1 Append

Recall that OCaml provides a built-in list append operator, @. That is, [17;18;22] @ [53;32] evaluates to [17;18;22;53;32]. Lists are still lists, and OCaml is presumably doing nothing magical.

1.1 Implementation

Quiz: Write append: 'a list * 'a list -> 'a list without using @

- input: two lists
- output: the list consisting of the items in the first input list followed by the items in the second input list

let rec append : 'a list * 'a list -> 'a list = function
| [], lst -> lst
| first::rest, lst -> first::(append(rest, lst)) ;;
1.2 Analysis

What is the runtime of this procedure? Let \( f(m, n) \) be worst-case time for appending an \( m \)-element list and an \( n \)-element list.

A note about the analysis of OCaml procedures:

- pattern matching takes constant time (assuming no “fancy” pattern-matching stuff)
- Usual stuff (consing, procedure application) takes constant time

Recurrence relation:

\[
\begin{align*}
    f(0, n) & \leq B \\
    f(m, n) & \leq A + f(m - 1, n) \quad \text{when } m > 0
\end{align*}
\]

If we plug and chug this recurrence relation out into a closed form solution, we get:

\[
f(m, n) \leq Am + B
\]

Thus, our append procedure runs in \( O(m) \), or linear, time.

2 Reverse

Last time at the end of lecture, we had a quiz during which we asked you to write the reverse procedure. Let’s do that again.

2.1 Implementation

**Quiz:** Write reverse: `’a list -> ’a list`. You are allowed to use the @ operator.

Reverse procedure using list appending operator:

```ocaml
let rec reverse : ’a list -> ’a list = function
  | [] -> []
  | first::rest -> (reverse rest) @ [first] ;;
```

This is a simple and clear solution, but what about the running time?

2.2 Analysis

For appending, we should use the time we computed.

Let \( g(n) \) be reverse’s worst-case running time for lists of length \( n \).

Recurrence relation:

\[
\begin{align*}
    g(0) & \leq C \\
    g(n) & \leq D + f(n - 1, 1) + g(n - 1) \quad \text{when } n > 0
\end{align*}
\]
Using plug and chug, we can derive the following closed form solution (the last line being the most simplified version):

\[
\begin{align*}
g(n) & \leq D + (A(n - 1) + B) + g(n - 1) \quad (6) \\
g(n) & \leq Dn + n(A(n - 1) + B) + C \quad (7) \\
g(n) & \leq Dn + An^2 - An + Bn + C \quad (8)
\end{align*}
\]

This means that \( g(n) \) is \( O(n^2) \). In fact, it is also \( \Omega(n^2) \) and \( \Theta(n^2) \), meaning that the worst case run time is also at least of the order \( n^2 \). \( \Theta \) just means that a procedure is both in \( O \) and \( \Omega \) of a function.

Can we do better? Can we achieve linear time for reversing a list?

3 Usual strategy for recursion

We have encouraged you to derive recursive solutions to given problems using a kind of strategy that is captured by recursion diagrams: derive the recursive input from the original input in a straightforward way, then derive the original output from the recursive output in some way that depends on the given problem.

We have asked you to do this for problems where we know there is a nice solution! One nice problem, and one you might need to solve if you are programming Connect 4, is diagonals:

- **input**: a matrix, represented as an `a list list`, where each item in the list is a list representing a row of the matrix
- **output**: a list consisting of the diagonals, where each diagonal is represented as an `a list` of elements of the original matrix

**Example:**

Consider the matrix

\[
\begin{bmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12
\end{bmatrix}
\]

The diagonals are

\[
\begin{align*}
[1] \\
[2; 5] \\
[3; 6; 9] \\
[4; 7; 10] \\
[8; 11] \\
[12]
\end{align*}
\]

Thus, diagonals \([ [1; 2; 3; 4]; [5; 6; 7; 8]; [9; 10; 11; 12] ] \) should evaluate to \([ [1]; [2; 5]; [3; 6; 9]; [4; 7; 10]; [8; 11]; [12] ] \)

We wrote this using helper procedures `take` and `drop` and another helper procedure.
4 Introduction to tail recursion

Sometimes you need to be a bit more creative in deriving recursive solutions, whether because it’s not clear how to derive the original output from the recursive output or because it’s not clear how to do so very efficiently.

One different strategy is called tail recursion. We have delayed talking about this strategy because we feared that students, especially those blessed/cursed with prior nonfunctional programming experience, would glom onto this to the exclusion of the more canonical (and often more beautiful) recursive approach.

But here goes... A tail-recursive procedure is a recursive procedure in which the original output is derived from the recursive output by making no change. That is, the original output must equal the recursive output.

For most problems, this cannot possibly work if we derive the recursive input from the original input in the usual way, e.g. cdr. It requires changing the problem.

One relatively simple change is adding an additional argument that binds to a partial solution. The procedure takes a regular argument and a partial-solution argument. The guiding idea is that the value to return in the base case is the value of that partial-solution argument.

5 Tail-recursive reverse procedure

Note that the original problem was to write a procedure that takes a single argument, a list. Thus the procedure that takes two arguments, the regular argument and the partial-solution argument, cannot be the reverse procedure. It is instead a helper, rev_helper. We will write rev in terms of this helper. The rev procedure calls the helper with the original input and with an empty partial solution:

\[
\text{let rev} = \text{function lst} \to \text{rev_helper (lst, [])}
\]

How to write rev_helper? The idea is that as we go from input to recursive input,

- the first argument gets smaller/simpler, but
- the second argument (the partial-solution argument) gets larger.

The two extremes are the initial call (the first argument is the original input list, and the second argument is empty) and the base case (the first argument is empty, and the second argument is the final output to be returned).

We know how we want to base case to look:

- recursive input: [], [10; 20; 30]
- recursive output: [10; 20; 30]

Consider a call that results in that recursive input. The partial-solution argument [10; 20; 30] in the recursive input should be derived in a simple way from the input. However, in a reversal
of the way we’re used to, the partial-solution argument in the recursive input is bigger than the partial-solution argument in the original input. Thus the natural thing to do is cons a value onto the the partial-solution argument. Where does that value come from? The natural think is to remove the car of the first argument.

input: [10], [20;30]
recursive input: [], [10;20;30]

This works!

let rec rev_helper = function
    | [], result -> result
    | first::rest, result -> rev_helper (rest, first::result)

Examples:

# rev_helper ([1;2], [10;20;30]);;
- : int list = [2; 1; 10; 20; 30]
# rev_helper ([1;2;3], [10;20;30]);;
- : int list = [3; 2; 1; 10; 20; 30]

What’s going on? The partial solution argument is the reverse of the list consisting of the elements removed so far from the original input to obtain the first argument.

6 Running-time analysis of tail-recursive reverse procedure

To analyze the new reverse procedure, we analyze the helper:

Let \( h(m, n) \) be the worst-case running time of \( \text{rev} \_\text{helper} \) when the regular argument has length \( m \) and the partial-solution argument has length \( n \).

The recurrence relation for \( \text{rev} \_\text{helper} \) is as follows:

\[ h(0, n) \leq A \]
\[ h(m, n) \leq B + h(m - 1, n + 1) \]

The first statement in the recurrence relation shows that the second argument doesn’t matter. Thus, if we were to plug and chug our recurrence relation into a closed form solution, we would get the following:

\[ h(m, n) \leq Bm + A \]

This shows that the new reverse procedure is \( O(m) \), a linear-time procedure.
7 Other benefits of tail recursion

Consider the append we just wrote:

OI: append ([1; 2; 3], [4; 5; 6])
   RI: append([2; 3], [4; 5; 6])
      RI: append([3], [4; 5; 6])
         RI: append([], [4; 5; 6])
            RO: [4; 5; 6]
               RO: 3 :: [4; 5; 6]
                  RO: 2 :: [3; 4; 5; 6]
                     OO: 1 :: [2; 3; 4; 5; 6]

Notice that we need to store information about every function call even after its made the recursive call, because it modifies the recursive call’s output before returning. Function calls are organized in a stack, which gives the last-in, first-out behavior that we want for recursive calls. However, each function call takes up a nontrivial amount of space in the stack, and indeed, programs can run out of memory if the space allocated for the program’s stack gets full (this is called a stack overflow exception, which some of you may have seen before if you accidentally wrote an infinite loop).

A tail recursive function, however, needs to store no information about the previous call, because every call to the function will return the same thing! Therefore, most compilers will notice when a function is tail recursive, and this lets tail recursive functions occupy constant stack space, which in many cases is a key benefit to using this strategy.