```
TA:
    Jason Mao
    jmmp8@cs.ucla.edu
    W 12:30-2:30 CS 2432


we're used to imperative languages
    telling the computer to do stuff at each line

imperative languages (like C++ or Java) have support for
    passing functions

OCaml : functional programming language
    funtions are another type of data (can pass them like ints, floats, etc)

    at OCaml website, go to learn section, the other tutorial is not great apparently


    unit is kinda like a void

    ( don't have to declare types)
        ints and floats are not very compatile without casts
        add ints with +
        add floats with +.

    OCaml variables are more like references (like Java and Python)
        not real variables like C

    in interpreter
        let x = 5;; (*for int*)
        let x = 5.0;; (*for float*)

        functions
        function g with parameter x

        let g x = x + 5 ;;
        will return what type it is
        int -> int

        if you return a constant let g x = 5 ;;
        then: a -> int

        OCaml returns the value of the last statement executed in the program

        function call
        g 6 ;;
```

```
    don't need paratheses (style guide doesn't like them)


    let g x y = x + y ;;
    int -> int -> int

        takes two integers returns an integer
        currying:
            not make a function that returns two parameters
                functions meant to take a single parameter
            takes a function that returns and int, returns a
            function that takes an int and returns and int


    let g f x = f x ;;
        ('a -> 'b) -> 'a -> 'b
        a and b are template parameters

for nested function calls probably use parantheses
    parantheses indicate a single argument DO NOT PARATHENSIZE THE ENTIRE LIST

recursive functions

    let rec g x = g (x-1) ;;
    not a good function, just an example


records ( like a struct ) :

type s  =
    {   x : int ;
        y : int ;
    };;

equivalent in C to
struct S {
    int x;
    int y;
};


Recall enumerations :
    Enum Color {
        Blue;
        Red;
```

```
        Yellow;
    }

    Color x = Blue;


Variants:
type basic_color =
    | Blue | Red | Yellow ;;

    first letter must be capatalized (because no idea)
    first letter of variant name must be lower case

list syntax : (singly linked list, must all be of the same type)
    [ element1; element2; element3 ] ;;

tuple : sequence of objects that do not need to be of the same type

ie tuple of three ints:
    int * int * int
    (5,6,5)

kinda acts like a Union too

type color =
    | Basic of basic_color * weight
    | RGB of int * int * int
    | Gray of int
;;

[RGB (250, 70, 70); Basic (Green, Regular)];;


can parametrize variant, see the homework spec

, indicates or (as below)
type ('nonterminal, 'terminal) =


more on lists
    have to be of same type

[5;6;5]

E::L
    appends E to head of L
```

```
    checking if list contains 5:

    let rec g l =
        match l with
        | [] -> false
        | h::t -> if h=5 then true else g t  (*any list of one or more elements*) SINGLE EQUALS IS TO CHECK EQUIVALENCE
    ;;


    problem 1 from hw 1

    mem is a function in list module (can also write a contains function as in above)

    let rec subset a b =
        match a with
        | [] -> true
        | h::t -> if mem h b then subset t b else false
    ;;



    modules :
        open List ;;


    operators are just functions
    (+ 5) is a function
    ((=) 5)


    problem 2 from hw 1
        just do subset back and forth

        let seteq ab =
            (subset a b) && (subset b a)


to test
    # use "name.ml";;  (*that means actually write a # into interpreter*)


go to OCaml websit and find the cheat sheets


Expr, [N Expr; N Binop; N Expr];
```

```
    => Expr -> E B E

Blind Alley : if you follow them, you can never resolve your work
    ie it can never reach a terminal

start with terminals and work upward (instead of down)
    can mark all the good rules
    whatever is left are blind alley rules
```