A Functional Introduction

COS 326
Andrew W. Appel
Princeton University

slides copyright 2013-2015 David Walker and Andrew W. Appel
In **Java** or **C**, you get (most) work done by *changing* something

```
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*

```
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
- persistent
- 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
- volatile
- unrepeateable
- 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 Racket, Haskell, SML, F#, Clojure, ....
Why OCaml?

Statically typed: debugging and testing aid
- 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, 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
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
  – “ocamllib” – a nice wrapper that computes dependencies

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

• See the course web pages for instructions on installing and using O’Caml
• 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)
Ocami Compiler and Interpreter

- 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:

print_string "Hello COS 326!!\n";;
A First O’Caml Program

hello.ml:

```ocaml
print_string "Hello COS 326!!\n";;
```

- a function
- its string argument enclosed in "..."
- top-level expressions terminated by ;;
A First O’Caml Program

hello.ml:

```
print_string "Hello COS 326!!\n";;
```

compiling and running hello.ml:

```
$ ocamlbuild hello.d.byte
$ ./hello.d.byte
Hello COS 326!!
$
```

.d for debugging
(other choices .p for profiled; or none)

.byte for interpreted bytecode
(other choices .native for machine code)
hello.ml:

```
print_string "Hello COS 326!!\n";;
```

interpreting and playing with hello.ml:

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

hello.ml:

```
print_string "Hello COS 326!!\n";;
```

interpreting and playing with hello.ml:

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

```
print_string "Hello COS 326!!\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();;

a comment
(* ... *)
let rec sumTo (n:int) : int = 
  match n with 
  | 0 -> 0 
  | _n -> _n + sumTo (n-1) 
;;

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

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

sumTo8.ml:
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;;
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

```ocaml
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.

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

sumTo8.ml:
```
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();

print the result of calling sumTo on 8
print a new line
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>&quot;moo&quot;, &quot;cow&quot;</td>
<td>&quot;moo&quot; ^ &quot;cow&quot;</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
Not every expression has a value

Expression:

42 * (13 + 1)  evaluates to  588
(3.14 + 12.0) * 10e6  \rightarrow  151400000.
int_of_char 'a'  \rightarrow  97
"moo" ^ "cow"  \rightarrow  "moocow"
if true then 3 else 4  \rightarrow  3
print_int 3  \rightarrow  ()

1 + "hello"  does not evaluate!
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 441/510

The O'Caml Manual fleshes out the syntax, evaluation and type checking rules informally.
O’CAML BASICS:
CORE EXPRESSION SYNTAX
The simplest O'Caml expressions $e$ are:

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

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

- **$e_1 \ op \ e_2$**  
  *operators (x+3, ...)*

- **id $e_1 \ e_2 ... \ e_n$**  
  *function call (foo 3 42)*

- **let id = $e_1$ in $e_2$**  
  *local variable decl.*

- **if $e_1$ then $e_2$ else $e_3$**  
  *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 *grouping*. For example,

\[
\begin{align*}
& f \ x \ y \ z \\
& f \ x \ (y \ z)
\end{align*}
\]

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.)
O’CAML BASICS:
TYPE CHECKING
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:

2 : int
"hello" : string

2 + 2 : int
"I say " ^ "hello" : string
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 "...")

(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) "$abc" : \text{string}$  
(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 $e_1 \cdot e_2 : \text{int}$

(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 $e : \text{string}$
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 \( e_1 \ast e_2 : \text{int} \)

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

6. if \( e : \text{int} \) then \( \text{string}_\text{of}_\text{int} \ e : \text{string} \)

• Using the rules:

\[ 2 : \text{int} \text{ and } 3 : \text{int}. \]  (By rule 1)
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)
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 "...")

- Using the rules:

  - \(2 : \text{int}\) and \(3 : \text{int}\). (By rule 1)
  - Therefore, \((2 + 3) : \text{int}\) (By rule 3)
  - \(5 : \text{int}\) (By rule 1)
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)
  Therefore, (2 + 3) * 5 : int (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) 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}

• Another perspective:

\[??\?? \ * \ ??\?? \ : \ \text{int}\]

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 \(e_1 \times e_2 : \text{int}\)

5. If \(e_1 : \text{string}\) and \(e_2 : \text{string}\) then \(e_1 \wedge e_2 : \text{string}\)

6. If \(e : \text{int}\) then \(\text{string}_\text{of}_\text{int}\ e : \text{string}\)

• Another perspective:

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

Rule (4) for typing expressions says I can put any expression 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) 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 \(e_1 \wedge e_2 : \text{string}\)

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

• 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 the type of a simple expression:

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

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

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

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"
# #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:

```ml
# "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.

• It is a **good thing** that this expression does not type check!

  “Well typed programs do not go wrong”

  *Robin Milner, 1978*
Well typed programs do not go wrong

“Well typed programs do not go wrong”
Robin Milner, 1978
Type Checking Rules

- Violating the rules:

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

- A possible fix:

```ocaml
# "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 e1 : bool and e2 : t and e3 : t (for some type t) then if e1 then e2 else e3 : 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) \( true : \text{bool} \)

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

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

- Using the rules:

  \( \text{if } true \text{ then } 7 \text{ else } ????: \text{int} \)
Type Checking Rules

• More rules:

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

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

(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:

\[ \text{if true then 7 else 8 : int} \]
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 \( e_3 \) : \( t \)

• Violating the rules

if false then "1" else 2 : ????

types don't agree -- one is a string and one is an int
Type Checking Rules

• Violating the rules:

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

Type Checking Rules

• What about this expression:

```plaintext
# 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:

```
# 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:

```
# 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
Isn’t that cheating?

“Well typed programs do not go wrong”

Robin Milner, 1978

(3 / 0) is well typed. Does it “go wrong?” Answer: No.

“Go wrong” is a technical term meaning, “have no defined semantics.” Raising an exception is perfectly well defined semantics, which we can reason about, which we can handle in ML with an exception handler.

So, it’s not cheating.

(Discussion: why do we make this distinction, anyway?)
Type Soundness

“Well typed programs do not go wrong”

Programming languages with this property have sound type systems. They are called safe languages.

Safe languages are generally immune to buffer overrun vulnerabilities, uninitialized pointer vulnerabilities, etc., etc. (but not immune to all bugs!)

Safe languages: ML, Java, Python, ...

Unsafe languages: C, C++, Pascal
Well typed programs do not go wrong

Robin Milner

Turing Award, 1991

“For three distinct and complete achievements:

1. LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2. ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3. CCS, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.”

“Well typed programs do not go wrong”

Robin Milner, 1978
Also in 1978...

37 years

1941
- First von Neumann computer
- Transistors!
- First programming languages

1978
- Functional programming languages (FP, Scheme, ML)

2015
- most of you were born
OVERALL SUMMARY:
A SHORT INTRODUCTION TO
FUNCTIONAL PROGRAMMING
OCaml is a *functional* programming language

- Java gets most work done by *modifying* data
- OCaml gets most work done by producing *new, immutable* data

OCaml is a *typed* programming language

- 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
- the type system is *sound*; the language is *safe*