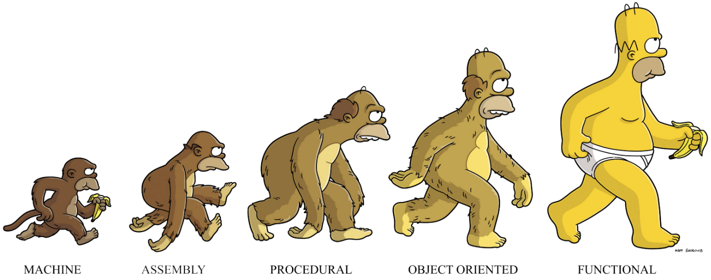
# COMP302: Programming Languages and Paradigms

Prof. Brigitte Pientka (Sec 01)
`bpientka@cs.mcgill.ca`

Francisco Ferreira (Sec 02)
`fferre8@cs.mcgill.ca`

School of Computer Science
McGill University

Week 1-2, Fall 2017



## Functional Tidbit: Haskell B. Curry

- Logician and Mathematician
- 12 Sept 1900 – 1 Sept 1982
- Most known for the Curry-Howard-Isomorphism i.e.direct relationship between programs and proofs
- Prog. language Haskell is named after him.

## What is OCaml?

Statically Typed Functional Programming Language

## What is OCaml?

Statically Typed Functional Programming Language

- Types approximate runtime behaviour
- Analyze programs **before** executing them
- Find and fix bugs before testing

## What is OCaml?

Statically Typed Functional Programming Language

- Types approximate runtime behaviour
- Analyze programs **before** executing them
- Find and fix bugs before testing

- Primary expressions are functions!
- Functions are first-class!
- Pure vs Not Pure
- Call-By-Value vs Lazy

## Concepts for Today

- Writing and executing basic expressions
- Learn how to read error message
- Names, Values, Basic Types
- Variable, Bindings, Scope of Variables
- Simple Functions

## Step 1:Basic Expressions

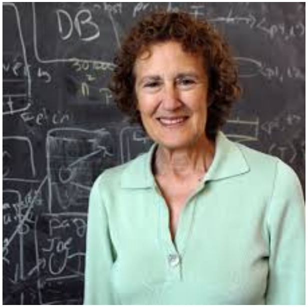
DEMO

## Step 2:Variables and Bindings

## Step 2:Variables and Bindings

- Variable binding is not an assignment
- Variables cannot be updated – we can only overshadow a previous binding
- Variable bindings persist
- Garbage collection disposes off variable bindings that are not needed anymore
- Variable bindings are local – they exist within a scope
- Variables are bound to a value – not an expression

## Function Tidbit: Barbara Liskov



- Professor at MIT
- John von Neumann Medal [2014]
- Turing Award for her work in the design of programming languages and software methodology [2008]

*"The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal."* *B. Liskov [1974]*

## Function Tidbit: Barbara Liskov (in the 70's)

## Functions

## Functions

- Functions are values
- Function names establish a binding of the function name to its body

```
let area (r: float) = pi *. r *. r;;
```

## Functions

- Functions are values
- Function names establish a binding of the function name to its body

```
let area (r: float) = pi *. r *. r;;
```

- Recursive functions are declared using the keyword `let rec`

```
let rec fact n =
  if n = 0 then 1
   else n * fact (n-1)

```

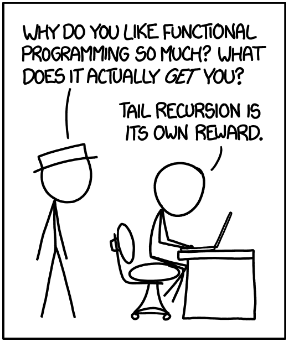
## Functions

DEMO

## Tail-recursive Functions

*A function is said to be "tail-recursive", if there is nothing to do except return the final value. Since the execution of the function is done, saving its stack frame (i.e. where we remember the work we still in general need to do), is redundant.*

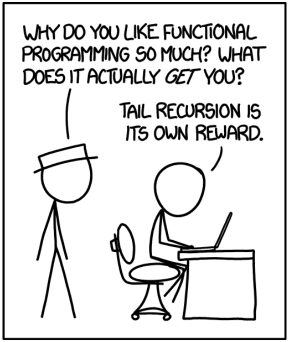
- Write efficient code
- All recursive functions can be translated into tail-recursive form!


WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY *GET* YOU?
TAIL RECURSION IS ITS OWN REWARD.

## Tail-recursive Functions

*A function is said to be "tail-recursive", if there is nothing to do except return the final value. Since the execution of the function is done, saving its stack frame (i.e. where we remember the work we still in general need to do), is redundant.*

- Write efficient code
- All recursive functions can be translated into tail-recursive form!


WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY *GET* YOU?
TAIL RECURSION IS ITS OWN REWARD.

## Example: Rewriting Factorial

## Example: Rewriting Factorial

```
let rec fact_tr1 n =
  let rec f (n, m) =
    if n=0 then
      m
    else f(n-1, n*m)
  in
   f(n,1)
```

- Second parameter to accumulate the result; in the base case we simply return its result
- Avoids having to return a value from the recursive call and subsequently doing further computation.
- Avoids building up a runtime stack to memoize what needs to be done once the recursive call returns a value

## Example: Rewriting Factorial

```
let rec fact_tr1 n =
  let rec f (n, m) =
    if n=0 then
      m
    else f(n-1, n*m)
  in
   f(n,1)
```

- Second parameter to accumulate the result; in the base case we simply return its result
- Avoids having to return a value from the recursive call and subsequently doing further computation.
- Avoids building up a runtime stack to memoize what needs to be done once the recursive call returns a value

What is the type of `f`?

## Passing Arguments

- Passing all arguments at the same time

```
'a * 'b -> 'c
```

- Passing one argument at a time

```
'a -> 'b -> 'c
```

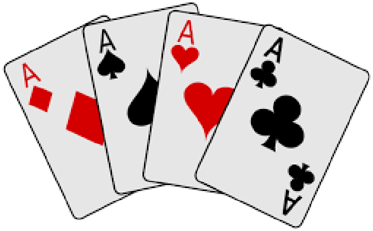
- **Remark:** We can translate any function of type `'a -> 'b -> 'c` to a function of type `'a * 'b -> 'c` and vice versa. This is called *currying* (*uncurrying* resp.)

## Topic

Data Types and Pattern Matching

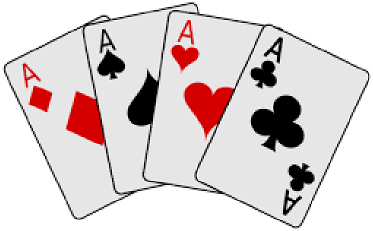
## Playing Cards

How can we model a collection of cards?



## Playing Cards

How can we model a collection of cards?



Declare a new type together with its elements

```
type suit = Clubs | Spades | Hearts | Diamonds
```

## User-Defined (Non-Recursive) Data Type

```
type suit = Clubs | Spades | Hearts | Diamonds
```

## User-Defined (Non-Recursive) Data Type

```
type suit = Clubs | Spades | Hearts | Diamonds
```

- The order in which we declare these elements does not matter

## User-Defined (Non-Recursive) Data Type

```
type suit = Clubs | Spades | Hearts | Diamonds
```

- The order in which we declare these elements does not matter
- We call Clubs, Spades, Hearts, Diamonds constructors.

## User-Defined (Non-Recursive) Data Type

```
type suit = Clubs | Spades | Hearts | Diamonds
```

- The order in which we declare these elements does not matter
- We call Clubs, Spades, Hearts, Diamonds constructors.
- Constructors must begin with a capital letter in OCaml.

## User-Defined (Non-Recursive) Data Type

```
type suit = Clubs | Spades | Hearts | Diamonds
```

- The order in which we declare these elements does not matter
- We call Clubs, Spades, Hearts, Diamonds constructors.
- Constructors must begin with a capital letter in OCaml.
- Use pattern matching to analyze elements of a given type.

```
match <expression> with
        | <pattern> -> <expression>
        | <pattern> -> <expression>
            ...
        | <pattern> -> <expression>
```

A pattern is either a variable, underscore (wild card), or a constructor.

## Comparing Suits

Write a function dom of type suit $*$ suit $->$ bool

dom(s1,s2) $=$ true iff suit s1 beats or is equal to suit s2 relative to the ordering Spades $>$ Hearts $>$ Diamods $>$ Clubs

## Comparing Suits

Write a function dom of type $\mathsf{suit} * \mathsf{suit} \; -> \mathsf{bool}$

$\mathsf{dom(s1,s2)} = \mathsf{true}$ iff suit s1 beats or is equal to suit s2 relative to the ordering

Spades $>$ Hearts $>$ Diamods $>$ Clubs

Demo