Week 1 Discussion

CS 131 Section 1B

1 April 2022

Danning Yu

Administrivia

- About me
  - Danning Yu, danningyu@ucla.edu
  - Did undergrad at UCLA, currently an MS student
  - Took this class W21
- Office Hours: T 9-10am, 1-2pm https://ucla.zoom.us/j/6682857378
- Piazza: https://piazza.com/ucla/spring2022/cs131
- See syllabus for homework deadlines, grading policies, etc.
- Make sure you have a SEASnet account

Assignments

- 6 homeworks + project
  - Project is just another homework, basically
- Make sure everything runs on SEASnet servers
  - lnxsrv[11, 12, 13, or 15].seas.ucla.edu
- Personal opinion: HW2 and HW5 are hardest
- Don't copy solutions from online sources or each other
  - Discussing general ideas and approach is okay, don't share code with each other
  - Submissions will be checked for plagiarism
- HW1 due 4/8 11:55pm

Functional Programming

Functional Programming

- No "side-effects"
  - Calling a function twice with the same arguments gives identical result
  - “Pure function”, behaving similar to mathematical functions
  - Side effects present to allow for I/O
- Functions are first class citizens
  - Functions can take other functions as arguments or return them as result (much like other variables)
- Iteration usually implemented with recursion

Why Functional Programming

- Similar ideas can be found in most modern programming languages
  - e.g. Python, C++, Java, Swift, Scala, Kotlin, and many more
    - Hint: if you're struggling with OCaml syntax, try doing the HW in Python first in a functional manner
  - We will see examples later when we cover other languages
- Functional programming makes compiling and testing easier
  - No side effects makes reasoning on the code easier
- Easier to build scalable systems
- Computing can be distributed on multiple machines more easily when there are no shared states or side effects

OCaml

OCaml

- Functional programming language
    - Statically typed: every value (including function) has a type
- Type checking at compile time
    - Catches a lot of errors!
- Type inference
    - In many cases, you don’t need to specify types
    - Allows you to deduce if you coded your function correctly
- Garbage collection
- Compiled language
    - But includes an interactive interpreter
- Has object oriented aspect, but we won't use that in this class

OCaml Basics

- Hello, world: `print_string "Hello, world!\n";;`
    - `print_string` is the function, everything that follows are arguments
    - End statement with 2 semicolons when using interpreter (not needed in code files)
    - OCaml will output the return value
- Comments: `(* this is a comment *)`
- Local bindings ("variables"): `let val1 = 5;;`
    - Only usable within the scope of the function
- Functions: `let <func_name> [args…] = <func_body>`

```
# let average a b =
    (a + b) / 2;;
val average : int -> int -> int = <fun>
```

```
# let average a b =
    (a +. b) /. 2.0;;
val average : float -> float -> float = <fun>
```

Recursive Functions

- Recursive functions: add a `rec` keyword after `let`
  - let **rec** <func_name> [args…] = <func_body>

```
# let rec factorial a =
    if a = 1 then 1 else a * factorial (a-1);;
val factorial : int -> int = <fun>
```

```
# factorial 5;;
- : int = 120
```

- Mutual recursion: use the `and` keyword

```
let rec even x =
  if x = 0 then true
  else odd (x - 1)
and odd x =
  if x = 0 then false
  else even (x - 1)
```

Local Variables in Functions

- Add keyword `in` after the `let` statement to make the value available in the rest of the function

```
# let average a b =
    let sum = a +. b in
    sum /. 2.0;;
val average : float -> float -> float = <fun>
```

Lambda Functions

- Lambda functions (aka anonymous functions) are defined with keyword `fun`
  - Useful when supplying a (small) routine as a function argument

```
# (fun x -> x * x) 5;;
- : int = 25
```

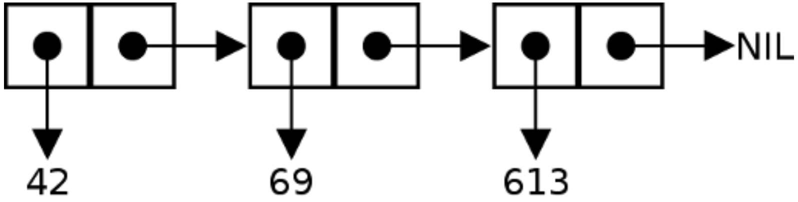
- The following 2 are equivalent

```
# let square x = x * x;;
val square : int -> int = <fun>
```

```
# let square = fun x -> x * x;;
val square : int -> int = <fun>
```

- Confused? Here's lambda functions in Python
  - `(lambda x, y: x + y)(2, 3)`
  - `high_ord_func = lambda x, func: x + func(x)`
  - `high_ord_func(2, lambda x: x + 3)`

OCaml Lists



- Defining a list: `let x = [1; 2; 3; 4; 5];;`
- All elements are of same type
- Under the hood, lists can thought of as immutable singly-linked lists
  - Iteration and appending to the end is fast, random access is slow
- Non-empty list consists of a head and a tail
  - Head: the first element; tail: the rest of the list
  - `List.hd x;;` returns int = 1
  - `List.tl x;;` returns int list = [2; 3; 4; 5]
- We can add an element to the beginning of the list with cons “operator” `::`
- `0::[1; 2; 3]` gives us `[0; 1; 2; 3]`
- `0::1::2::[3]` also gives us `[0; 1; 2; 3]`
  - `::` is right-associative
- `1::2` or `[1]::[2]` is not correct
- Lists are immutable: if you want to change a value in the list, need to create new list

Useful List Operations

- `List.map`: transforms each element with a function
- `List.filter`: returns a list that contains elements that matches the function
- `List.rev`: reverses the list
- `List.for_all`: checks if **all** elements satisfy the specified function
- `List.exists`: checks if **any** element in the list satisfies the specified function
- Lambda functions can be useful in these operations

```
# List.filter (fun x -> x < 3) [1; 2; 3; 4; 5];;
- : int list = [1; 2]
```

```
# List.rev [1; 2; 3; 4; 5];;
- : int list = [5; 4; 3; 2; 1]
```

```
# List.for_all (fun x -> x < 3) [1; 2; 3; 4; 5];;
- : bool = false
# List.for_all (fun x -> x < 6) [1; 2; 3; 4; 5];;
- : bool = true
```

```
# List.exists (fun x -> x = 3) [1; 2; 3; 4; 5];;
- : bool = true
# List.exists (fun x -> x = 6) [1; 2; 3; 4; 5];;
- : bool = false
```

Pattern Matching

- "Better" version of switch statements
  - List possible cases in a cleaner way
- Sequential check from the beginning and find the first rule that matches
- Underscore is often used as a wildcard that matches any value
- Cleaner than conditional when there are many cases
- Compiler will tell you if there exist cases your rules can't match
  - Highly useful in catching bugs!
- Use conditionals with `when` keyword

```
# let is_zero x = match x with
    0 -> true
  | _ -> false;;
```

```
# let is_zero x =
    if x = 0 then true
    else false;;
```

```
# let first l = match l with
    head :: tail -> head
  | _ -> 0;;
val first : int list -> int = <fun>
# first [1;2;3];;
- : int = 1
```

```
# let rec factorial x = match x with
    x when x < 2 -> 1
  | x -> x * factorial (x - 1);;
val factorial : int -> int = <fun>
```

Tuples

- A collection of heterogeneous values
  - Values separated by commas
  - Typically surrounded by parentheses
- `let x = "foo", 3.14, 'c', false;;`
  - `val my_tuple : string * float * char * bool`
- Accessing tuple elements
  - Tuples with two elements: `fst my_tuple`, `snd my_tuple`
  - Pattern matching

```
# let (x, y) = (3, "foo");;
val x : int = 3
val y : string = "foo"
```

```
# let my_fst (a, b) = a;;
val my_fst : 'a * 'b -> 'a = <fun>
# my_fst (3, "foo");;
- : int = 3
```

Variants

- Another way to have user defined types
- Use as enumeration

```
# type color = Red | Green | Blue;;
type color = Red | Green | Blue
# let my_blue = Blue;;
val my_blue : color = Blue
```

- Use as C/C++ like union

```
# type my_type =
    | A of int
    | B of string;;
type my_type = A of int | B of string
# let my_a = A 15;;
val my_a : my_type = A 15
# let my_b = B "hello";;
val my_b : my_type = B "hello"
```

```
# let my_print x =
    match x with
      A a -> print_int a
    | B b -> print_string b;;
val my_print : my_type -> unit = <fun>
# my_print (A 8);;
8- : unit = ()
# my_print (B "foobar");;
foobar- : unit = ()
```

Running OCaml From File

- Pretend the following is in `somecode.ml`

```
let rec get_head x =
  match x with
    | [] -> 0
    | first :: rest -> first

let x = print_int (get_head [1; 2; 3])
```

- To run: `ocaml somecode.ml`
- Or: start OCaml interpreter, and copy paste everything in to run
  - Don't forget to add `;;`
- When you submit your code, `ocaml <file_name>` should run without error

Context Free Grammars
Homework 1

Context Free Grammars (CFGs)

- A grammar defines a language (programming or otherwise)
  - What sentences are valid for a language
- In programming languages
  - Grammar does not say what instructions in that language mean, it just defines the allowed syntax
  - We can check if `print("Hello, World!")` is valid, without defining what it does
  - Syntax, not semantics
- There are multiple types of grammars, for this homework, you only need to know context-free grammars
  - Covered in CS 181 as well

CFG Rules

- Symbol:
  - Terminal: symbol that can't be replaced by other symbols (e.g. +)
  - Nonterminal: symbol that can be replaced by other symbols (e.g. BINOP)
- Rule:
  - Defines how a nonterminal symbols can be replaced with other symbols
- Grammar contains a starting symbol (e.g. EXPR), and a set of rules

Sample grammar:

```
EXPR -> NUM
EXPR -> NUM BINOP NUM
BINOP -> +
BINOP -> -
NUM -> 0
NUM -> 1
```

Possible strings generated by this grammar:

```
0, 1,
0+0, 0+1, 1+0, 1+1,
0-0, 0-1, 1-0, 1-1
```

Unreachable Rules

- Sample grammar, now slightly modified
- The nonterminal symbol INCROP can never be reached from EXPR (starting symbol)
  - Thus, rules with INCROP on the left are unreachable rules in this grammar
  - In this homework, you need to remove all the unreachable rules according to grammar and starting point

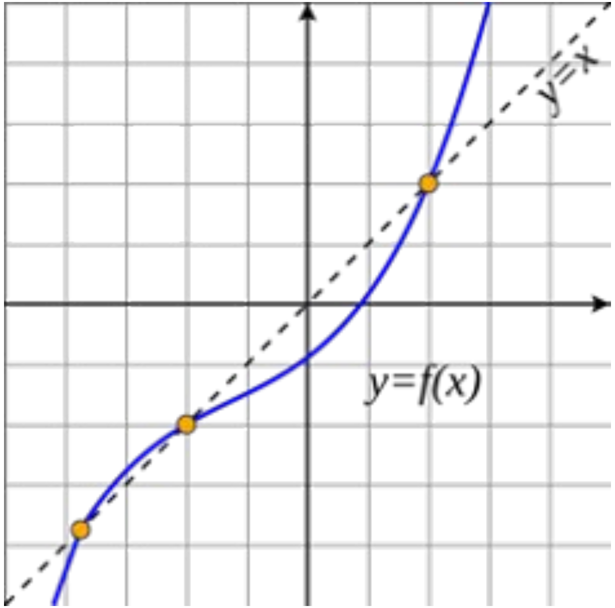
Sample grammar:

```
EXPR -> NUM
EXPR -> NUM BINOP NUM
BINOP -> +
BINOP -> -
NUM -> 0
NUM -> 1
INCROP -> ++
```

HW1 Question Explanation

- Q1-5: Typical set operations, you can look up the definitions
  - Make sure you watch out for if versus iff
- Q6: Fixed point x, where x = f(x)
  - You can assume a fixed point can be obtained by repeatedly applying the function: f(x), f(f(x)), f(f(f(x))), ...
- Q7: A generalization of fixed point to p > 1
- Q8: Related to questions 7 and 8
- Q9: Copy the following definition into your code

```
type ('nonterminal, 'terminal) symbol =
    | N of 'nonterminal
    | T of 'terminal
```



HW1 Tips

- Lists are used to represent sets
    - Sets are a unique collection of elements, but lists can contain duplicates!
- Read through and understand the test cases
    - If problem definition seems unclear, the test cases may help you understand what the expected behavior is
- Read through documentation of `List` and `Stdlib` modules
    - Lots of useful functions in there!
    - Problems 1-8 can be written without too much code if you use `List` and `Stdlib` modules
- You're allowed to (and encouraged to) reuse functions from earlier problems
- All functions must be free of side effects
- Start early, ask questions on Piazza if you get stuck

Thank You