Contents

Objectives

By the end of this lecture, you will be able to:

- know how to use accumulators in writing recursive functions
- simplify code to make debugging easier

1 Accumulators

So far we have seen some strategies for recursion. The strength of recursion is solving a more general problem. We often add another argument. Sometimes an extra argument is just a counter (today’s quiz) or a yes/no toggle. We used this to know if we should start with a * or +. We also used it in bignum-add-with-carry. Accumulators or other stored data are used to help with recursion. We saw accumulators used in Reverse. Other recursion strategies include Divide and conquer (mergesort) and breaking the problem into manageable pieces (Rackette).

Let’s look at the slow version of reverse.

```
OI. [3, 4, 1, 5, 6]
RI. [4, 1, 5, 6]
RR [6, 5, 1, 4]
OR [6, 5, 1, 4, 3]
```

What do you do to get from the Recursive input to the recursive output? You should append a list containing the head of OI to the recursive result. Here is the code:

```ocaml
/* reverse: list('a) => list('a)
** Input: a list of items
** Output: a list containing the same items,
** in reverse order
*/
let rec slowReverse: list('a) => list('a) = fun
    | [] => []
    | [hd, ..tl] => slowReverse(tl) @ [hd];
```

Because append takes time proportional to the length of the first list, the runtime is in O(n => n^2)

We can improve reverse with an accumulator and a helper.
let fastReverse: list('a) => list('a) = input => reverse_help(input, []);

/* reverseHelp: (list('a), list('a)) => list('a)
 ** Input:
 **   start, a list of items
 **   partial, a list to append to the reversed version of "start"
 ** Output: a list containing the items in "start" in reverse order,
 ** followed by those in "partial"
 */
let rec reverseHelp = (start, partial) =>
  switch(start) {
    | [] => partial
    | [hd, ... tl] => reverseHelp(tl, [hd, ...partial])
  };

The runtime of fastReverse is that of reverseHelp. And the runtime of reverseHelp is linear in the length of 'start', so that fastReverse is linear time. Here our accumulator is the 'partial' argument, it saves the initial items in the list.

1.1 Quiz: mapi

Consider the function mapi. With the following type signature: let mapi: ((int, 'a) => 'b, list('a)) => list('b) Same as List.map, but the function is applied to the index of the element as first argument (counting from 0), and the element itself as second argument. For example: mapi((x, y) => (x, y), ['a', 'b', 'c']) produces [(0, "a"), (1, "b"), (2, "c")]
mapi((x, y) => x*y, [3, 1, 4]) produces [0, 1, 8]

Here is a recursion diagram:

```
OI "+", [1, 3, 4]
   RI"+", [3, 4]
  RO [3, 5]
OO [1, 4, 6]
```

You may think that you can just add one to each item in RO, then cons on 1 + 0? That won’t work when it’s "*" instead of "+". What you really need to add is 0 to 1 and 1 to 3 and 2 to 4 and so on. We can strengthen the recursion with a helper. mapiHelper will take a function, a list of data, and an index starting at zero. The quiz was to write a type signature for mapiHelper and fill in some code. Here is the completed mapiHelper:

```
let rec mapiHelper: ((int, 'a) => 'b), list(int), int) => list('b) = (f,
  alod, n) =>
switch(alod) {
  | [] => []
  | [hd, ... tl] => [f(n, hd), ... mapiHelper(f, tl, n+1)]
};
```
2 Debugging programs

before you can begin debugging it is important to make sure your code is readable. It is always worth splitting into subtasks and cleaning up and simplifying code. This will make it easier for you and your TAs! We are going to take a block of code that a student wrote for iroot from last week’s homework that doesn’t work and try to find the bug. The initial code is 40 lines long, it complies but doesn’t produces the right answer.

```ocaml
let rec iroot: (int, int, int => int) => int =
  (n, m, proc) =>{
    let newBound = (n+m)/2;
    let procApply = proc(newBound);
    let procApplyN = proc(n);
    switch(procApply){
      |x when x > 0 => {
        if(procApplyN > 0){
          if(newBound-n == 1){
            n
            }
          else iroot(newBound, m, proc)
        } else if (procApplyN < 0) {
          if(newBound-n == 1){
            n
          } else iroot(n, newBound, proc)
        } else {
          newBound
        }
      } |x when x < 0 => {
        if(procApplyN > 0){
          if(newBound-n == 1){
            n
          } else iroot(n, newBound, proc)
        } else if (procApplyN < 0) {
          if(newBound-n == 1){
            n
          } else iroot(newBound, m, proc)
        } else {
          newBound
        }
      } |0 => newBound
      |_ => failwith("Incomplete Match Case error")
    };
  }
```

Can anyone find the bug? I can’t! Let’s forget about finding the bug and focus on making the code readable. We will start simply with the variable names. ’m’ and ’n’ are not the best choice because
they sound similar and they appear in the opposite of alphabetical order messing with our natural ordering skills. We will replace 'm' and 'n' with 'a' and 'b'. The variable name NewBound is really the midpoint so we will change it to mid. Similarly ProcApply is really the value at the midpoint so it will become midVal. ProcApplyN is really the value at the left end so it become leftVal. Also we noticed that newbound-a == 1 four times while we could just compute it once at the beginning, called short. Making these adjustments we get more readable code with better spacing.

```plaintext
let rec iroot: (int, int, int => int) => int =
    (a, b, proc) =>{
    let mid = (a + b) / 2;
    let midVal = proc(mid);
    let leftVal = proc(a);
    let short = (mid a == 1);
    switch (midVal){
        | x when x > 0 => {
            if (leftVal > 0){
                if (short) {
                    a
                }
                else iroot(mid, b, proc)
            } else if (leftVal < 0) {
                if (short) {
                    a
                }
                else iroot(a, mid, proc)
            } else {
                mid
            }
        }
        | x when x < 0 => {
            if (leftVal > 0){
                if (short) {
                    a
                }
                else iroot(a, mid, proc)
            } else if (leftVal < 0) {
                if (short) {
                    a
                }
                else iroot(mid, b, proc)
            } else {
                mid
            }
        }
        | 0 => mid
        | _ => failwith("Incomplete Match Case error")
    };
}
```

We can now pull out the short check so we only check it once at the beginning. We also seem to be working with midVal and leftVal a lot. If either is zero we know the answer. We can add checks to see if they are zero before we switch but we would still keep testing if they are greater than zero a
lot. To clean this up we can switch on if midVal > 0. Then our cases are true and false. Here is the updated code.

```plaintext
let rec iroot: (int, int, int => int) => int =
(a, b, proc) => {
    let mid = (a + b) / 2;
    let midVal = proc(mid);
    let leftVal = proc(a);
    let short = mid - a == 1;
    if (short) {a;}
    else {
        if (midVal == 0) {mid} else
        if (leftVal == 0) {a} else
        switch (midVal > 0) {
            | true =>
                if (leftVal > 0) {
                    iroot(mid, b, proc);
                } else if (leftVal < 0) {
                    iroot(a, mid, proc);
                } else {
                    mid;
                }
            | false =>
                if (leftVal > 0) {
                    iroot(a, mid, proc);
                } else if (leftVal < 0) {
                    iroot(mid, b, proc);
                } else {
                    mid;
                }
        };
    };
};
```

We are getting closer! Now we can get rid of the two else cases. We are still checking if leftVal > 0 twice so maybe we can switch on that as well.

```plaintext
let rec iroot: (int, int, int => int) => int =
(a, b, proc) => {
    let mid = (a + b) / 2;
    let midVal = proc(mid);
    let leftVal = proc(a);
    let short = mid - a == 1;
    if (short) {a;}
    else if (midVal == 0) {mid;}
    else if (leftVal == 0) {a;}
};;
```
The right hand side of the middle two cases seem the same and right hand side of the outer two cases also seem the same. In those cases you are really checking if two booleans are equal to each other. So we can switch on whether or not they are equal to each other. A few small things are still left to be changed. We define short and use it once. Why not just use it? We now have code that is simple to look at and we can debug it.

let rec iroot: (int, int, int => int) => int =
  (a, b, proc) => {
    let mid = (a + b) / 2;
    let midVal = proc(mid);
    let leftVal = proc(a);
    if (mid - a == 1) { a; }
    else if (midVal == 0) { mid; }
    else if (leftVal == 0) { a; }
    else {
      switch ((midVal > 0) == (leftVal > 0)) {
        | true => iroot(mid, b, proc)
        | false => iroot(a, mid, proc)
      :
    };
  };

A few observations are that if b - a = 1 then we know the interval [a,a+1] contains a zero, so that is a quick check to make. If b - a > 1, then mid is different from both b and a. Also there is no harm in recurring all the way to an interval of length one. It is only log cost. Using these observations and changing our switch statement to an "if" we get much simpler and readable code.

let rec iroot: (int, int, int => int) => int =
  (a, b, proc) =>
    if (b == a + 1) { a }
  else {
    let mid = (a + b) / 2;
    if ((proc(a) > 0) == (proc(mid) > 0)) {
      iroot(mid, b, proc)
    } else {
      iroot(a, mid, proc)
    };
  };


This only 10 line of code compared to the original 40! This process is walked through with even more detail in the lecture slides.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci0170/feedback