 or 1, and using a double arrow `=>` to denote a simplification step, we have

```
match (Random.int 2) = 0 with | true -> Heads | false -> Tails =>
  match 1 = 0 with | true -> Heads | false -> Tails =>
  match false with | true -> Heads | false -> Tails =>
  Tails
```

We can package this up in a function as follows.

```
(* flip : unit -> coin
 *)
let flip () =
  match Random.int 2 = 0 with              (* A random integer in the range 0..1
*)
   | true  -> Heads
   | false -> Tails
```

Note the formatting, the vertical lines (again pronounced "or") are lined up directly beneath the `m` in `match`.

```
type fruit = Durian | Lemon | Mangosteen | Orange | Lychee

# Lemon;;
- : fruit = Lemon

(* isCitrus : fruit -> bool
```

```
 *)
let isCitrus fruit =
  match fruit with
  | Durian -> false
  | Lemon -> true
  | Mangosteen -> false
  | Orange -> true
  | Lychee -> false
```

> Heads up! In the snippet above we're recycling the name `fruit`, using it both to name a new type and for a variable that will take on a value of that type. This is fine.

In writing a match expression, if we don't have a clause for each of the variants of the type of the dispatch expression, OCaml (and/or OCaml's Merlin editor support) will warn us that we might have forgotten something.

```
(* isCitrus : fruit -> bool           forgetting to handle Lychee ...
 *)
let isCitrus fruit =
  match fruit with
  | Durian -> false
  | Lemon -> true
  | Mangosteen -> false
  | Orange -> true;;

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Lychee
```

This feature is exceedingly helpful for software developers, in the absence of this feature, developers can spend days trying to track down omitted-case bugs.

Match clauses can be written a little more concisely in many cases.

```
(* isCitrus : fruit -> bool
 *)
let isCitrus fruit =
  match fruit with
  | Lemon | Orange -> true            (* NB: Two variants with the same outcome.
*)
  | _ -> false                        (* NB: _ is a wildcard, it matches anything.
*)
```

Information Carrying Variants

Some constructors would naturally require some sort of input to build their values. We'll look at two examples.

Example: Shapes

A circle can be understood given only its radius, a rectangle requires a width and height.

```
# type shape = Circle of float
             | Rectangle of float * float;;
```

The snippet `Circle of float` tells OCaml that the variant/constructor named `Circle` requires an input value of type `float` to produce a value of type `shape`. The snippet `Rectangle of float * float` tells OCaml that the variant/constructor named `Rectangle` requires a pair of `float`s to produce a value of type `shape`. It's useful to think of the variant `Circle` as a *function* from `float` to the new type `shape` and likewise for the variant `Rectangle`.

> Heads up! A new type like `shape` is usually written with one line per variant. The stick should be lined up directly beneath the = sign.

```
# let myCircle = Circle 1.0;;
val myCircle : shape = Circle 1.0

# let myRectangle = Rectangle (2.0, 4.0);;
val myRectangle : shape = Rectangle(2.0, 4.0)
```

The match expression makes it easy to work with values of type `shape`.

```
(* area : shape -> float
 *)
let area shape =
  match shape with
  | Circle radius -> Code.pi *. radius ** 2.0
  | Rectangle (width, height) -> width *. height
```

Notice that in the pattern `Circle radius` we've used the variable `radius` to match against whatever floating point number was used in constructing the circle and in the pattern `Rectangle (width, height)` we've used variables `width` and `height` similarly. This is helpful for the reader of the `shape` function. Unfortunately, when they laid eyes on the definition of the type `shape`, they learned that these three items were of type `float` but they didn't know from the definition that one was used to denote a radius, and the other two to denote width and height. We'll soon introduce another structured type -- the super-useful *record type*, which will provide these symbolic names in the definitions of the types.

Example: Numbers

The programming language JavaScript doesn't have a type `float` or `int`, it has only a combination type `number`. So both `3.14` and `3` are of type `number` in JavaScript. Sometimes this is useful. But what if we need to, say, double a number in OCaml? If the number is an `int` we need `number * 2`, if the number is a `float` we need `number *. 2.0`.

In OCaml we would manage this by introducing a new sum type `number` as follows.

```
# type number = Int of int | Flt of float;;
type number = Int of int | Flt of float

# Int 4;;
- : number = Int 4

# Flt 4.0;;
- : number = Flt 4.0
```

The snippet `Int of int` tells OCaml that the variant/constructor named `Int` requires an input value of type `int` to produce a value of type `number`. And likewise for `Flt`, it needs an input value of type `float`.

Now we can write our doubling function.

```
type number = Int of int | Flt of float

(* double : number -> number
 *)
let double number =
  match number with
  | Int n -> Int (n * 2)
  | Flt n -> Flt (n *. 2.0)
```

2. Lists & Repetive Functions

Earlier we alluded to 3 important built-in sum types and we saw the first of these, the sum type `bool`. In this section, we'll discuss the 2nd major built-in sum type, the type of *lists*. The basic idea is quite simple: a list is one of two things: it is either 1. empty or 2. non-empty. An empty list is written with adjacent square brackets `[]`, pronounced "nil" while a non-empty list can be written as `x :: xs`, pronounced "x cons xs", i.e., the symbol `::` is pronounced "cons". The item `x` is a list element of some kind and `xs` is a list containing elements of the same kind.

```
# [];;                     (* The empty list, called "nil" *)
- : 'a list = []

# 2 :: [];;                (* The cons operator ::, given a list element and a list,
*)
- : int list = [2]         (* it constructs a new list. *)

# 2 :: 4 :: [];;
- : int list = [2; 4]

# 2 :: (4 :: []);;
- : int list = [2; 4]
```

> Heads up! In OCaml, when a list has several elements they are separated by semi-colons rather than the more common comma. In Python and Javascript one would type a list as `[1, 2, 3]` with the elements separated by commas. But it turns out that lists in Python and JavaScript actually correspond to *arrays* in OCaml (and Java). We'll come back to this important topic later.

```
# 1 :: (1 + 2) :: (2 + 3) :: [];;
- : int list = [1; 3; 5]

# let odds = [1; 1 + 2; 2 + 3];;
val odds : int list = [1; 3; 5]
```

Like the built-in sum type `bool`, the built-in list type comes with special syntax using square brackets.

The Standard Library has a `List` module with many useful functions on lists.

```
# List.hd odds;;                          (* hd for "head" *)
- : int = 1

# List.tl odds;;                          (* tl for "tail" *)
- : int list = [3; 5]

# List.length odds;;
- : int = 3

# List.mem 3 odds;;
- : bool = true

# List.rev odds;;
- : int list = [5; 3; 1]

# odds;;
```

```
- : int list = [1; 3; 5]                 (* odds wasn't changed/mutated by
List.rev *)

# let fiveOdds = odds @ [7; 9];;         (* The append operator @ aka List.append
*)
val fiveOdds = [1; 3; 5; 7; 9]

# odds;;
- : int list = [1; 3; 5]                  (* odds wasn't changed/mutated by append
*)

# let lincoln = ["Four"; "score"; "and"; "seven"; "years"; "ago"]
val lincoln : string list = ["Four"; "score"; "and"; "seven"; "years"; "ago"]

# Code.explode "Four";;
- : char list = ['F'; 'o'; 'u'; 'r']

# Code.implode ['s'; 'c'; 'o'; 'r'; 'e'];;
- : string = "score"

# type fruit = Durian | Lemon | Mangosteen | Orange | Lychee;;

# let fruits = [Lemon; Durian; Lychee; Lemon];;
val fruits : fruit list = [Lemon; Durian; Lychee; Lemon];;
```

Lists can contain values of any type but values of mixed types within a list aren't allowed.

```
# [2; 2.5];;
Error: This expression has type float but an expression was expected of type
         int
```

We'll return to this issue below.

Functions on Lists

Let `xs` be a list and let `f` be a function accepting `xs`:

```
let f xs = ...
```

A list can have zero elements or some number of elements that might vary from one call of `f` to another. This means that most functions working on lists involve *repetition* — some number of computation steps need to be taken for each element of `xs`. It is a touchstone of OCaml and functional programming languages in general (typed or not) that this sort of repetition is implemented by writing `f` as a *recursive function*, i.e., one that can refer to itself in its own definition.

Recursive functions are quite powerful and fun to write. When they're writen well, they can be quite elegant while still being very efficient. In OCaml, the keyword `rec` is required to distinguish a recursive function definition from the definition of a non-recursive function:

```
let rec f xs = ...
```

Example: List.length `(length xs)` returns the integer length of list `xs`.

We'll start with a simple function that determines the length of a list. There is a built-in function for this purpose, `List.length` but we'll write it ourselves. Given a call `(length xs)` the idea is to identify the simplest case first, in this case when `xs` is `[]`. Such a case is usually called a *base case*. Clearly the length of the empty list is 0. The non-base case, the *recursive case*, is when `xs` is a cons `y :: ys`. This is called the recursive case because the type of `ys` is the same as the type of `xs` — a list of items, but with one fewer elements. **To determine the number of elements in `xs` it suffices to determine the number of elements in `ys` and then add 1**. We're invited to use the `length` function that we're presently writing.

```
(* length 'a list -> int
*)
let rec length xs =
  match xs with
  | [] -> 0
  | y :: ys -> 1 + length ys
```

The form above with the body of the function being a match expression is completely standard. Tracing a call `length [2; 4]` yields:

```
length [2; 4] =                (* NB: this step isn't a run-time computation step
*)
  length (2 :: [4]) =>
  match (2 :: [4]) with | [] -> 0 | y :: ys -> 1 + length ys =>
  1 + length [4] =
  1 + length (4 :: []) =>
  1 + (match (4 :: []) with | [] -> 0 | y :: ys -> 1 + length ys) =>
  1 + (1 + (length [])) =>
  1 + (1 + (match [] with | [] -> 0 | y :: ys -> 1 + length ys)) =>
  1 + (1 + 0) =>
  1 + 1 =>
  2
```

It's worth noting that since the variable `y` is never used, it might be replaced with the don't-care variable `_`.

```
(* length 'a list -> int
*)
let rec length xs =
  match xs with
  | [] -> 0
  | _ :: ys -> 1 + length ys
```

Example: List.mem `(mem x xs)` returns `true` if `x` is an element of `xs`. Otherwise it returns `false`.

```
(* mem : 'a -> 'a list -> bool
*)
let rec mem x xs =
  match xs with
  | [] -> false
  | y :: ys -> (match x = y with
               | true  -> true
               | false -> mem x ys)
```

> Heads up! There is a bit of a fast one being played here. We've left it to OCaml to infer the types. Fair enough. It reports that our code is well-defined and of polymorphic type `'a -> 'a list -> bool`. This means that it should work for *any* type that we might plug in for the type variable `'a`. But inspecting the code, we see that the inputs must be of a type that is acceptable to the equality operator `=`. It turns out that not all types are comparable. We're going to let OCaml slide on this one for now.

We might consider rewriting the `mem` function a little more concisely using the `||` form.

```
(* mem : 'a -> 'a list -> bool
*)
let rec mem x xs =
  match xs with
  | [] -> false
  | y :: ys -> (x = y) || (mem x ys)
```

We could go further still as follows but this third version is maybe too terse, placing the reader at a disadvantage.

```
(* mem : 'a -> 'a list -> bool
*)
let rec mem x xs = (xs != []) && ((x = List.hd xs) || (mem x (List.tl xs)))
```

> Heads up! It's worth noting that the combination of the test `(xs != [])` and short-circuit evaluation of the logical forms prevent possible errors, e.g., attempting to take `(List.hd [])`, and allows the function to terminate. In logic, the forms `&&` and `||` are commutative operators, i.e., `P && Q` is the same as `Q && P` (and likewise for `||`). But the short-circuiting versions of these forms used in programming languages are not commutative.

Example: addList `(addList ns)` returns the sum of the integers in list `ns`.

```
(* addList : int list -> int
 *)
let rec addList ns =
  match ns with
  | [] -> 0
  | m :: ms -> m + addList ms
```

This one is worth tracing.

```
addList [1; 2; 3] =
  addList (1 :: [2; 3]) =>
  match (1 :: [2; 3]) with | [] -> 0 | m :: ms -> m + addList ms =>
  1 + addList [2; 3] =
  1 + addList (2 :: [3]) =>
  1 + (match (2 :: [3]) with | [] -> 0 | m :: ms -> m + addList ms) =>
  1 + (2 + addList [3]) =
  1 + (2 + addList (3 :: [])) =>
  1 + (2 + (match (3 :: []) with | [] -> 0 | m :: ms -> m + addList ms)) =>
  1 + (2 + (3 + addList [])) =>
  1 + (2 + (3 + (match [] with | [] -> 0 | m :: ms -> m + addList ms)) =>
  1 + (2 + (3 + 0)) =>
  1 + (2 + 3) =>
  1 + 5 =>
  6
```

Here's another version, this one requires the caller to provide an initial answer as in `(addList ns 0)`.

```
(* addList : int list -> int -> int
 *)
let rec addList ns answer =
  match ns with
  | [] -> answer
  | m :: ms -> addList ms (m + answer)
```

```
addList [1; 2; 3] 0 =
  addList (1 :: [2; 3]) 0 =>
  match (1 :: [2; 3]) with | [] -> 0 | m :: ms -> addList ms (m + 0) =>
  addList [2; 3] (1 + 0) =
  addList (2 :: [3]) (1 + 0) =>
  addList (2 :: [3]) 1 =>
  match (2 :: [3]) with | [] -> 1 | m :: ms -> addList ms (m + 1) =>
  addList [3] (2 + 1) =
  addList (3 :: []) (2 + 1) =>
  addList (3 :: []) 3 =>
  match (3 :: []) with | [] -> 3 | m :: ms -> addList ms (m + 3) =>
  addList [] (3 + 3) =>
  addList [] 6 =>
  match [] with | [] -> 6 | m :: ms -> addList ms (m + 6) =>
  6
```

We'll come back to this topic in a bit. For now, it's worth noting that the first version of `addList` performed its additions *on the way out* while the second performed its additions *on the way in*. It turns out the latter is more efficient.

Example: List.append `(append xs ys)` returns the list resulting from appending xs to ys.

```
(* append : 'a list -> 'a list -> 'a list
 *)
let rec append xs ys =
  match xs with
  | [] -> ys
  | z :: zs -> z :: append zs ys
```

It's worth stepping through this one.

```
append [3; 5] [7; 9; 11] =
  append (3 :: [5]) [7; 9; 11] =>
  match (3 :: [5]) with | [] -> [7; 9; 11] | z :: zs -> z :: append zs [7; 9; 11]
=>
  3 :: (append [5] [7; 9; 11]) =
  3 :: (append (5 :: []) [7; 9; 11]) =>
  3 :: (match (5 :: []) with | [] -> [7; 9; 11] | z :: zs -> z :: append zs [7;
9;11]) =>
  3 :: (5 :: (append [] [7; 9; 11])) =>
  3 :: (5 :: (match [] with | [] -> [7; 9; 11] | z :: zs -> z :: append zs [7; 9;
11]) =>
  3 :: (5 :: [7; 9; 11]) =
  [3; 5; 7; 9; 11]
```

Example: List.rev `(rev xs)` returns the reverse of xs.

```
(* rev : 'a list -> 'a list
 *)
let rec rev xs =
  match xs with
  | [] -> []
  | z :: zs -> append (rev zs) [z]
```

This is a reasonably straightforward answer but it turns out to be very problematic. Read on!

Challenge: addNumbers `(addNumbers ns)` Revisiting the problem of heterogenous list elements, can you write a function that yields the floating point sum of a list of numbers?

```
(* addNumbers : number list -> float
 *)
let rec addNumbers ns =
  match ns with
  | [] -> 0.0
  | (Int m) :: ms -> (float m) +. addNumbers ms
  | (Flt m) :: ms ->         m  +. addNumbers ms
```

---

3. Work

When writing repetitive functions, we should always be mindful of the amount of work each function requires. Let's revisit the simple examples above with an eye toward sorting out their performance properties. For now we'll focus on the number of computation steps required. This corresponds to the amount of time taken as well as the amount of energy consumed. We should also be concerned with the amount of computer storage required. We'll defer consideration of this latter aspect for a few weeks.

How Much Work does the `length` function do?

Inspecting the definition of `length` and the example trace of `(length [2; 4])`, we count 8 simplification steps, the first `=>` is a *call step*, the second is a *dispatch step*, later on there is an *addition step*. The list `[2; 4]` has 2 elements but lets say in general the input list `xs` has N elements. For each of the N elements there will be a call, a dispatch and an addition step. This is $3N$ steps. When the list is empty there will be one call step followed by one dispatch step, i.e., 2 steps. So then for any input list `xs` with N elements, the total number of steps required is $3N + 2$.

When considering resource consumption, we're generally interested in resource consumption for large inputs, i.e., large values of N. Obviously the 2 is irrelevant. But it turns out, so is the factor 3. As N grows large, the constant factor isn't contributing much. So for all intents and purposes, this function consumes N resource units (time, energy, e.g.,) for inputs of size N. I.e., the resource consumption is said to grow *linearly* with the input size.

How Much Work does the `mem` function do?

Let's say the input list `xs` has N elements. If `x` isn't an element of `xs`, the `mem` function will confirm this by inspecting all N elements of `xs`. So we're inclined to view `mem`'s work as being linear. On the other hand, if `x` happens to be the first element of `xs`, then `mem` will return `true` after one step. So unlike `length` which depended only on the number of elements in `xs`, the amount of work performed by the `mem` function depends on whether or not `x` appears in the list, and if it does, where it sits.

The amount of work a function performs in the *worst case* might be different from how much work it might perform in the *average case*. For `List.length` the worst case and the average case are the same. But for many functions the worst case performance is quite bad while the average case performance is quite good.

> Example: the problem of inferring types turns out to be stupendously expensive in the worst case but is super-fast on average!

Average case analysis is more involved, requiring consideration of the probabilities of different arrangements of elements in `xs`. This important topic is taken up in detail in our 2000-level *Randomness and Computation* and 3000-level *Algorithms* courses.

How Much Work does the `addList` function do?

A quick inspection of the code shows that it has precisely the same analysis as the `length` function, the resource consumption is linear in the length of `ns`.

How Much Work does the `append` function do?

An inspection of the above trace of `(append xs ys)` shows that the amount of work performed is linear in the length of `xs`. The number of elements in `ys` is irrelevant.

How Much Work does the `rev` function do?

Consider the following brief trace of a call `(rev [1; 2; 3])`.

```
rev [1; 2; 3] =
  rev (1 :: [2; 3]) =>
  match (1 :: [2; 3]) with | [] -> [] | z :: zs -> append (rev zs) [z] =>
  append (rev [2; 3]) [1] =
  append (rev (2 :: [3])) [1] =>
  append (match (2 :: [3]) with | [] -> [] | z :: zs -> append (rev zs) [z]) [1]
=>
  append (append (rev [3]) [2]) [1] =
  append (append (rev (3 :: [])) [2]) [1] =>
  append (append (match 3 :: [] wth | [] -> | z :: zs -> append (rev zs) [z]) [2])
[1] =>
  append (append (append (rev []) [3]) [2]) [1] =>
  append (append (append (match [] with | [] -> [] | ...) [3]) [2]) [1] =>
  append (append (append [] [3]) [2]) [1] =>
  ...
```

We see that for each of the N items in the input list to `rev`, we've stacked up one call of the `append` function.

```
append (append (append (append ... [4]) [3]) [2]) [1]
```

The rightmost (mostly deeply nested) call of `append` receives as its first argument a list of length 0. That call produces a list of length 1, the next call of `append` receives that list of length 1 (as its first argument) and returns a list of length 2, and so on. We've already established the work requirements of `(append xs ys)`, if `xs` contains N elements, `append` will require N units of work. So we see that `(rev xs)` requires

$$1 + 2 + \ldots + (N - 1) = \frac{N(N-1)}{2}$$

steps. This is $\frac{1}{2}(N^2 - N)$. As before, we can ignore the constant factor $1/2$. It turns out that we can ignore the linear term N too. So the amount of work performed by this simple version of `rev` is N^2, i.e., it is *quadratic* in the size of the input. If N is 10, then the amount of work is $10^2 = 100$, if N is 100, then the amount of work is $10^4 = 10000$. Our computers are fast (!), so our `rev` function seems fine for small inputs. But when N is 10^6, the work required is 10^{12}. This is not nothing. And if we were considering using our simple `rev` function to reverse a list of the 3 billion ($3 \cdot 10^9$) base pairs in the human genome, well, we would be looking at roughly ${10^9}^2 = 10^{18}$ steps. This is big but, again, our computers are fast. Using a back-of-the-envelop calculation, let's say our computer can perform one billion (i.e., 10^9) steps in one second. Since $\frac{10^{18}}{10^9} = 10^9$, our calculation will require 10^9 seconds, roughly 31 years.

Obviously we must do better.