Lab 10: Sequences, Comparators, and Sorting
12:00 PM, Nov 11, 2019

Contents

1 A last ReasonML type, and its utility 1
  1.1 Sequences in ReasonML 2
  1.2 Using printing in debugging 3

2 Comparators 3

3 Just For Fun: Stable sorting 5

Objectives

By the end of this lab, you will know:

- what comparators are
- how to make a stable merge-sort
- whether insertion sort is stable

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

- print out debugging information in the middle of an ReasonML program
- sort things other than numbers, like BigNums

1 A last ReasonML type, and its utility

ReasonML has one more type that you haven’t seen, unit.

The usual way to construct a unit is with empty parens: (). Open up Sketch.sh and try typing (); just to see how it looks.

There are other functions in ReasonML that produce units. One of these is print_string. When ReasonML evaluates print_string("Hello!\n"), two things happen:

1. The string “Hello!” , followed by a newline, gets printed out.
2. The value () gets returned
Try typing `print_string("Hello!\n");` into Sketch and see what you get.

Take a look at [https://reasonml.github.io/api/Pervasives.html](https://reasonml.github.io/api/Pervasives.html) and read about “Output functions on standard output.”

Now think about printing a `list(int)`. You’d like to write

```reasonml
let rec printIntList: list(int) : unit = aloi => switch (aloi) ...
```

This is a little tricky. At the start of the list, you want to print a left bracket. But if you call this recursively, you’ll print another left bracket before the REST of the list, and so on. Here’s a solution: write

```reasonml
let rec printIntListContents: list(int) : unit = aloi => switch (aloi) ...
```

which prints out the items in a list, each item (except the last) being followed by a comma and a space.

Now how do you print a whole list? You print an open bracket, then the contents, then a right bracket. The problem is that we don’t have a way to “do a sequence of things in ReasonML” yet.

(There’s a very good alternative here: instead of writing a procedure that prints the list contents, you could write one that converted those contents to string form, and then the main list-printing program could put a left bracket at the front of that string, a right bracket at the end, and call `print_string` on the whole result. But we’re walking you through this alternative to illustrate something else, so bear with us.)

### 1.1 Sequences in ReasonML

You’ve seen semicolons in ReasonML as a way of separating bindings and commands. There is another use, however: semicolons are used to form *sequences*.

Try typing

```reasonml
{4;5;"hello";[1,2];};
```

into ReasonML. You’ll get some warnings, but the value of the expression you just typed (yes, it’s an expression!) is `[1,2]`, i.e., the last item in the sequence.

What’s going on here is that `;` is actually an *infix operator* in ReasonML, and the value of `a;b` is just the value of `b`. It seems as if it’d be a waste of time to ever write a sequence, because you could just have written the last item and gotten the same result.

The exception is when the expression `a` has a *side effect*, like printing out some text to your terminal. Try typing

```reasonml
{print_string("Hello!\n"); 4;};
```

and see what happens.
There were no warnings this time. Why not? Well, the type of `print_string` is `str -> unit`, and when the semicolon operator gets anything `except` a unit as its first argument, you get a warning, but when it gets a unit as the first argument, it produces no warning. This is why we are doing this exercise so you are able to print out in the middle of your procedures.

With this in mind, you can write

```reason
let printIntList: list(int) => unit = aloi =>
  (print_string("["); printIntListContents(aloi); print_string("]"););
```

**Task:** Write `printIntListContents`, as well as some tests for your procedure. Why is the output type for this procedure `unit`?

**Hint:** You will want to use `print_string` and `print_int`.

### 1.2 Using printing in debugging

Suppose you write a `length` procedure like this:

```reason
let rec len: (list('a)) => int = alod => switch alod {
  | [] => 0
  | [hd, ...tl] => 1 + len(tl)
};
```

which happens to be perfectly OK code, but because of a mistake in some check-expect, you think it might be wrong. How can you use our `printIntList` code from above to help you debug? Well, you’re probably pretty confident about the base case, so let’s do something with the recursive case. Because our printing procedure only works on int lists, we’ll have to change the signature a little. But after that, look at this small change to the recursive case:

```reason
let rec len : list(int) => int = alod => switch alod {
  | [] => 0
  | [hd, ...tl] => {printIntList (tl); print_string("\n"); 1 + len(tl)};
};
```

Go ahead and type that into Sketch, and try computing the length of the list `\[1, 2, 3\]`.

Helpful, eh?

**Task:** Now write `subsets` in ReasonML, and insert something that prints out the argument right before the switch expression (where it does the main pattern matching on the split between an empty list and a cons list), followed by a new line. See what happens when you compute the subsets of `\[1, 2, 3\]`.

**Note:** You should define `set('a)` as a type.

**Note:** If you don’t have your implementation of subsets on a department machine, feel free to use the following Racket code:

```racket
(define (subsets set)
  (cond
    [(empty? set) (list empty)])
```

3
2 Comparators

We call operators like $>$, $<$, and $=$ *comparators*, because they compare values. These comparators are built in to ReasonML. Moreover, they are *polymorphic* predicates—they consume `int` values, or `float` values, etc.—and they produce `boolean` values.

Here is an example of a comparator that is not built in to ReasonML. It compares two strings, based on their respective lengths:

```reasonml
let stringCompare : (string, string) => bool = (str1, str2) => (String.length(str1) < String.length (str2));
```

As you can see, our `stringCompare` predicate is designed specifically for strings. It consumes two strings, and produces `true` when the first is shorter than the second, and `false` otherwise.

But here’s the thing: it’s hard to remember when `stringCompare` produces `true`: is it when the first string is *shorter* than the second one, or *longer* than the second one? Wouldn’t it be preferable if `stringCompare` produced a more informative value, such as `Less` or `Greater`? With ReasonML’s powerful variant type system, we can do exactly this.

First, we define a generalized comparison type:

```reasonml
type comparison =
  | Less
  | Equal
  | Greater;
```

Second, we rewrite `stringCompare` so that it produces a value of type `comparison` instead of a boolean:

```reasonml
let stringCompare : (string, string) => comparison = (str1, str2) =>
  if (String.length(str1) < String.length(str2)) { Less }
  else if (String.length(str1) > String.length(str2)) { Greater }
  else { Equal };
```

**Task:** Write a procedure `bignumCompare` which compares two bignums. Your procedure should consume two `bignum` values and produce the appropriate `comparison` value.

**Hint:** Remember that the number 123 is represented in reverse order by the bignum `[3, 2, 1]`. For example:

1 We wrote it ourselves!
bignumCompare([], [1, 2, 3]);
=> Less

bignumCompare([4, 7, 1], [5, 7, 1]);
=> Less

bignumCompare([4, 7, 1], [5, 7, 1, 2]);
=> Less

bignumCompare([1, 2], [2, 1]);
=> Greater

bignumCompare([1, 8, 8, 9], [1, 8, 8, 8]);
=> Greater

bignumCompare([7, 1], [7, 1]);
=> Equal

**Hint:** Do not use library procedures like List.length or List.reverse. They will not lead to an efficient solution. Instead, write a procedure that recurs on two lists.

**Note:** You should define bignum as a type.

**Task:** What is the run time of your bignum comparator?

**Task:** Rewrite insertionSort so that it can use a comparator like bignumCompare as its last argument. The type becomes

```
insertionSort : (list('a), ('a, 'a) => comparison) => list('a)
```

To get you started, here’s the code for the ordinary insertion sort:

```plaintext
let rec insert : (int, list(int)) => list(int) = (n, aloi) => switch (n, aloi) {
  | (n, []) => [n]
  | (n, [hd, ...tl]) => if (n < hd) {[n, ...aloi]}
  | else {[hd, ...insert(n, tl)]}
};

let rec insertionSort : list(int) => list(int) = aloi => switch aloi {
  | [] => []
  | [hd, ...tl] => insert(hd, insertionSort(tl))
};
```

You’ll need to change all the “int”s to 'a, so that it can work with arbitrary data.

Now define a new comparison, pairCompare, that compares two int * string pairs, and says that (n1, str1) is less than (n2, str2) exactly when n1 < n2 (i.e., it compares only the “int” part). Use your insertion sort to sort a list that looks like this:

```
[(2, "two"), (1, "first"), (1, "second"), (0, "zero"), (0, "last")]
```
**Task:** Is the resulting sorted list a *stable* sort of the data, in the sense that if two items in the input are equal, then their order in the output is unchanged? (i.e., \((1, \text{"first"})\) should still come before \((1, \text{"second"})\).) Experiment to find out.

| You've reached a checkpoint! Please sign up to get a lab TA to review your work. |

### 3 **Just For Fun: Stable sorting**

What about building a stable sorting algorithm?

**Task:** Rewrite `mergesort` (with a comparator argument) and once again check whether it is stable, by applying it to the example above that used `pairCompare`.

Here's a slightly modified version of the `mergesort` from class. The two main differences are:

- The code uses the “compute the length of the list to split it in two” approach rather than the card-dealing approach, which helps in making it a stable sort.
- The code tests whether the head of the left list is less than or equal to the head of the right list; this too helps in maintaining stability.

```
open CS17setup;

let rec merge: (list (int), list (int)) => list (int) = (aloi1, aloi2) =>
  switch (aloi1, aloi2) {
  | (\[], \_ ) => aloi2
  | (\_, \[] ) => aloi1
  | ([hd1, ...tl1], [hd2, ...tl2]) =>
    if (hd1 <= hd2) {
      [hd1, ...merge(tl1, aloi2)]
    } else {
      [hd2, ...merge(aloi1, tl2)]
    }
  }

checkExpect (merge ([1,3,4], [2,5,6,8]), [1,2,3,4,5,6,8], "both non-empty")
checkExpect (merge ([], [1,2]), [1,2], "first is empty")
checkExpect (merge ([1,2], []), [1,2], "second is empty")
```

6
take2: (int, list (int)) => (list (int), list (int))
inputs:
  a number, n, of items to select from the input list
  a list, aloi, of integers
output:
  a pair p1, p2 where p1 contains the first n items of aloi, in the order
  they appear in aloi,
  and p2 contains the remaining items, again in the order they had in aloi.
*/
let rec take2: (int, list (int)) => (list (int), list (int)) = (n, aloi) =>
  switch (n, aloi) {
  | (0, []) => ([], [])
  | (0, _) => ([], aloi)
  | (_, []) => failwith ("No items left to take")
  | (n, [hd, ... tl]) => {
    let
    (p1, p2) = take2 (n - 1) (tl);
    ([hd, ... p1], p2)
  }
}

checkExpect (take2 (3, [1,2,3]), ([1,2,3], []), "take all");
checkExpect (take2 (0, [1,2,3]), ([], [1,2,3]), "take none");
checkExpect (take2 (0, []), ([], []), "take none empty");
checkExpect (take2 (3, [1,2,3,4,5]), ([1,2,3], [4,5]), "take some");

let split: list (int) => (list (int), list (int)) = aloi =>
  switch (aloi) {
  | [] => ([], [])
  | _ => {
      let n = List.length (aloi);
      take2 ((n/2), aloi)
    }
  }

checkExpect (split ([1,2,3]), ([1], [2,3]), "split odd");
checkExpect (split ([], []), ([], []), "split empty");
checkExpect (split ([1,2,3,4]), ([1,2], [3,4]), "split even");

mergesort: list (int) => list (int)
input:
  a list, aloi, of integers
output:
  a list containing the same integers, but
  sorted so that each is greater than or
  equal to the previous one
*/
let rec mergesort: list (int) => list (int) = aloi =>
  switch (aloi) {
    | [] => []
    | _ => {
      let n = List.length (aloi);
      take2 ((n/2), aloi)
    }
  }

checkExpect (mergesort ([1,2,3]), [1,2,3], "mergesort");
checkExpect (mergesort ([3,2,1]), [1,2,3], "mergesort");
checkExpect (mergesort ([4,3,2,1]), [1,2,3,4], "mergesort");
| [[]] | [] => aloi |
| _ => {} |
|     let (s1, s2) = split (aloi); |
|     merge (mergesort (s1), mergesort (s2)) |
| ; |

You may need to change the `open` line at the top to make it work for you.

**Task:** Finally, make a copy of this mergesort procedure, called `mergeSortB` (the “b” is for “broken”) that uses a less-than rather than a less-than-or-equal-to comparison between `hd1` and `hd2`, which should make the sort unstable. Then find a list of int-string pairs that are sorted unstably by this revised version. Because the code we gives you takes in int lists, you will need to convert it to take in 'a lists so that it could work with pairs.

You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

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](http://cs.brown.edu/courses/csci0170/feedback).