Introduction to OCaml
Vastly Abbreviated FP Genealogy

- LCF Theorem Prover (70s)
  - Edinburgh ML
    - Miranda (80s)
    - Haskell (90s - now)
    - Standard ML (90s - now)
    - OCaml (90s - now)
  - Coq (80s - now)
  - Scala (00s - now)
  - F# (now)

- LISP (50s-now)
  - Scheme (70s-now)
  - Racket (00s-now)

Languages:
- Lazy
- Call-by-value
- Typed, polymorphic
- Untyped
- Dependently typed
But Why Functional Programming Now?

• Functional programming will introduce you to new ways to think about and structure your programs:
  – new reasoning principles
  – new abstractions
  – new design patterns
  – new algorithms
  – elegant code

• Technology trends point to increasing parallelism:
  – multicore, gpu, data center
  – functional programming techniques such as map-reduce provide a plausible way forward for many applications
Functional Languages: Who’s using them?

- Google: map-reduce in their data centers
- Facebook: Scala for correctness, maintainability, flexibility
- Microsoft: F# in Visual Studio
- Jane Street: Haskell for concurrency, Haskell for managing PHP
- Barclays: Coq proof of 4-color theorem
- Haskell: for specifying equity derivatives
- Mathematicians: Coq proof of 4-color theorem

Related links:
- www.artima.com/scalazine/articles/twitter_onScala.html
- http://www.haskell.org/haskellwiki/Haskell_in_industry
Functional Languages: Join the crowd

- Elements of functional programming are showing up all over
  - **F#** in Microsoft Visual Studio
  - **Scala** combines ML (a functional language) with Objects
    - runs on the JVM
  - **C#** includes “delegates”
    - delegates == functions
  - **Python** includes “lambdas”
    - lambdas == more functions
  - **Javascript**
    - find tutorials online about using functional programming techniques to write more elegant code
  - **C++** libraries for map-reduce
    - enabled functional parallelism at Google
  - **Java** has generics and GC
  - ...

Thinking Functionally

In **Java** or **C**, you get (most) work done by *changing* something

```java
temp = pair.x;
pair.x = pair.y;
pair.y = temp;
```

commands *modify* or *change* an existing data structure (like pair)

In **ML**, you get (most) work done by *producing something new*

```ml
let (x,y) = pair in (y,x)
```

you *analyze* existing data (like pair) and you *produce* new data (y,x)
This simple switch in perspective can change the way you think about programming and problem solving.
Thinking Functionally

pure, functional code:

```javascript
let (x,y) = pair in (y,x)
```

- outputs are everything!
- output is function of input
- data properties are stable
- repeatable
- parallelism apparent
- easier to test
- easier to compose

imperative code:

```javascript
temp = pair.x;
pair.x = pair.y;
pair.y = temp;
```

- outputs are irrelevant!
- output is not function of input
- data properties change
- unrepeatable
- parallelism hidden
- harder to test
- harder to compose
Why OCaml?

Small, orthogonal core based on the *lambda calculus*.  
– Control is based on (recursive) functions.  
– Instead of for-loops, while-loops, do-loops, iterators, etc.  
  • can be defined as library functions.  
– Makes it easy to define semantics

Supports *first-class, lexically scoped, higher-order* procedures  
– a.k.a. first-class functions or closures or lambdas.  
– *first-class*: functions are data values like any other data value  
  • like numbers, they can be stored, defined anonymously, ...  
– *lexically scoped*: meaning of variables determined statically.  
– *higher-order*: functions as arguments and results  
  • programs passed to programs; generated from programs

These features also found in Scheme, Haskell, Scala, F#, Clojure, ....
Why OCaml?

**Statically typed:** debugging and testing aid
- compiler catches many silly errors before you can run the code.
  - A type is worth a thousand tests
- Java is also strongly, statically typed.
- Scheme, Python, Javascript, etc. are all strongly, *dynamically typed* – type errors are discovered while the code is running.

**Strongly typed:** compiler enforces type abstraction.
- cannot cast an integer to a record, function, string, etc.
  - so we can utilize *types as capabilities*; crucial for local reasoning
- C/C++ are *weakly typed* (statically typed) languages. The compiler will happily let you do something smart (*more often stupid*).

**Type inference:** compiler fills in types for you
OCaml Resources

• Home: https://ocaml.org/

• Tutorial: https://ocaml.org/learn/tutorials/

• User Manual: https://caml.inria.fr/pub/docs/manual-ocaml-4.09/

• Cheat Sheets: https://ocaml.org/docs/cheat_sheets.html

• 99 Problems (solved) in OCaml: https://ocaml.org/learn/tutorials/99problems.html
OCaml Installation

- [https://ocaml.org/docs/install.html](https://ocaml.org/docs/install.html)

- **Linux/macOS:**
  - Compiler: follow the online instructions
  - Editor: any text editor you like, e.g. Emacs, Vim

- **Windows:**
  - Compiler:
    - recommend OCPWin ([https://www.typerex.org/ocpwin.html](https://www.typerex.org/ocpwin.html))
    - easy installation: EXE file
  - Editor:
    - recommend OCaml-Top ([https://www.typerex.org/ocaml-top.html](https://www.typerex.org/ocaml-top.html))
    - recommend Version 1.1.1 ([https://github.com/OCamlPro/ocaml-top/releases](https://github.com/OCamlPro/ocaml-top/releases)): easy installation with MSI file
Install Successfully?

Chunhui Guo@ChunhuiGuo-PC ~
$ ocaml -version
The OCaml toplevel, version 4.02.1+ocp1

Chunhui Guo@ChunhuiGuo-PC ~
$ ocamlc -version
4.02.1+ocp1
OCaml Online Compiler

• Try OCaml by OCamlPRO: https://try.ocamlpro.com/

• IOCamlJS notebook:
  • https://andrewray.github.io/iocamljs/
  • similar to Jupyter Notebook for Python
A First OCaml Program

- [https://caml.inria.fr/pub/docs/u3-ocaml/ocaml-steps.html](https://caml.inria.fr/pub/docs/u3-ocaml/ocaml-steps.html)
- "Hello world" program
  - file: hello.ml
    ```ml
    print_string "Hello world!\n";;
    ```
  - a function
  - string argument enclosed in "...", no parens
  - normally call a function f like this: `f arg1 arg2 ...`
  - parens are used for grouping, precedence only when necessary
  - ;; end of code block
How to execute OCaml program?

- (1) compile and execute

```bash
Chunhui Guo@ChunhuiGuo-PC /cygdrive/d/OCaml_Code
$ ocamlc -o hello hello.ml

Chunhui Guo@ChunhuiGuo-PC /cygdrive/d/OCaml_Code
$ ./hello
Hello world!
```

```bash
D:\OCaml_Code>ocamlc -o hello.o hello.ml

D:\OCaml_Code>hello.o
Hello world!
```
How to execute OCaml program?

- (2) type interactively, using the interpreter `ocaml` as a big desk calculator

```
Chunhui Guo@ChunhuiGuo-PC /cygdrive/d/OCaml_Code
$ ocaml hello.ml
Hello world!

Chunhui Guo@ChunhuiGuo-PC /cygdrive/d/OCaml_Code
$ ocaml < hello.ml
OCaml version 4.02.1+ocp1

# Hello world!
- : unit = ()
#```
How to execute OCaml program?

• (3) use the interpreter `ocaml` in batch mode for running scripts
(* sum the numbers from 0 to n
   precondition: n must be a natural number *)

let rec sumTo (n:int) : int =
  match n with
  0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
Let's break down the OCaml code snippet provided:

```ocaml
(* sum the numbers from 0 to n
   precondition: n must be a natural number *)
let rec sumTo (n:int) : int =
  match n with
  0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
```

**Explanation:**
- The function `sumTo` is defined to calculate the sum of numbers from 0 to `n`, where `n` is a natural number.
- The `let rec` construction indicates recursion, allowing the function to call itself with a decremented value until reaching 0.
- The `match` statement handles the base case and recursive case:
  - If `n` is 0, it returns 0 (base case).
  - Otherwise, it returns `n + sumTo (n-1)` (recursive case).
- The `let _` block prints the result of `sumTo 8` and outputs a newline.

---

**Notes:**
- The keyword “let” begins a definition; keyword “rec” indicates recursion.
- The precondition ensures that `n` is a natural number.
- The function calculates the sum efficiently using recursion.
sumTo8.ml:

(* sum the numbers from 0 to n
   precondition: n must be a natural number *)
let rec sumTo (n:int) : int =
  match n with
  | 0   -> 0
  | n   -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8); print_newline()
A Second OCaml Program

deconstruct the value n using pattern matching

sumTo8.ml:

(* sum the numbers from 0 to n
   precondition: n must be a natural number
*)

let rec sumTo (n:int) : int =
  match n with
  0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
(* sum the numbers from 0 to n  
   precondition: n must be a natural number  
*)
let rec sumTo (n:int) : int = 
   match n with 
      0 -> 0 
    | n -> n + sumTo (n-1)

let _ = 
   print_int (sumTo 8); 
   print_newline()
Each branch of the match statement constructs a result

sumTo8.ml:

```ocaml
(* sum the numbers from 0 to n
   precondition: n must be a natural number *)
let rec sumTo (n:int) : int =
  match n with
  | 0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
```

construct the result 0

construct a result using a recursive call to sumTo
A Second OCaml Program

sumTo8.ml:

(* sum the numbers from 0 to n 
   precondition: n must be a natural number *)
let rec sumTo (n:int) : int = 
  match n with 
    0 -> 0 
  | n -> n + sumTo (n-1)

let _ = 
  print_int (sumTo 8); 
  print_newline()