CS131 - Week 1

UCLA Spring 2019
TA: Kimmo Karkkainen

Where to find these slides

- Piazza -> Resources -> Discussion 1C
    - Slides are usually uploaded after the discussion section

Today

- Administration
- Introduction to OCaml
- Homework 1

Me

- TA: Kimmo Karkkainen
- My email: kimmo@cs.ucla.edu
    - (please use Piazza for questions on homework)
- Office hours Wednesday 10:30-12:30 @ Boelter Hall 3256S

Homework

- Working on homework yourself is important to success in CS131!
  - Midterm/final usually have some questions related to homework problems
  - Discussing general ideas is ok, sharing code or details is not
    - **We will check submissions against old solutions**
- Homework will be mostly graded using automated scripts
  - Not compiling -> No credit :(
  - Code must behave exactly as the specs say
  - Function signature must match your function signature, otherwise you will get no credit
- Correctness is more important than performance, but you might lose some points if the solution is really inefficient
  - E.g. a given test case takes an hour to finish
  - Mainly a problem in homeworks 2 (OCaml) and 4 (Prolog)

Homework

- All homework related questions - ask on Piazza
    - Can use private note/question if not sure
    - Questions can be sent anonymously, so that only TAs will see your name
- Homework will be submitted to CCLE

Homework

- First homework due next Tuesday (April 9th) 11:55pm
- **Warning**: Second homework will take significantly more time than the first homework
    - Start working early, even though there is a bit more time reserved for it
- Tentative homework schedule available on course website
- Larger project at the end of the course

Grading

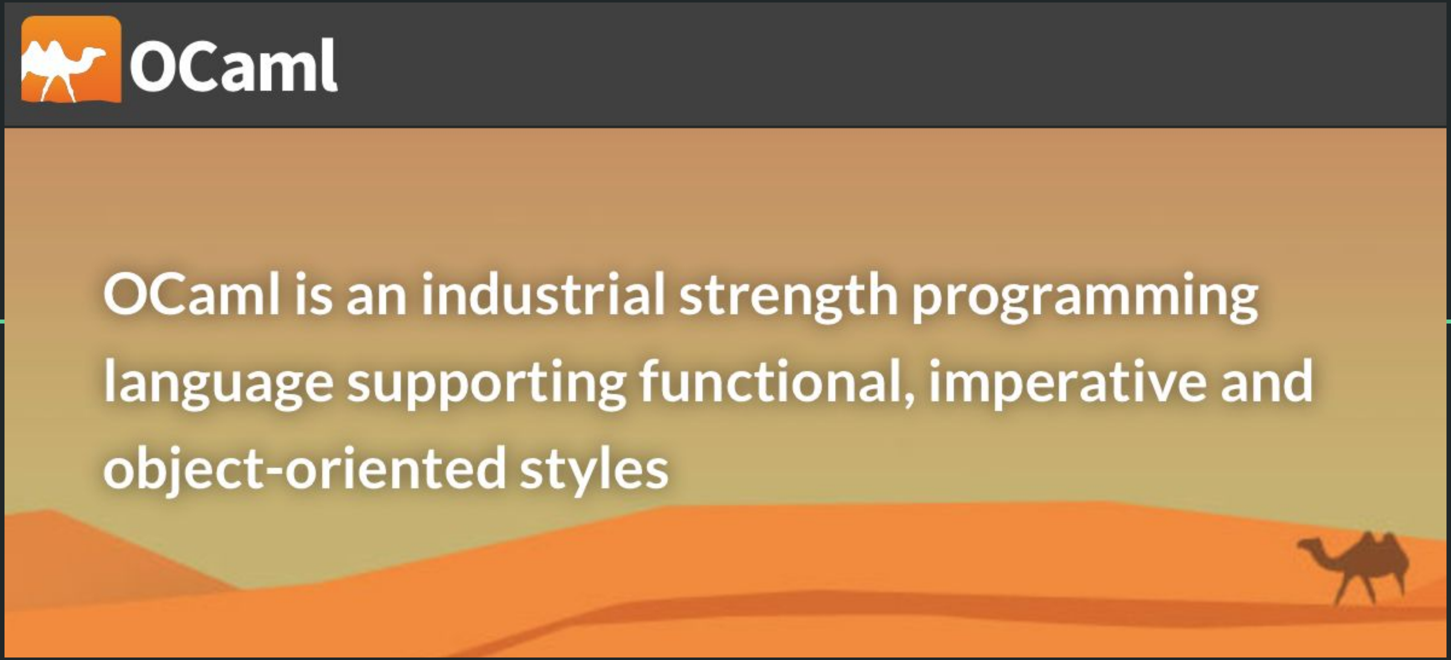
- Homework 32%
  - Each homework has equal weight, project will be worth twice as much
  - Late homeworks will be penalized, penalty doubles every day
    - 1 day = 1%, 2 days = 2%, 3 days = 4%, 4 days = 8%...
- Midterm 24%
- Final exam 44%

Discussion sections

- Main focus is on skills that are needed to solve the homeworks
    - E.g. The basics of programming languages that we use
- Tentative schedule:
    - Week 1: OCaml + HW1
    - Week 2: OCaml + HW2
    - Week 3: OCaml + HW2
    - Week 4: Midterm review
    - Week 5: Java + HW3
    - Week 6: Prolog + HW4
    - Week 7: Scheme + HW5
    - Week 8: Python + Project
    - Week 9: ? + HW6
    - Week 10: Final exam review

Questions about the course?

OCaml programming language



What is Functional Programming?

What is Functional Programming?

- **There are no side effects - variable’s value never changes**
  - No global variables that can be changed from multiple places
  - If you call a function twice with the same same arguments, the output should be the same

Why are we learning this?

- Similar ideas can be found in most modern programming languages, even if they would not be considered functional languages
  - We will see this later when we cover other languages
- Functional programming makes debugging and testing easy
  - Functions will always behave in the same way with the same input, not depending on a global state
- Easy to build scalable systems
  - Distributing code on multiple machines is easier when there is no state that needs to be shared between them
- Many problems can be solved with very little code
  - Less code -> Less bugs

OCaml introduction

- Functional programming language
- Statically typed
  - Every variable has a type, functions define what the types of input parameters should be
  - Compiler/interpreter can warn you about many programming mistakes early on
  - Makes it faster to execute, as there is less need for safety checks when running the code
- Garbage collection
- Compiled
  - But includes an interactive interpreter

OCaml introduction

Companies using OCaml:

facebook

DASSAULT SYSTEMES

Mount Sinai

docker

CITRIX

Jane Street
A very different street

Bloomberg

galois

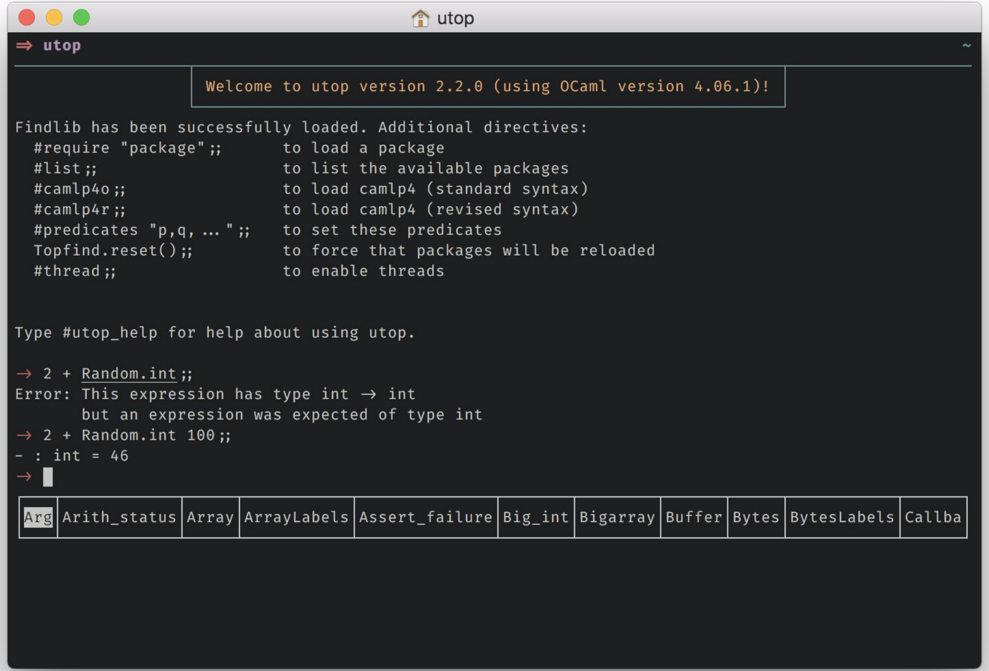
+ many more use other functional programming languages

Installing OCaml

- Installation instructions: https://ocaml.org/docs/install.html
  - Make sure you are using version 4.07.1
- You can use SEASnet servers too
  - lnxsrv06.seas.ucla.edu, lnxsrv07.seas.ucla.edu, lnxsrv09.seas.ucla.edu, and lnxsrv10.seas.ucla.edu
  - If you don't have a SEASnet account, apply for one ASAP: https://www.seas.ucla.edu/acctapp/
  - Make sure that the OCaml version is correct (ocaml --version should show 4.07.1)
    - If not, check that /usr/local/cs/bin is in your path
    - Instructions for this are on the course website under homework #1

Alternative Toplevel - utop

- https://github.com/ocaml-community/utop
- Not necessary, but makes coding a bit nicer



Hello, World!

```
# print_string "Hello, World!\n";;
Hello, World!
- : unit = ()
```

- First part (print_string) is the function that is called, next is argument (“Hello, World\n”)
- Statement ends with **two** semi-colons (;;)
    - Necessary when using the interactive session, not needed in code files
- Next line is printed by the function
- Last line is the return value (unit), which conveys no information in this case

Comments

- (* This is a comment *)
- (* This is
  * a very
  * long
  * comment *)
- (* Nested (* comments *) are allowed too *)

“Variables”

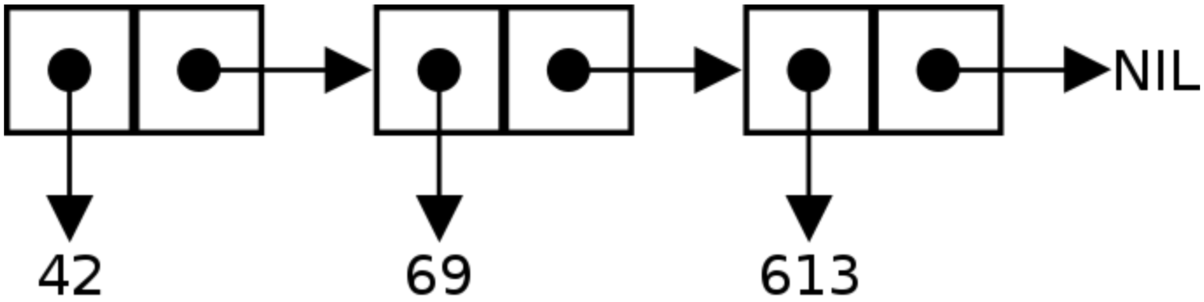
- Not really variables, the value cannot be changed

```
# let my_value = 5;;
val my_value : int = 5
```

- Note: OCaml supports mutable variables too, but they should not be used in the homework
    - The purpose of the homework is to learn how to program using functional paradigm

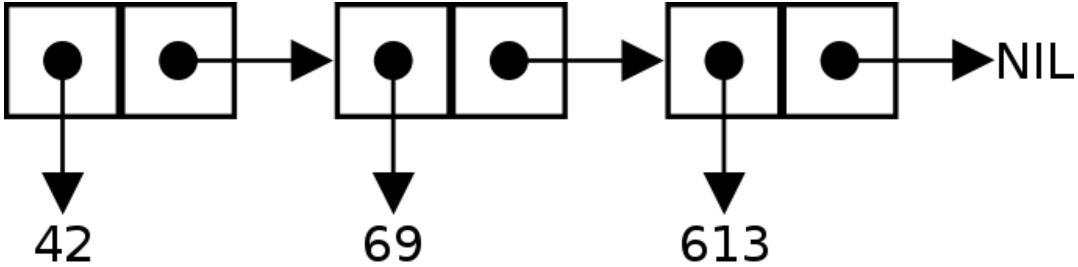
Lists

- Defining a list: *let numbers = [1; 2; 3; 4; 5]*
- All elements have the same type
- Under the hood, lists are immutable singly-linked lists
    - i.e. iterating them is fast, but random access is slow



List operations

- List consists of a head and a tail
  - Accessing these elements can be done with *List.hd* and *List.tl*
- Adding a new element into the beginning of a list is easy:
  - *0 :: [1; 2; 3]* gives us a new list *[0; 1; 2; 3]*
  - *0 :: 1 :: 2 :: [3]* gives us *[0; 1; 2; 3]* as well!
  - :: is right associative, meaning that the previous statement becomes *0 :: (1 :: (2 :: [3]))*
  - *1 :: 2* or *[1] :: [2]* are not valid! Why?
- **Note:** Lists are immutable!
  - Need to create a new list to change any of the values



Functions

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

- *let* binds a function with parameters *a* and *b* to name *average*
- Note that inputs and outputs are inferred to be **integers**
- Assigning a value and defining a function use the same syntax
  - Variables can be thought of as functions that take no input values and return a constant output value

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

Functions

- Calling a function:

```
# let my_average = average 3 5;;
Error: This expression has type int but an expression was expected of type
         float
# let my_average = average 3.0 5.0;;
val my_average : float = 4.
```

- **Note**: The input values are listed without parentheses or commas

Recursive functions

- Recursive functions must be specified explicitly (*let rec*), otherwise the compiler will give an error about an undefined function

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

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

- **Note:** Parentheses necessary around (a-1), otherwise OCaml will try to call factorial a

Defining local variables in functions

- Add keyword *in* after the let statement to make the value available in the following statement

```
# 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 not bound to any name
- Useful when using a function as a function argument
  - Very common in functional programming
  - *“Higher-order function”*

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

Useful list operations - map

- *Map* transforms a list by applying a function on each element

```
# List.map (fun x -> x*x) [1; 2; 3; 4; 5];;
- : int list = [1; 4; 9; 16; 25]
```

Useful list operations - filter

- *Filter* returns a list containing elements that match a given condition

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

Useful list operations - rev

- *Rev* returns a reversed list

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

Useful list operations - for_all

- *For_all* returns true if a condition applies to every element in the list

```
# 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
```

Useful list operations - exists

- *Exists* checks if any element in the list matches a condition

```
# 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
```

List module problems

Solve the following using only List-module functions (e.g. map, filter, for_all, exists, …)

[1; 2; 3; 4; 5] → [2; 3; 4; 5; 6]

List module problems

Solve the following using only List-module functions (e.g. map, filter, for_all, exists, …)

[1;2;3;4;5;6;7;8] → [2;4;6;8]

List module problems

Solve the following using only List-module functions (e.g. map, filter, for_all, exists, ...)

[2;4;6] → True

[2;4;7] → False

Pattern matching

- Some problems can be solved using either conditional statements or pattern matching:

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

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

- More powerful version of the *switch* statement used in some other languages
- Pattern matching allows you to list all the different cases in a clean way
  - Underscore (_) matches any value that does not match the earlier rules
  - Cleaner than conditionals when there is a large number of possible cases
- Compiler lets you know which cases do not match to any of the rules

Pattern matching - Conditions

- Patterns can also include conditions using *when* statement:

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

Pattern matching - Tuples

```
let tuple_matcher x = match x with
| (1, a) -> a
| _ -> 0;

tuple_matcher (1, 5);;
- : int = 5

tuple_matcher (0, 5);;
- : int = 0
```

Data types - Native types

- Native data types in OCaml:
    - int (1, 2, -100)
    - float (1.53, -1053.6)
    - char (‘a’, ‘\n’)
    - string (“foo”, “bar”, “this is a longer text”)
    - bool (true, false)
    - unit (())
        - Means “no value”
        - Usually the return value of functions that have side effects, e.g. print_string

Data types - Native types

- More native data types:
    - list ([1; 2; 3; 4; 5], [“foo”; “bar”])
        - Elements have the same type
        - As mentioned earlier, lists immutable -> list manipulation functions return a new list
    - tuple ((1, 5, “foo”, “bar”))
        - Can combine different data types
        - Accessing elements: fst my_tuple, snd my_tuple
            - No easy access to other elements
        - Less easy to manipulate than lists
        - Very useful with pattern matching
    - functions (let my_fun x = x + 1)

Data types - Own data types

- The most simple use case is to wrap an existing type:

```
type age = int;;
type name = string;;
type person = age * name;;

let print_name (p : person) = match p with
| (p_age, p_name) -> print_string p_name;;

let my_person = (111,"Bilbo" : person);;
print_name my_person;;
Bilbo
```

Data types - Variants

- Used when there are multiple subtypes of one main type

```
type ccle_user =
   Student of string
   | TA of string
   | Professor of string ;;

let user = Professor "Eggert";;

val user : ccle_user = Professor "Eggert"
```

Pattern matching - Types

```
type my_type =
| A of string
| B of int;;

let my_print x = match x with
| A a -> print_string a
| B b -> print_int b;;

my_print (A "some string");;
some string

my_print (B 5);;
5
```

Context-Free Grammars

Context-Free Grammars

- Grammar defines a language
  - What strings are valid in a language
- E.g. We could define a grammar for our own programming language to define what kind of syntax is allowed
  - Grammar does not say what the instructions in that language mean, it just defines what syntax is allowed
  - E.g. We could check that ***print(“Hello World!”)*** is valid, without defining what it does
- There are multiple types of grammars, but for the first homework you only need to know about *Context-Free Grammars*
  - Some other grammars will be covered in the lectures

Context-Free Grammars

- Grammar consists of rules (E.g. *NOUN -> Mary*), non-terminal symbols (E.g. NOUN), and terminal symbols (e.g. mary)
- Grammar has a starting point, in this case PHRASE
- Rules tell us how non-terminal symbols can be replaced
- Possible strings: mary eats; mary drinks; mark eats; mark drinks

Example grammar:

PHRASE -> NOUN VERB
NOUN -> mary
NOUN -> mark
VERB -> eats
VERB -> drinks

Context-Free Grammars

- Let's consider a slightly modified grammar
- The non-terminal symbol ADJECTIVE is not used on the right-hand side of any rule, so it will never be used -> unreachable rule
    - In your homework, you have to remove all the rules that can’t be reached from the starting point using one or more rules

Example grammar:

PHRASE -> NOUN VERB
NOUN -> mary
NOUN -> mark
VERB -> eats
VERB -> drinks
ADJECTIVE -> red

Homework #1

Homework #1

- Deadline next Tuesday (April 9th) at 11:55pm
- See http://web.cs.ucla.edu/classes/spring19/cs131/hw/hw1.html for details

Allowed modules

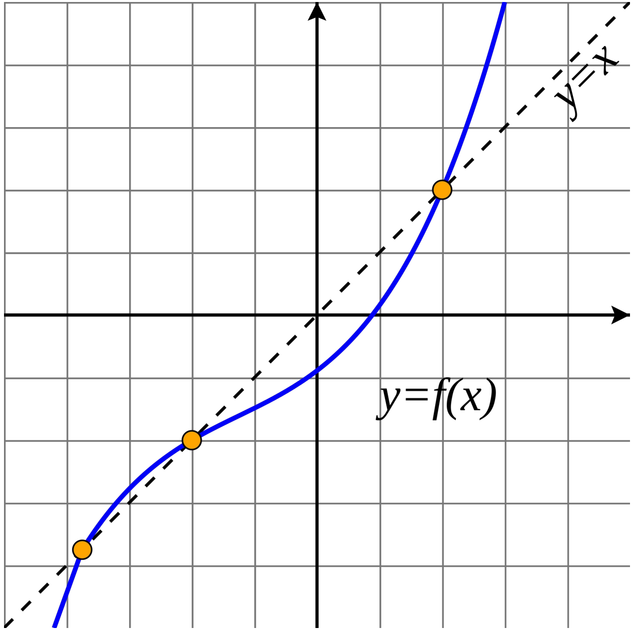
- In the first homework, you are allowed to use two modules: *List* and *Pervasives*
  - **Pervasives** module provides the core functionality of OCaml
    - No need to explicitly import this module
  - **List** module contains functions that are useful when operating with lists
    - This module is not imported by default!
- Before you can call a function that is in a module, you need to import it:
  open List
- Alternatively, you can add the module name into the function call,
  e.g. List.filter

Homework #1

1. Write a function to determine if one list is a subset of another list
   - i.e. is every element of list *a* also in list *b*
2. Write a function to determine if two sets are equal
   - Both should contain the same elements
3. Write a function that returns the union of two sets
   - A set that has every element that exists in either set or in both of them
4. Write a function that returns the intersection of two sets
   - A set that contains every element that is in both of the given sets
5. Write a function that returns the difference of two sets
   - All elements that belong to the first set but do not belong to the second set

Homework #1

6. Write a function that returns the computed fixed point of a given function
   - Value x where f(x) = x



Computed fixed point can be found by testing iteratively f(x), f(f(x)), f(f(f(x))), and so on

- Some functions do not have a fixed point
- This technique does not always find fixed points

Homework #1

7. Write a function that takes a grammar as its input and returns a grammar where all the unreachable rules have been removed
   - Recommended reading: https://en.wikipedia.org/wiki/Context-free_grammar
8. Write at least one test case for each of the previous functions

Homework #1 - Submission

- You are expected to submit 3 files:
    - hw1.ml - The functions that you implemented
    - hw1test.ml - Test cases for your functions
    - hw1.txt - Written assessment of how you ended up solving the problems the way you did
        - See course website for details

HW1 - Note

- Copy the type definition into your code file

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

Homework #1 - How to get started

- Read and understand the given test cases
  - If the problem definition seems unclear, the test cases might help you understand how your code should behave
- Read through the documentation for *List* and *Pervasives* modules
  - Most problems can be solved with very little code if you don't reinvent the wheel
- Think whether the functions that you wrote for earlier problems can be used to solve the later problems
  - The power of functional programming comes from reusing very simple functions to implement more powerful functions
- **Make sure your solution works on SEASnet servers!**

Helpful resources

- OCaml tutorial https://ocaml.org/learn/tutorials/
- Try OCaml https://try.ocamlpro.com
    - Interactive tutorial in your browser
    - Covers some topics that are not used in this course

Questions?

- Piazza
  - Fastest way to get answers
  - TAs and your classmates can answer your question, so this is the best channel to get help when you're stuck
- Come to office hours
  - My office hour is Wednesday 10:30-12:30, others will be posted on CCLE
- Come ask after the discussion sections