CS 131 Week 2 Worksheet Answers

(20 min) Week 1 Crash Course

As we didn't have a discussion last week, let's do a crash course on OCaml basics! I would highly recommend this [Tour of OCaml]; I relied heavily on this page when I learned OCaml last year. Another useful resource that goes more in depth on how OCaml works is [OCaml Programming: Correct + Efficient + Beautiful].

Some basics you'll need:

- OCaml is strongly, statically typed with a powerful type inference system
- We use `let` to bind values to names; for the purposes of this class, all our variables are immutable
- We use `let ... = ... in ...` to define names locally
- We also use `let` to define functions and `fun` to define anonymous functions
  - Use `let rec` when you want to write recursive functions (don't use loops!)
  - Functions can be partially applied (and thus higher order)
- The empty tuple `()` is of the `unit` type, indicating a lack of data
- Lists are ordered collections of elements of a single type separated with `;` and enclosed in `[]`
- Tuples are ordered collections of elements of any type separated with `,` and enclosed in `()`
  - Express a tuple's type by separating the type of each element with `*`

And some quick tips:

- Don't be afraid to write helper functions
  - Break down your problems into smaller chunks and solve each one individually; stitch things together with function calls
  - The longest function I wrote for Homework 1 or 2 was 23 lines long (and that one was already getting a bit unwieldy)
- Let the compiler and typing system work for you
  - Read your errors! Fix your typing issues and logic will be shockingly easy to debug
- Think recursively
  - Thinking "If I've solved this same problem for the n-1 case, can I solve it for the n case?" can get you nice recursive relationships
  - Recursive functions are similar to proofs by induction on natural numbers (such as base case of 0, then inductive case of n -1). Always think in terms of the base case and the recursive case.
- Pattern match everything!

If you don't have OCaml installed locally, I would highly recommend doing that right now (link: W25 CS 131 Software)

Let's get warmed up! I recommend developing in a .ml file, then opening a REPL and loading that file when you want to test (You can accomplish this with `#use "FILENAME";;`).

Implement a function that takes a tuple of two `int option`s and returns another `int option`.
If the first value is `None`, return none.
If only the second is `None`, return the first value.
If neither are `None`, return their sum.

```
let enforced_sum xy =
    match xy with
        | (None, _) -> None
        | (x, None) -> x
        | (Some x, Some y) -> Some (x + y)
```

Implement List.length, i.e. a function that takes a list and outputs its length. Don't be scared to write a function inside of this function! (Hint: the pattern in this problem is very useful)
Non-tail-recursive solution:

```
let rec length list =
    match list with
        | [] -> 0
        | _::t -> 1 + length t
```

Tail-recursive solution:

```
let length list =
    let rec helper acc list =
        match list with
            | [] -> acc
            | _ :: t -> helper (acc + 1) t
    in helper 0 list
```

Implement List.rev, i.e. a function that reverses a given list. Make sure your function works with lists of all types! (Recall that when we work with lists in OCaml, think in terms of `h::t`)
Non-tail-recursive solution:

```
let rec reverse list =
    match list with
        | [] -> []
        | h::t -> reverse t @ [h]
```

Tail-recursive solution:

```
let reverse list =
    let rec helper acc = function
        | [] -> acc
        | h :: t -> helper (h :: acc) t
    in helper [] list
```

Use the function you just implemented to implement a function that checks if a given list is a palindrome. Hint: a palindrome is equivalent to its reverse! If you can't figure out the last problem, feel free to use `List.rev` instead. If you've figured out the one-liner, try to compare equality manually instead!

```
let palindrome list =
    let reversed = reverse list in
    list = reversed
```

or for some style points:

```
let palindrome list =
    let reversed = reverse list in
    let rec helper a b =
        match a, b with
            | h1 :: t1, h2 :: t2 -> h1 = h2 && helper t1 t2
            | [], [] -> true
            | _ -> false
    in helper list reversed
```

(10 mins) BNF & EBNF Notation

[Backus–Naur form](#) is a notation used for context-free grammars (and for our purposes, programming languages).

- In this class you will also be tested on extended BNF, aka EBNF, which is just BNF with some syntactic sugar borrowed from regex
- You should know how to desugar EBNF into plain BNF

Common EBNF notation:

`[x]`	`x` is optional	`x?`	`x` is optional
`{x}`	0 or more repetitions of `x`	`x*`	0 or more repetitions of `x`
`{x}+`	1 or more repetitions of `x`	`x+`	1 or more repetitions of `x`
		`x|y`	`x` or `y`

Usually people will stick to the style on the left or the style on the right; Eggert tends to use the left. Parentheses are usually used for grouping. Terminals are usually distinguished with quotes.

Here's how to convert these constructs to BNF:

EBNF	**BNF**
`S ::= [x]`	`S ::= ""` `S ::= x`
`S ::= {x}`	`S ::= ""` `S ::= x S`
`S ::= {x}+`	`S ::= x` `S ::= x S`
`S ::= a(x|y)`	`S ::= ax` `S ::= ay`

Let's do a slightly harder one. Translate `S ::= a({x|y}+)` to BNF. Feel free to create "helper rules" (more non-terminals).

```
S ::= aR
R ::= x
R ::= y
R ::= xR
R ::= yR
```

Read the grammar for Rust's match expression and translate it to BNF.

```
MatchExpression ::= "match" Scrutinee "{" InnerAttributes  "}"
MatchExpression ::= "match" Scrutinee "{" InnerAttributes  MatchArms"}"

InnerAttributes ::= ""
InnerAttributes ::= InnerAttribute InnerAttributes

Scrutinee ::= Expression

MatchArms ::= ""
MatchArms ::=  MatchArm "=>" ExpressionWithoutBlock "," MatchArms
MatchArms ::=  MatchArm "=>" ExpressionWithBlock "," MatchArms
MatchArms ::=  MatchArm "=>" ExpressionWithBlock MatchArms
MatchArms ::= MatchArm "=>" Expression
MatchArms ::= MatchArm "=>" Expression ","

MatchArm ::= OuterAttribute MatchArm
```

```
MatchArm ::= Pattern MatchArmGuard
MatchArm ::= Pattern

MatchArmGuard ::= “if” Expression
```

Challenge: read the [grammar for the DOT language] and write a valid graph. Try to use at least eight rules so this exercise isn’t trivial.
Answers may vary.

(10 mins) Railroad Diagrams

Eggert once referred to these as diagrams "in the style of Webber" 🤨so remember that if you don't read the textbook.

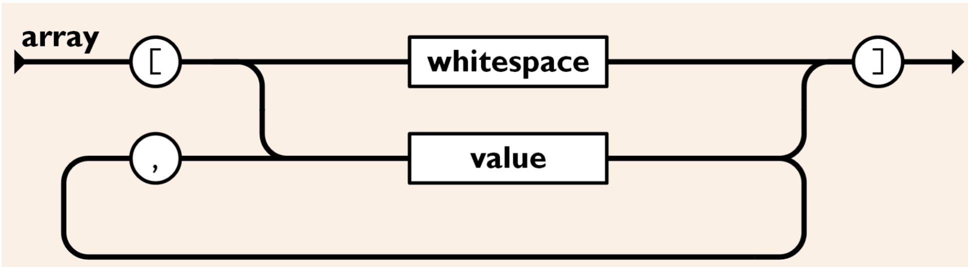
JSON is a popular data-interchange format. It's also fairly simple to parse.
Today we'll look at objects in JSON, which look like this:

```
{
  "latitude": 34.07099896148599,
  "longitude": -118.44504522008893,
  "name": "Bruin Statue"
}

{
  "center": {
    "x": 5.0,
    "y": 0.0
  },
  "radius": 4.0
}

{
  "error": null,
  "response": [
    "878ec3ec",
    "c1a01689",
    "e956be4d",
    "1fc07907"
  ]
}
```

As reference here is the railroad diagram for `array` from the JSON website, which has a simpler grammar than `Object`.



Create a grammar and corresponding railroad diagram for `Object`. You can ignore whitespace and you can assume there are nonterminals `String` and `Value`. You can define more nonterminals if you want.

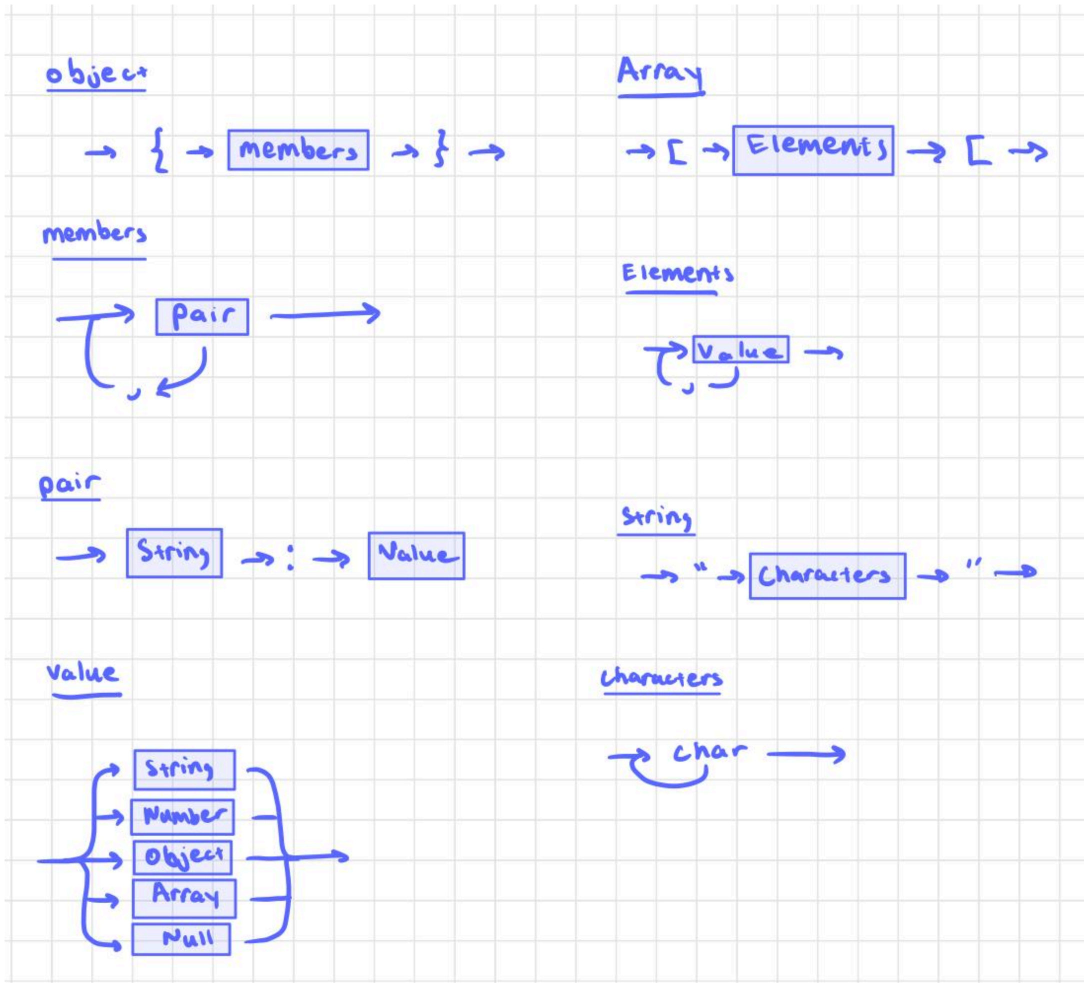
Remember objects can be empty: `{}`. Also remember that commas only go **between** attribute-value pairs.

```
Object -> '{' '}' | '{' Members '}'
Members -> Pair | Pair ',' Members
Pair -> String ':' Value
```

Past this point was not intended to be required in your answer, but for completeness,

```
Value -> String | Number | Object | Array | Null
Array -> '[' ']' | '[' Elements ']'
Elements -> Value | Value ',' Elements
```

```
String -> '"' Characters '"'
Characters -> <nothing> | Char Characters
Char -> 'any Unicode character except “ or \ or escape sequence'
Number -> 'integer or floating-point number'
Null -> 'null'
```



(25 mins) More OCaml Practice

Open up [the `List` module documentation](). It will be handy for hw1 and hw2.

`List.fold_left` is the functional way to process a linked list. It lets you "build up" a result between iterations and returns the result at the end.
`List.fold_right` is the same idea but iterates in reverse. And the order of parameters is switched for some reason.

	OCaml	Python
left fold		

map	`('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc`
	`let value =`
	`  List.fold_left f init list`
	`in ...`
		`value = init`
		`for x in list:`
		`  value = f(value, x)`
		`...`

filter	`('a -> 'b) -> 'a list -> 'b list`
	`let mapped =`
	`  List.map f list`
	`in ...`
		`mapped = []`
		`for x in list:`
		`  mapped.append(f(x))`
		`...`

concat_map
(flat map)
	`('a -> bool) -> 'a list -> 'a list`
	`let filtered =`
	`  List.filter predicate list`
	`in ...`
		`filtered = []`
		`for x in list:`
		`  if predicate(x):`
		`    filtered.append(x)`
		`...`

	`('a -> 'b list) -> 'a list -> 'b list`
	`let newlist =`
	`  List.concat_map f list`
	`in ...`
		`newlist = []`
		`for x in list:`
		`  # f(x) returns a list`
		``  # `+` concatenates lists``
		`  newlist += f(x)`
		`...`

Many of the functions in the `List` module are ""redundant"" in that you can implement them *in terms of* other functions. Higher-order functions are very flexible.

In the below table, implement X in terms of Y. Work together!

<table>
<tr><td>X</td><td>Y</td><td>Answer</td></tr>
<tr><td><code>map</code></td><td><code>concat_map</code></td><td><pre>let my_map f list =
  (* remember [f x] is a list of one element *)
  concat_map (fun x -> [f x]) list</pre></td></tr>
<tr><td><code>filter</code></td><td>recursion</td><td><pre>let rec my_filter p list =
  match list with
  | [] -> []
  (* when is a keyword that means some boolean
     expression must be true for the match
     to succeed. Here it means "keep h
     in the result list if p h is true". *)
  | h::t when p h -> h::(filter p t)
  | _::t -> filter p t</pre></td></tr>
<tr><td><code>map</code></td><td><code>fold_right</code>
(Why is fold_left less appropriate here?)</td><td><pre>let my_map f list =
  fold_right (fun e acc -> (f e)::acc) list []</pre></td></tr>
<tr><td><code>concat</code></td><td><code>concat_map</code></td><td><pre>let concat list_of_lists =
  List.concat_map (fun x -> x) list_of_lists</pre>
<code>map</code> in terms of <code>concat_map</code> was done by "nullifying" the concat part with lists of length 1.
<code>concat</code> in terms of <code>concat_map</code> is done by "nullifying" the map part with the identity function.</td></tr>
<tr><td><code>map</code></td><td>recursion</td><td><pre>let rec map f list =
  match list with
  | [] -> []
  | h :: t -> (f h) :: (map f t)</pre></td></tr>
<tr><td><code>exists</code></td><td><code>fold_left</code></td><td><pre>let exists f list =
  List.fold_left (fun acc x -> acc || f x) false
list</pre></td></tr>
<tr><td><code>exists</code></td><td>recursion</td><td><pre>let rec exists f list = match list with
| [] -> false</pre></td></tr>
</table>

<table>
<tr><td></td><td></td><td><pre>| h :: t -> f h || exists f t</pre></td></tr>
<tr><td><code>find_opt</code></td><td><code>fold_left</code></td><td><pre>let find_opt f =
  let fold_func = fun acc x ->
    match acc with
    | None -> if f x then Some x
    | Some y -> Some y (* can also be y -> y *)
  in
  fold_left fold_func None
  (* usually fold_left f acc list *)</pre>Note the use of partial application; we don't need an explicit parameter <code>list</code>.</td></tr>
<tr><td><code>filter</code></td><td><code>concat_map</code></td><td><pre>let filter f =
  let map_func = fun x ->
    if f x then [x] else []
  in
  concat_map map_func</pre></td></tr>
<tr><td><code>partition</code></td><td><code>fold_right</code></td><td><pre>let partition f list = List.fold_right (fun x
(true_list, false_list) ->
   if f x then (x::true_list, false_list)
   else (true_list, x::false_list))
  list ([], [])</pre></td></tr>
<tr><td><code>combine</code>
(if the lists have unequal lengths, do whatever is easy to implement)</td><td>recursion</td><td><pre>let rec combine xs ys = match (xs,ys) with
   | (xh::xt, yh::yt) -> (xh,yh) :: combine xt yt
   | _ -> []</pre></td></tr>
<tr><td><code>rev</code></td><td><code>fold_left</code>
(Why is fold_right less appropriate here?)</td><td><pre>let rev = fold_left (fun acc x -> x :: acc) []</pre></td></tr>
</table>

(Supplement) Sponsorship Break: Iterators

"How is CS 131 useful for *real* programming?"
Higher-order functions on collections are one big example. Many, many, programming tasks are "do this thing for every item in this collection".

Python: `itertools`, list/dict/set/generator comprehensions (think about how easy `x = [func(i) for i in list]` would be to implement in OCaml)
C#: `IEnumerable<T>`
Rust: iterators
JavaScript: methods on arrays
Go: just added in 1.23!

C++ and Java: kinda… they have "iterators" but (1) those are closer to pointers and not as high-level as the others, and (2) the stdlibs don't make mature use of higher-order functions.

Some light reading:

IEnumerable, for a More Elegant C# (focus on the method chaining over the SQL-like syntax)
If you intern at a .NET shop you may encounter EF Core. You write code that works with `IEnumerable<T>`s and it generates(!!) SQL that does the corresponding operation on a database. It uses serious wizardry with .NET's compiler API.

Rust Iterators; here's a peek:

> In my opinion, iterators are one of the most powerful tools you can add to your toolbelt as a programmer. I've met many programmers who struggled to understand the various iterator functions and how to use them. In this article, I explore a number of iterator functions available in Rust's standard library … When I'm writing code in Rust, I find that my ideas can often be expressed more clearly and more succinctly if I forgo the for loop and rather make heavy use of Rust's iterator functions.

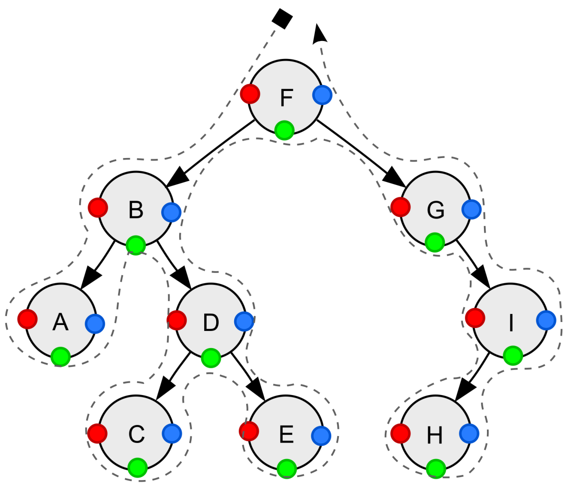
Why People are Angry over Go 1.23 Iterators; There was some fun niche internet drama when Go had iterators added in its last release: “TL;DR It makes Go feel too “functional” rather than being an unabashed imperative language.”

As per the original proposer of Go’s iterators: “The key point here is our programmers are Googlers, they’re not researchers. They’re typically, fairly young, fresh out of school ... the language that we give them has to be easy for them to understand and easy to adopt.” Maybe this is worth mulling over a little bit...

(Supplement) Type constructors: trees

Consider this type for a binary tree:

```
; Node is (value, left subtree, right subtree)
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree
```



Source: Wikipedia

Here's the representation of the above tree:

```
Node ("F", Node ("B", Node ("A", Empty, Empty), Node ("D", Node ("C",
Empty, Empty), Node("E", Empty, Empty) ) ), Node ("G", Empty, Node
("I", Node ("H", Empty, Empty), Empty)))
```

Write the following functions that traverse a `'a tree` and return a `'a list` of the values in the order they were visited.

Pre-order (node visited at position red 🔴): F, B, A, D, C, E, G, I, H
The rule for pre-order is root-left-right.

```
let rec pre_order root =
```

In-order (node visited at position green 🟢): A, B, C, D, E, F, G, H, I
The rule for in-order is left-root-right.

```
let rec in_order root =
```

Post-order (node visited at position blue 🔵) is trivial after solving `pre_order`.

Further practice

Complete the "implement X in terms of Y" table.

Implement `pre_order` and `in_order`, but without list append `@`. Hint: you could create helper functions with the signature `'a tree -> 'a list -> 'a list`.

Implement breadth-first traversal of `tree`s. Hint: think about what information you need at each level of the tree.

Homework 1 Hints

W25 CS 131 Homework 1 Hints