Thinking Functionally

In Java or C, you get (most) work done by \textit{changing} something

\begin{verbatim}
temp = pair.x;
pair.x = pair.y;
pair.y = temp;
\end{verbatim}

commands \textit{modify} or \textit{change} an existing data structure (like pair)

In OCaml, you get (most) work done by \textit{producing} something

\begin{verbatim}
let (x,y) = pair in (y,x)
\end{verbatim}

you \textit{analyze} existing data (like pair) and you \textit{produce} new data (y,x)
pure, functional code:

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

• outputs are everything!
• output is function of input
• persistent
• repeatable
• parallelism apparent
• easier to test
• easier to compose

imperative code:

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

• outputs are irrelevant!
• output is not function of input
• volatile
• unrepeateable
• parallelism hidden
• harder to test
• harder to compose

Thinking Functionally
What else makes OCaml different?

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 aspects are in common with other functional languages such as Scheme, Haskell, SML, Clojure, CoffeeScript.
What else makes OCaml different?

**Statically typed:**
- compiler catches many silly errors before you can run the code.
- e.g., calling a function with the wrong number of arguments
- Java is also strongly, statically typed.
- Scheme, Bash, Python, Javascript, Basic, 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_ languages. The compiler will happily let you do something smart (_more often stupid_).

**Type inference:** compiler fills in types for you
Installing, running Ocaml

• Ocaml comes with an interactive, top-level loop.
  – useful for testing and debugging code.
  – “ocaml” at the prompt.

• It also comes with compilers
  – “ocamlc” – fast bytecode compiler
  – “ocamlopt” – optimizing, native code compiler
  – command line interface similar to GCC

• And many other tools
  – e.g., debugger, dependency generator, profiler, etc.

• See the course web pages for instructions on installing and using O’Caml
Editing Ocaml Programs

- Many options: pick your own poison
  - Emacs
    - what I’ll be using in class.
    - good but not great support for Ocaml.
    - on the other hand, it’s still the best code editor I’ve encountered.
    - (extensions written in elisp – a functional language!)
  - Ocaml IDE
    - integrated development environment written in Ocaml.
    - haven’t used it much, so can’t comment.
  - Eclipse
    - I’ve put up a link to an Ocaml plugin
    - I haven't tried it but others recommend it
XKCD on Editors

nano? REAL PROGRAMMERS USE emacs

HEY. REAL PROGRAMMERS USE vim.

WELL, REAL PROGRAMMERS USE ed.

NO, REAL PROGRAMMERS USE cat.

REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.

EXCUSE ME, BUT REAL PROGRAMMERS USE BUTTERFLIES.

THEY OPEN THEIR HANDS AND LET THE DELICATE WINGS FLAP ONCE.

THE DISTURBANCE RIPPLES OUTWARD, CHANGING THE FLOW OF THE EDDY CURRENTS IN THE UPPER ATMOSPHERE.

WHICH ACT AS LENSES THAT DEFLECT INCOMING COSMIC RAYS, FOCUSING THEM TO STRIKE THE DRIVE PLATTER AND FLIP THE DESIRED BIT.

NICE. ‘COURSE, THERE’S AN EMACS COMMAND TO DO THAT.

OH YEAH! GOOD OL’ C-x M-c M-butterfly...

DAMMIT, EMACS.
AN INTRODUCTORY EXAMPLE (OR TWO)
Demo:
- emacs
- ml files
- writing simple programs: hello.ml, sum.ml
- simple debugging and unit tests
- ocamlc compiler
- ocaml top-level loop
  - #use
  - #load
  - #quit
A First O’Caml Program

hello.ml:

```ocaml
camlex hello.ml

print_string "hello cos326!!\n";;
```

```bash
0
```

```bash
quit
```
A First O’Caml Program

hello.ml:

```ocaml
code
print_string "hello cos326!!\n";;
```

da function

it’s string argument enclosed in " . . . "

top-level expressions terminated by ;;
A First O’Caml Program

hello.ml:

```ocaml
print_string "hello cos326!!\n";;
```

compiling and running hello.ml:

```bash
$ ocamlc hello.ml -o hello
$ ./hello
hello cos326!!
$
```
hello.ml:

print_string "hello cos326!!\n";;

interpreting and playing with hello.ml:

$ ocaml
   Objective Caml Version 3.12.0
#
A First O’Caml Program

hello.ml:

```
print_string "hello cos326!!\n";;
```

interpreting and playing with hello.ml:

```
$ ocaml
    Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
#
```
A First O’Caml Program

hello.ml:

```
print_string "hello cos326!!\n";;
```

interpreting and playing with hello.ml:

```
$ ocaml
    Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
# #use "hello.ml";;
hello cos326!!
- : unit = ()
#```
A First O’Caml Program

hello.ml:

```ocaml
print_string "hello cos326!!\n";;
```

interpreting and playing with hello.ml:

```
$ ocaml
    Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
# #use "hello.ml";;
hello cos326!!
- : unit = ()
# #quit;;
$
```
A Second O’Caml 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)
;;

print_int (sumTo 8);;
print_newline();;
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)
  ;;
print_int (sumTo 8);;
print_newline();;
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)
  ;;
print_int (sumTo 8);;
print_newline();;

result type int
argument named n with type int
A Second O’Caml 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)
  ;;

print_int (sumTo 8);;
print_newline();;
A Second O’Caml Program

vertical bar "|" separates the alternative patterns

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)
;;

print_int (sumTo 8);
print_newline();

deconstructed data matches one of 2 cases:
(i) the data matches the pattern 0, or (ii) the data matches the variable pattern n
Each branch of the match statement constructs a result

```
(* 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)
;;
print_int (sumTo 8);;
print_newline();;
```

`sumTo8.ml:`

Each branch of the match statement constructs a result:

- The branch `match n with 0 -> 0` constructs the result 0.
- The branch `| n -> n + sumTo (n-1)` constructs a result using a recursive call to `sumTo`.

`A Second O’Caml 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)
;;
print_int (sumTo 8);;
print_newline();;
O’CAML BASICS: EXPRESSIONS, VALUES, SIMPLE TYPES
Expressions, Values, Types

- **Expressions** are computations
  - 2 + 3 is a computation

- **Values** are the results of computations
  - 5 is a value

- **Types** describe collections of values and the computations that generate those values
  - int is a type
    - values of type int include
      - 0, 1, 2, 3, ..., max_int
      - -1, -2, ..., min_int
More simple types, values, operations

<table>
<thead>
<tr>
<th>Type</th>
<th>Values:</th>
<th>Expressions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>-2, 0, 42</td>
<td>42 * (13 + 1)</td>
</tr>
<tr>
<td>float</td>
<td>3.14, -1., 2e12</td>
<td>(3.14 +. 12.0) *. 10e6</td>
</tr>
<tr>
<td>char</td>
<td>‘a’, ‘b’, ‘&amp;’</td>
<td>int_of_char ‘a’</td>
</tr>
<tr>
<td>string</td>
<td>“moo”, “cow”</td>
<td>“moo” ^ “cow”</td>
</tr>
<tr>
<td>bool</td>
<td>true, false</td>
<td>if true then 3 else 4</td>
</tr>
<tr>
<td>unit</td>
<td>()</td>
<td>print_int 3</td>
</tr>
</tbody>
</table>

For more primitive types and functions over them, see the Ocaml Reference Manual here:

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html
There are a number of ways to define a programming language.

In this class, we will briefly investigate:

- Syntax
- Evaluation
- Type checking

Standard ML, a very close relative of O'Caml, has a full definition of each of these parts and a number of proofs of correctness.

- For more on this theme, see COS 441

The O'Caml Manual fleshes out the syntax and some of the evaluation constraints and type checking rules.
O’CAML BASICS:
CORE EXPRESSION SYNTAX
Core Expression Syntax

The simplest O'Caml expressions e are:

- **values**
  - numbers, strings, bools, ...

- **id**
  - variables (x, foo, ...)

- **e₁ op e₂**
  - operators (x+3, ...)

- **id e₁ e₂ ... eₙ**
  - function call (foo 3 42)

- **let id = e₁ in e₂**
  - local variable decl.

- **if e₁ then e₂ else e₃**
  - a conditional

- **(e)**
  - a parenthesized expression

- **(e : t)**
  - an expression with its type
A note on parentheses

In most languages, arguments are parenthesized & separated by commas:

\[
f(x, y, z) \quad \text{sum}(3, 4, 5)
\]

In Ocaml, we don’t write the parentheses or the commas:

\[
f \; x \; y \; z \quad \text{sum} \; 3 \; 4 \; 5
\]

But we do have to worry about \emph{grouping}. For example,

\[
f \; x \; y \; z
\]

\[
f \; x \; (y \; z)
\]

The first one passes three arguments to \( f \) (\( x, y, \) and \( z \))
The second passes two arguments to \( f \) (\( x, \) and the result of applying the function \( y \) to \( z \).)
Type Checking

• Every value has a type and so does every expression
• This is a concept that is familiar from Java but it becomes more important when programming in a functional language
• The type of an expression is determined by the type of its subexpressions
• We write \( (e : t) \) to say that expression \( e \) has type \( t \). eg:

\[
\begin{align*}
2 & : \text{int} & "hello" & : \text{string} \\
2 + 2 & : \text{int} & "I say " \ ^ \ "hello" & : \text{string}
\end{align*}
\]
Type Checking Rules

• There are a set of **simple rules** that govern type checking
  – programs that do not follow the rules will not type check and *O’Caml* will refuse to compile them for you (the nerve!)
  – at first you may find this to be a pain ...

• But types are a great thing:
  – they help us think about how to construct our programs
  – they help us find stupid programming errors
  – they help us track down compatibility errors quickly when we edit and maintain our code
  – they allow us to enforce powerful invariants about our data structures
Type Checking Rules

• Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "...")
Type Checking Rules

• Example rules:

(1) \(0 : \text{int}\) (and similarly for any other integer constant \(n\))

(2) \"abc\" : \text{string} (and similarly for any other string constant \"\ldots\")

(3) if \(e_1 : \text{int}\) and \(e_2 : \text{int}\) then \(e_1 + e_2 : \text{int}\)

(4) if \(e_1 : \text{int}\) and \(e_2 : \text{int}\) then \(e_1 * e_2 : \text{int}\)
Type Checking Rules

• Example rules:

(1) \(0 : \text{int}\) (and similarly for any other integer constant \(n\))

(2) "abc" : string (and similarly for any other string constant "...")

(3) if \(e_1 : \text{int}\) and \(e_2 : \text{int}\) then \(e_1 + e_2 : \text{int}\)

(4) if \(e_1 : \text{int}\) and \(e_2 : \text{int}\) then \(e_1 \times e_2 : \text{int}\)

(5) if \(e_1 : \text{string}\) and \(e_2 : \text{string}\) then \(e_1 \^ e_2 : \text{string}\)

(6) if \(e : \text{int}\) then \(\text{string\_of\_int}(e) : \text{string}\)
Type Checking Rules

• Example rules:

(1) \(0 : \text{int}\) (and similarly for any other integer constant \(n\))

(2) "abc" : \text{string} (and similarly for any other string constant "..."

(3) if \(e_1 : \text{int}\) and \(e_2 : \text{int}\) then \(e_1 + e_2 : \text{int}\)

(4) if \(e_1 : \text{int}\) and \(e_2 : \text{int}\) then \(e_1 * e_2 : \text{int}\)

(5) if \(e_1 : \text{string}\) and \(e_2 : \text{string}\) then \(e_1 ^ e_2 : \text{string}\)

(6) if \(e : \text{int}\) then \text{string_of_int} e : \text{string}

• Using the rules:

\(2 : \text{int}\) and \(3 : \text{int}\). (By rule 1)
Type Checking Rules

• Example rules:

(1) \(0 : \text{int}\) (and similarly for any other integer constant \(n\))

(2) \"abc\" : \text{string} (and similarly for any other string constant \"...\")

(3) if \(e_1 : \text{int}\) and \(e_2 : \text{int}\) then \(e_1 + e_2 : \text{int}\)

(4) if \(e_1 : \text{int}\) and \(e_2 : \text{int}\) then \(e_1 \ast e_2 : \text{int}\)

(5) if \(e_1 : \text{string}\) and \(e_2 : \text{string}\) then \(e_1 ^ e_2 : \text{string}\)

(6) if \(e : \text{int}\) then \text{string_of_int} e : \text{string}

• Using the rules:

\(2 : \text{int}\) and \(3 : \text{int}\). (By rule 1)
Therefore, \((2 + 3) : \text{int}\) (By rule 3)
Type Checking Rules

• Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "...")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

• Using the rules:

2 : int and 3 : int. (By rule 1)
Therefore, (2 + 3) : int (By rule 3)
5 : int (By rule 1)
Type Checking Rules

• Example rules:

(1) \(0 : \text{int}\)  \hspace{1cm} (and similarly for any other integer constant \(n\))

(2) \"abc\" : \text{string}  \hspace{1cm} (and similarly for any other string constant ":...\")

(3) if \(e_1 : \text{int}\) and \(e_2 : \text{int}\) then \(e_1 + e_2 : \text{int}\)

(4) if \(e_1 : \text{int}\) and \(e_2 : \text{int}\) then \(e_1 * e_2 : \text{int}\)

(5) if \(e_1 : \text{string}\) and \(e_2 : \text{string}\) then \(e_1 ^ e_2 : \text{string}\)

(6) if \(e : \text{int}\) then \text{string_of_int} e : \text{string}

• Using the rules:

\(2 : \text{int}\) and \(3 : \text{int}\).  \hspace{1cm} (By rule 1)

Therefore, \((2 + 3) : \text{int}\)  \hspace{1cm} (By rule 3)

\(5 : \text{int}\)  \hspace{1cm} (By rule 1)

Therefore, \((2 + 3) * 5 : \text{int}\)  \hspace{1cm} (By rule 4 and our previous work)
Type Checking Rules

• Example rules:

(1) \[ 0 : \text{int} \]  (and similarly for any other integer constant \( n \))

(2) \[ "abc" : \text{string} \]  (and similarly for any other string constant "…")

(3) \[
\text{if } e_1 : \text{int} \text{ and } e_2 : \text{int} \\
\text{then } e_1 + e_2 : \text{int}
\]

(4) \[
\text{if } e_1 : \text{int} \text{ and } e_2 : \text{int} \\
\text{then } e_1 \times e_2 : \text{int}
\]

(5) \[
\text{if } e_1 : \text{string} \text{ and } e_2 : \text{string} \\
\text{then } e_1 ^ e_2 : \text{string}
\]

(6) \[
\text{if } e : \text{int} \\
\text{then } \text{string_of_int } e : \text{string}
\]

• Another perspective:

\[ \text{rule } (4) \text{ for typing expressions} \]
\[ \text{says } \text{I can put any expression} \]
\[ \text{with type int in place of the } ????
\]
Type Checking Rules

• Example rules:

(1) \[ 0 : \text{int} \] (and similarly for any other integer constant \(n\))

(2) \[ "abc" : \text{string} \] (and similarly for any other string constant "...")

(3) \[ \text{if } e_1 : \text{int} \text{ and } e_2 : \text{int} \text{ then } e_1 + e_2 : \text{int} \]

(4) \[ \text{if } e_1 : \text{int} \text{ and } e_2 : \text{int} \text{ then } e_1 * e_2 : \text{int} \]

(5) \[ \text{if } e_1 : \text{string} \text{ and } e_2 : \text{string} \text{ then } e_1 \ ^\wedge e_2 : \text{string} \]

(6) \[ \text{if } e : \text{int} \text{ then } \text{string_of_int } e : \text{string} \]

• Another perspective:

\[ 7 \ast \text{???} \text{ : int} \]

rule (4) for typing expressions says I can put any expression with type \text{int} in place of the ????
Type Checking Rules

- Example rules:

  1. $0 : \text{int}$ (and similarly for any other integer constant $n$)
  2. "abc" : string (and similarly for any other string constant "...")
  3. if $e_1 : \text{int}$ and $e_2 : \text{int}$ then $e_1 + e_2 : \text{int}$
  4. if $e_1 : \text{int}$ and $e_2 : \text{int}$ then $e_1 * e_2 : \text{int}$
  5. if $e_1 : \text{string}$ and $e_2 : \text{string}$ then $e_1 ^ e_2 : \text{string}$
  6. if $e : \text{int}$ then string_of_int $e$ : string

- Another perspective:

  rule (4) for typing expressions says I can put any expression with type int in place of the ????
Type Checking Rules

• You can always start up the O’Caml interpreter to find out a type of a simple expression:

```
$ ocaml
   Objective Caml Version 3.12.0
#
```
• You can always start up the O’Caml interpreter to find out a type of a simple expression:

```
$ ocaml
    Objective Caml Version 3.12.0
# 3 + 1;;
```
Type Checking Rules

• You can always start up the O’Caml interpreter to find out a type of a simple expression:

```
$ ocaml
   Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
#
```

press return and you find out the type and the value.
Type Checking Rules

- You can always start up the O’Caml interpreter to find out a type of a simple expression:

```
$ ocaml
   Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
# "hello " ^ "world";;
- : string = "hello world"
#
```
You can always start up the O’Caml interpreter to find out a type of a simple expression:

```
$ ocaml
       Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
# "hello " ^ "world";;
- : string = "hello world"
# #quit;;
$
```
Type Checking Rules

• Example rules:

(1) 0 : int  (and similarly for any other integer constant n)
(2) "abc" : string  (and similarly for any other string constant "...")
(3) if e1 : int and e2 : int
    then e1 + e2 : int
(4) if e1 : int and e2 : int
    then e1 * e2 : int
(5) if e1 : string and e2 : string
    then e1 ^ e2 : string
(6) if e : int
    then string_of_int e : string

• Violating the rules:

"hello" : string  (By rule 2)
1 : int  (By rule 1)
1 + "hello" : ??  (NO TYPE! Rule 3 does not apply!)
Type Checking Rules

• Violating the rules:

```haskell
# "hello" + 1;;
Error: This expression has type string but an expression was expected of type int
```

• The type error message tells you the type that was expected and the type that it inferred for your subexpression.

• By the way, this was one of the nonsensical expressions that did not evaluate to a value.

• I consider it a good thing that this expression does not type check.
Type Checking Rules

• Violating the rules:

```plaintext
# "hello" + 1;;
Error: This expression has type string but an expression was expected of type int
```

• A possible fix:

```plaintext
# "hello" ^ (string_of_int 1);;
- : string = "hello1"
```

• One of the keys to becoming a good ML programmer is to understand type error messages.
Type Checking Rules

• More rules:

(7)  true : bool

(8)  false : bool

(9)  if $e_1 : \text{bool}$
    and $e_2 : t$ and $e_3 : t$ (for some type $t$)
    then if $e_1$ then $e_2$ else $e_3 : t$

• Using the rules:

if ???? then ???? else ???? : int
Type Checking Rules

• More rules:

(7)  true : bool

(8)  false : bool

(9)  if e1 : bool
      and e2 : t and e3 : t (for some type t)
      then if e1 then e2 else e3 : t

• Using the rules:

  if true then  ????  else  ????  : int
Type Checking Rules

• More rules:

(7) \( \text{true : bool} \)

(8) \( \text{false : bool} \)

(9) \( \text{if } e_1 : \text{bool} \) and \( e_2 : t \) and \( e_3 : t \) (for some type \( t \)) then \( \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \)

• Using the rules:

\[ \text{if true then } 7 \text{ else } ????: \text{int} \]
Type Checking Rules

• More rules:

(7) \( \text{true : bool} \)

(8) \( \text{false : bool} \)

(9) if \( \text{e1 : bool} \) and \( \text{e2 : t} \) and \( \text{e3 : t} \) (for some type \( t \))
then if \( \text{e1 then e2 else e3 : t} \)

• Using the rules:

\[
\text{if true then 7 else 8 : int}
\]
Type Checking Rules

• More rules:

(7) \( \text{true} : \text{bool} \)

(8) \( \text{false} : \text{bool} \)

(9) \( \text{if } e1 : \text{bool} \)

and \( e2 : t \) and \( e3 : t \) (for some type \( t \))

then \( \text{if } e1 \text{ then } e2 \text{ else } e3 : t \)

• Violating the rules

\[ \text{if false then } "1" \text{ else } 2 \quad : \text{????} \]

types don't agree -- one is a string and one is an int
• Violating the rules:

```haskell
# if true then "1" else 2;;
Error: This expression has type int but an expression was expected of type string
#```

Type Checking Rules
Type Checking Rules

• What about this expression:

```
# 3 / 0 ;;
Exception: Division_by_zero.
```

• Why doesn't the ML type checker do us the favor of telling us the expression will raise an exception?
Type Checking Rules

• What about this expression:

```ml
# 3 / 0 ;;
Exception: Division_by_zero.
```

• Why doesn't the ML type checker do us the favor of telling us the expression will raise an exception?
  – In general, detecting a divide-by-zero error requires we know that the divisor evaluates to 0.
  – In general, deciding whether the divisor evaluates to 0 requires solving the halting problem:

```ml
# 3 / (if turing_machine_halts m then 0 else 1) ;;
```

• There are type systems that will rule out divide-by-zero errors, but they require programmers supply proofs to the type checker
OVERALL SUMMARY:
A SHORT INTRODUCTION TO FUNCTIONAL PROGRAMMING
OCaml

OCaml is a *call-by-value, strong, statically typed, functional* programming language

- **functional**: OCaml functions *analyze* their inputs and *generate new* outputs
  - as opposed to C or Java functions which typically *modify/change* state
  - in OCaml, *outputs* of a function are *typically* completely determined by their *inputs*

- **call-by-value**: OCaml expressions compute *values* eagerly
  - as opposed to Haskell or Unix pipes that compute values on demand, lazily

- I like the *strong, static type*: all OCaml expressions are assigned a *type* before execution of the expression
  - the *type* of an expression correctly *predicts* the kind of *value* the expression will generate when it is executed
  - types help us *understand* and *write* our programs
  - *type inference* makes our programs compact
  - the type system is *strong* (ie: sound): there’s no funny business like in C where you think you have a pointer, but you actually have some non-pointer
END