CS131: Programming Languages

Week #0
ROYCE 164
Seunghyun Yoo

TA

Seunghyun Yoo (shyoo1st@cs.ucla.edu)

Office hours (BH 2432)

- Wed 5:30 pm - 6:30 pm
- Thu 1:30 pm - 2:30 pm
- By appointment

Class website: http://web.cs.ucla.edu/classes/fall15/cs131/

Grading

Homework Assignments	40%
Midterm	20%
Final exam	40%

HW 1 and 2 - **OCaml**
HW 3 - **Java**
HW 4 - **Prolog**
HW 5 - **Scheme**
Project - **Python**
HW 6 - To be determined...

All homeworks are weighted **equally**, except for the project (worth **double**).

First assignment (OCaml)

- Due date: **10/2 Fri 11:55 pm**
- Submission: **CCLE** (will be updated soon)

SEASnet account

- Student Account Application: http://www.seas.ucla.edu/acctapp/
- Create an account as soon as possible!

Piazza

- We’ll be using Piazza predominantly to answer questions
- Join @ piazza.com/ucla/fall2015/cs131
    - Link on course webpage
- If you have a question, please post on Piazza!

Today

- OCaml basics
- HW1 - Fixpoints and grammar filters

OCaml

- A programming language supporting **functional**, imperative and object-oriented style.
- On SEASnet server, OCaml 4.02.3 is already installed.
  - Might have to add it to your path in the *bash* profile manually.
- To install on your local machine, visit ocaml.org

Functional Programming

- It treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. -- *from wikipedia*

  ```
  # let rec fact x =
      if x <= 1 then 1 else x * fact (x - 1);;

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

- Why is it good?

OCaml: Get started

- Use an interactive shell, which is called “toplevel”
  - Type `ocaml` in the command line
- How to load your OCaml source file?
  - # `#use “file-name.ml”;;`
- Redirection operator
  - $ `ocaml < fact.ml`

OCaml: How to define a function?

```
# let square x = x * x;;
val square : int -> int = <fun>
# square 3;;
- : int = 9

# let add x y = x + y;;
val add : int -> int -> int = <fun>
# add 1 2;;
- : int = 3

(cf) # let add (x, y) = x + y;;
val add : int * int -> int = <fun>
```

```
# let square = fun x -> x * x;;

# let square = function
    | x -> x * x;;

# let square x = match x with
    | x -> x * x;;
```

OCaml: if and match statement

```
# let max a b =
    if a > b then a else b;;
val max : ‘a -> ‘a -> ‘a = <fun>

# let eval_op op v1 v2 =
    match op with
    | “+” -> v1 + v2
    | “-” -> v1 - v2
    | “*” -> v1 * v2
    | _ -> failwith (“undefined”);;
val eval_op : string -> int -> int -> int = <fun>
```

OCaml: List

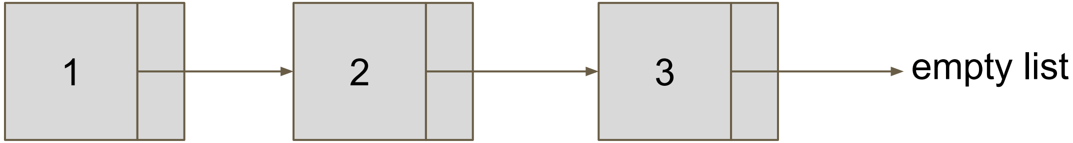
- An immutable, finite sequence of element of the same type

```
# [ 1; 2; 3 ];;
- : int list = [1; 2; 3]

# 1 :: (2 :: (3 :: [])) ;;
- : int list = [1; 2; 3]

# 1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]
```

OCaml: List operations

- :: operator

  (ex) `1 :: 2 :: 3 :: []`



- Append
  - `List.append [1;2;3] [4;5;6];;`
  - `[1;2;3] @ [4;5;6];;`

OCaml: Extract data from a list

- We can read data out of a list using a **match** statement.
- The keyword **rec** stands for “recursion.”

```
# let rec sum l =
  match l with
  | [] -> 0
  | h :: t -> h + sum t;;
val sum : int list -> int = <fun>

# sum [1;2;3];;
- : int = 6

# sum [];;
- : int = 0
```

OCaml: Type

- `type name = typedef;;`
- `type ‘a name = typedef;;`
  - Type declarations can be parameterized by type variables.

(ex)

- `# type ‘param paired_with_integer = int * ‘param;;`
  - `type ‘a paired_with_integer = int * ‘a`
- `# type specific_pair = float paired_with_integer;;`
  - `type specific_pair = float paired_with_integer`

OCaml: Equal and Identical operator

```
let a = [ 1; 3 ];;
let b = [ 1; 3 ];;
```

- `a = b ?`
- `a == b ?`
- `a == a ?`

OCaml: Arithmetic operators

- Must specify the type of number explicitly
    - 1 + 0.5 ;; -- is not allowed (type mismatch)
    - 1. + 0.5 ;; -- is not allowed (operator is only for integer)
    - 1. +. 0.5 ;; -- correct

- +. -. *. /.   for float numbers

OCaml: Tail Recursion

-
  ```
  let rec make_list n =
    if n = 0
      then []
      else n :: make_list (n-1);;   <- a reversed list
  ```

-
  ```
  let big_list = make_list 10000000;;
  ```
  - Stack overflow during evaluation

VS.

-
  ```
  let rec make_list n list =
    if n = 0
      then list
      else make_list (n-1) (n::list);;   <- fixed the typo
  ```

-
  ```
  let big_list = make_list 10000000 [];;
  ```

-
  ```
  let rec make_list n =
    if n = 0 then []
    else make_list (n-1) @ [n];;
  ```

```
OUTPUT
make_list 5;;
  -    : int list = [1; 2; 3; 4; 5]
```

Exercise: Reverse list

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

let lst = [1 ; 2 ; 3];;

‘h’ will be 1 (integer) and ‘t’ will be [2; 3] (integer list)

(Basic idea)
Let’s assume that the function ‘reverse’ works.
Then, `reverse [2;3]` will result out `[3;2]`.
We solve the problem by specifying what the problem is and what the relationship between the original problem and its subproblems.

Function Currying

- Don’t have to pass every argument to a function
- Passing fewer arguments will return a function with the remaining args

```
let add x y = x + y;;

let add2to_x x = add x 2;;
let add2to_y y = add 2 y;;
let add2to_y = add 2;;
```

```
Type of add:  int -> int -> int
        add2:         int -> int
```

OCaml: misc.

- Comments

```
(* hello world *)
```

- Trace function call
    - `#trace <function name>;;`

HW: Todo

hw1.ml

- subset a b
- equal_sets a b
- set_union a b
- set_intersection a b
- set_diff a b
- computed_fixed_point eq f x
- computed_periodic_point eq f p x
- filter_blind_alleys g

hw1test.ml

- (at least one test case for each of these functions)
  - (ex) my_subset_test0, my_subset_test1, ...

hw1.txt

- (a report)

subset & equal_sets

- **`subset`**
  - A set is a subset of itself
  - The empty set is a subset of any set
- **`equal_sets`**
  - Must use set semantics
  - it is not just "(=) a b"
  - [3;1], [1;3], [1;3;3] are equal

Fixed point

Definition

- A <u>fixed point</u> is a point `x` such that `f x = x`
  - In OCaml, parentheses are not needed around arguments. The actual meaning is f(x) = x.
- Computed fixed point
  - A fixed point of `f` computed by calculating `x, f x, f (f x), f (f (f x)),` …

Fixed point: example

```
let div2 x = x / 2;;
- # div2 8;;
- : int = 4
- # div2 (div2 8);;
- : int = 2
- # div2 (div2 (div2 8));;
- : int = 1
- # div2 (div2 (div2 (div2 8)));;
- : int = 0
- # div2 (div2 (div2 (div2 (div2 8))));;
- : int = 0
```

Periodic point

Definition

- Periodic point
  - A point x such that f (f … (f x)) = x, where there are p occurrences of f in the call.
- Computed periodic point
  - A period point of f with period p, computed by calculating x, f x, f (f x), … , stopping when a periodic point with period p is found for f.

Periodic point: example

```
# let f x = x *. x -. 1.;;
val f : float -> float = <fun>
# f (f (f (f (f (f (f 0.5))))));;
- : float = -0.949233276114730073
# f (f (f (f (f (f (f (f 0.5)))))));;
- : float = -0.0989561875164966
# f (f (f (f (f (f (f (f (f 0.5))))))));;
- : float = -0.990207672952199913
# f (f (f (f (f (f (f (f (f (f 0.5)))))))));;
- : float = -0.0194887644265890891
…
# f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f 0.5))))))))))))))));;
- : float = -1.
# f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f 0.5)))))))))))))))));;
- : float = 0.
# f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f 0.5))))))))))))))))));;
- : float = -1.
```

Precision, Infinity and nan

When computing the fixed point of ‘sqrt’,

`# sqrt 1.00000000000000001 = 1;;` -- because of the finite precision

- Infinity
    - infinity *. 2. = infinity;; -- ?
    - infinity /. infinity;; -- ?
- nan (not a number)
    - nan = nan;; -- ?
    - nan *. 2. = nan;; -- ?

Grammar

Symbol → nonterminal or terminal

Right hand side → a list of symbols.

Rule → a pair of (nonterminal, right hand side)

Grammar → a pair of (a start nonterminal symbol, a list of rules)

filter_blind_alleys

- Filter out rules that are impossible to derive a terminal string.
  - Some rules cannot have nonterminals completely substituted out
- Useless rules

```
Expr, [T"("; N Expr; T")"];
Expr, [N Num];
Expr, [N Expr; N Binop; N Expr];
Expr, [N Lvalue];
Expr, [N Incrop; N Lvalue];
Expr, [N Lvalue; N Incrop];
Lvalue, [T"$"; N Expr];
```

```
~~Expr, [T"("; N Expr; T")"];~~
~~Expr, [N Num];~~
Expr, [N Expr; N Binop; N Expr];
Expr, [N Lvalue];
Expr, [N Incrop; N Lvalue];
Expr, [N Lvalue; N Incrop];
Lvalue, [T"$"; N Expr];
```

(Another example)

```
Expr, [N Num];
Expr, [N Lvalue];
Expr, [N Expr; N Lvalue];
Expr, [N Lvalue; N Expr];
Expr, [N Expr; N Binop; N Expr];
Lvalue, [N Lvalue; N Expr];
Lvalue, [N Expr; N Lvalue];
Lvalue, [N Incrop; N Lvalue];
Lvalue, [N Lvalue; N Incrop];
Incrop, [T"++"];
Incrop, [T"--"];
Binop, [T"+"];
Binop, [T"-"];
Num, [T"0"];
Num, [T"1"];
Num, [T"2"];
Num, [T"3"];
Num, [T"4"];
Num, [T"5"];
Num, [T"6"];
Num, [T"7"];
Num, [T"8"];
Num, [T"9"]]
```

```
Expr, [N Num];
Expr, [N Lvalue];
Expr, [N Expr; N Lvalue];
Expr, [N Lvalue; N Expr];
Expr, [N Expr; N Binop; N Expr];
Lvalue, [N Lvalue; N Expr];
Lvalue, [N Expr; N Lvalue];
Lvalue, [N Incrop; N Lvalue];
Lvalue, [N Lvalue; N Incrop];
Incrop, [T"++"];
Incrop, [T"--"];
Binop, [T"+"];
Binop, [T"-"];
Num, [T"0"];
Num, [T"1"];
Num, [T"2"];
Num, [T"3"];
Num, [T"4"];
Num, [T"5"];
Num, [T"6"];
Num, [T"7"];
Num, [T"8"];
Num, [T"9"]]
```

These are blind alley rules.