cs131.week#1

If programming languages were cars...

http://users.cms.caltech.edu/~mvanier/hacking/rants/cars.html
http://perevodik.net/en/posts/39/

Fortran

Sturdy and dependable, though not particularly maneuverable or sexy.

C

Still the best systems programming language 40 years later.

C++

Because C wasn't tricked out enough.

Homework 3

Java

Not pretty or fun to drive, but solid.

C#

Based on Java and slightly more modern, but in some ways less solid.

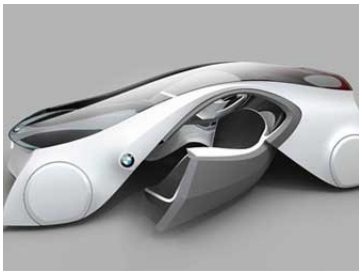
Ruby

Similar to Python in many ways, though arguably slightly more maneuverable and less elegant.

Homework 1 & 2

OCaml

A very sexy European car, never breaks down, and you end up going further in less time. However, because it's French, none of the controls are in the usual places.



Haskell

How do I drive this thing? Despite being older than many languages on the list, Haskell is more modern in most ways.

Project (tentatively)

Python

A stylish ride, though not the fastest or most maneuverable.

Visual Basic

Loads of fun to drive endlessly around the Microsoft app track.

Homework 5

Lisp/Scheme

After all these years, no other car has vertical doors. And, with enough tweaking, you can turn it into a pretty effective airplane or submarine.

Homework 4

Prolog

A self-drive car: you tell it what your destination looks like, and it does all the driving for you. With a trial-and-error GPS system, it goes down the road looking for your destination until you get where you need to go.

<table>
<tr><td rowspan="2">MIDTERM</td><td>Ocaml Homeworks</td><td>Java and Prolog</td><td>Scheme and Homework 6</td><td>PROJECT (Python?)</td></tr>
<tr><td colspan="4">FINAL</td></tr>
</table>

TA

- Thuy Vu
- thuyvu@cs.ucla.edu
- Office hours
  - Wednesdays 09:00–11:00 in Boelter 2432
  - By appointments

"I got a bug…"

1. Try to fix it
2. Google it
3. Explain the problem to the TA and what you found in the first two steps
4. If the TA doesn't respond within 36h, send him a reminder

Today

- OCaml
- Grammar Filters

OCaml: add two numbers

- Add 2 numbers

```
let addxy x y = x + y;;
```

  - val addxy : int -> int -> int = <fun>
- Increase by 3

```
let add3 x = x + 3;;
```

  - val add3 : int -> int = <fun>

Or

```
let add3 x = addxy 3;;
let add3   = addxy 3;;
```

OCaml: curry function

- *f(x,y) = 2*x + y*

```
let fxy x y = 2*x + y;;
```

  - val fxy: int -> int -> int = <fun>
- *f(3,y)*

```
let f3y = fxy 3;;
```

  - val f3y : int -> int = <fun>
- *f(x,3)*

```
let fx3 x = fxy x 3;;
```

  - val fx3 : int -> int = <fun>

OCaml: write a function

- `let max (a,b) = if a > b then a else b;;`
- ```
  let max = function
    | (a,b) -> if a > b then a else b;;
  ```
- ```
  let max atuple = match atuple with
    | (a,b) -> if a > b then a else b;;
  ```
- `let max = fun (a,b) -> if a > b then a else b;;`

val max : 'a * 'a -> 'a = <fun>

OCaml: recursive - factorial

- ```
  let rec fact n =
    if n=0 then 1 else n * fact(n-1);;
  ```
- ```
  let rec fact = function
    | 0 -> 1
    | n -> n * fact(n-1);;
  ```

val fact : int -> int = <fun>

OCaml: pattern match & list

- ```
  let rec sumlist alist = match alist with
    | [] -> 0
    | head::tail -> head + sumlist tail;;
  ```

val sumlist : int list -> int = <fun>

OCaml: last element

- ```
  let rec last = function
    | [a] -> a
    | _::tail -> last tail
  ```

```
Warning 8: this pattern-matching is not
exhaustive. Here is an example of a value
that is not matched: []
```

val last : 'a list -> 'a = <fun>

OCaml: concat two lists

- `let concat a b = a @ b;;`

val concat: 'a list -> 'a list -> 'a list = <fun>

OCaml: reverse a list

- ```
  let rec reverse = function
    | [] -> []
    | head::tail -> (reverse tail)@[head]
  ```

val reverse : 'a list -> 'a list = <fun>

Homework 1

1. `subset`
2. `proper_subset`
3. `equal_sets`
4. `set_diff`
5. `computed_fixed_point`
6. `computed_periodic_point`
7. `filter_blind_alleys`

`computed_fixed_point`

- fixed point (of a function `f`) is a point `x` such that `f x = x`
- a fixed point of `f` computed by calculating `x, f x, f (f x), f (f (f x))...` stopping when a fixed point is found for `f`
- ```
  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
  ```

computed_periodic_point

- periodic point of a function `f` with period `p` is a point `x` such that

  `f (f ... (f x)) = x`
  - `p` occurrences of `f` in the call
  - a periodic point is like a fixed point, except the function returns to the point after p iterations instead of 1 iteration
- For example:

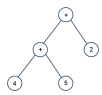
```
- # let f x = x *. x -. 1.;;
    val f : float -> float = <fun>
- # f (f (f (f (f (f 0.5)))));;
  - : float = -0.949233276114730073
- # f (f (f (f (f (f (f 0.5))))));;
  - : float = -0.0989561875164966
- # f (f (f (f (f (f (f (f 0.5)))))));;
  - : float = -0.990207672952199913
- # f (f (f (f (f (f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

computed_periodic_point

```
- # f (f (f (f (f (f (f (f (f (f 0.5)))))))));;
  - : float = -0.999620188061125
- # f (f (f (f (f (f (f (f (f (f 0.5))))))))));;
  - : float = -0.000759479620641156927
- # f (f (f (f (f (f (f (f (f (f 0.5)))))))));;
  - : float = -0.999999423190705783
- # f (f (f (f (f (f (f (f (f (f (f 0.5))))))))));;
  - : float = -1.15361825570037269e-06
- # f (f (f (f (f (f (f (f (f (f (f (f 0.5)))))))))));;
  - : float = -0.999999999998669176
- # f (f (f (f (f (f (f (f (f (f (f (f (f 0.5))))))))))));;
  - : float = -2.66164867923635029e-12
- # 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 0.5))))))))))))));;
  - : float = 0.
- # f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f 0.5)))))))))))))));;
  - : float = -1.
```

filter_blind_alleys

- Symbol
  - Terminal
  - Non-terminal
- Rule is a pair
  - Left hand side: non-terminal
  - Right hand side: list of symbols
- Grammar consists of
  - A non-terminal start symbol
  - A list of rule
- Parsing (hw2)
  - Given a string: 4 + 5 + 2
  - Find a derivative tree
  - Leaves are terminals
- Blind-alley rules
  - Impossible to derive a terminal string
  - 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];
• 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"]]
```

filter_blind_alleys

```
• 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];
• 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"]]
```