A Functional Introduction

COS 326
David Walker
Princeton University

Thinking Functionally

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:

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

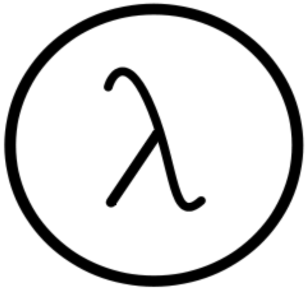
- *outputs are everything!*
- *output is <u>function</u> of input*
- *persistent*
- *repeatable*
- *parallelism apparent*
- *easier to test*
- *easier to compose*

imperative code:

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

- *outputs are irrelevant!*
- *output is not function of input*
- *volatile*
- *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 Racket, Haskell, SML, 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 (start at 6:20):
    - https://www.youtube.com/watch?v=q1Yi-WM7XqQ
- 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

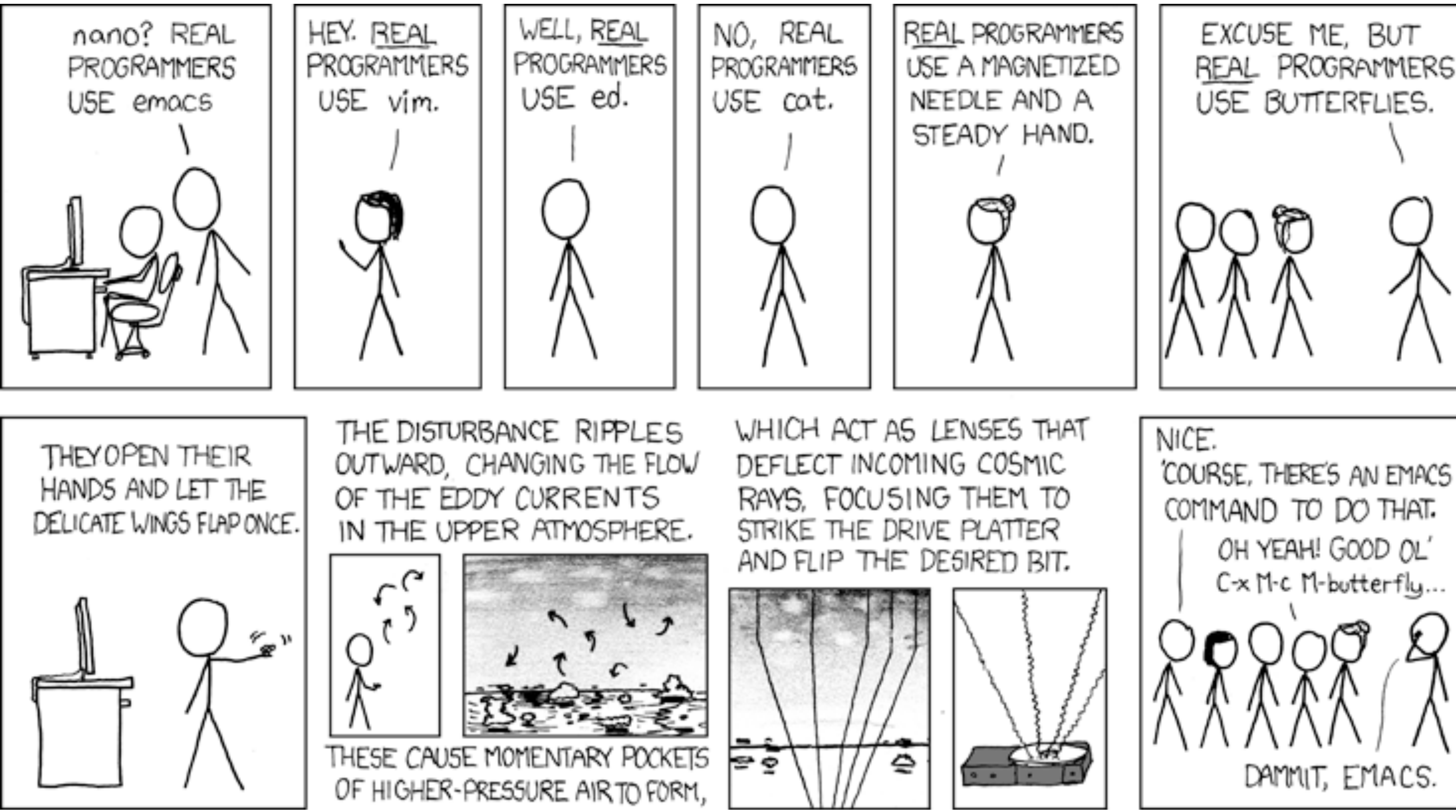
Installing, running Ocaml

- OCaml comes with compilers:
  - "ocamlc" – fast bytecode compiler
  - "ocamlopt" – optimizing, native code compiler
  - "ocamlbuild – a nice wrapper that computes dependencies
- And an interactive, top-level shell:
  - occasionally useful for trying something out.
  - "ocaml" at the prompt.
- And many other tools
  - e.g., debugger, dependency generator, profiler, etc.
- See the course web pages for installation pointers
  - also OCaml.org

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 used
      - that may be because I'm old and stuck in my ways :-)
    - (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
  - Sublime
    - A lot of students seem to gravitate to this

XKCD on Editors



AN INTRODUCTORY EXAMPLE (OR TWO)

OCaml Compiler and Interpreter

- Demo:
  - emacs
  - ml files
  - writing simple programs: hello.ml, sum.ml
  - simple debugging and unit tests
  - ocamlc compiler

A First OCaml Program

hello.ml:

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

A First OCaml Program

hello.ml:

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

a function

its string argument enclosed in " . . . "

top-level expressions terminated by ;;

A First OCaml 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)

A First OCaml Program

hello.ml:

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

interpreting and playing with hello.ml:

```
$ ocaml
        Objective Caml Version 3.12.0
#
```

A First OCaml 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
#
```

A First OCaml 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
# #use "hello.ml";;
hello cos326!!
- : unit = ()
#
```

A First OCaml 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
# #use "hello.ml";;
hello cos326!!
- : unit = ()
# #quit;;
$
```

A Second OCaml Program

a comment
(* ... *)

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 OCaml Program

the name of the function being defined

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();;
```

top-level declaration ends with ";;"

the keyword "let" begins a definition
the keyword "rec" indicates the definition is recursive

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)
;;

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

result type int

argument named n with type int

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)
;;

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

data to be deconstructed appears between key words “match” and “with”

A Second OCaml 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

A Second OCaml Program

Each branch of the match statement constructs a result

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();;
```

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)
;;

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

print the result of calling sumTo on 8

print a new line

OCAML 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

Type:	Values:	Expressions:
`int`	`-2, 0, 42`	`42 * (13 + 1)`
`float`	`3.14, -1., 2e12`	`(3.14 +. 12.0) *. 10e6`
`char`	`'a', 'b', '&'`	`int_of_char 'a'`
`string`	`"moo", "cow"`	`"moo" ^ "cow"`
`bool`	`true, false`	`if true then 3 else 4`
`unit`	`()`	`print_int 3`

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        ↦       151400000.
int_of_char ’a’               ↦       97
"moo" ^ "cow"                 ↦       “moocow"
if true then 3 else 4         ↦       3
print_int 3                   ↦       ()

1 + "hello"   does not evaluate!
```

Language Definition

- 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 OCaml, 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 OCaml Manual fleshes out the syntax, evaluation and type checking rules informally

OCAML BASICS:
CORE EXPRESSION SYNTAX

Core Expression Syntax

The simplest OCaml 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)        sum(3,4,5)
```

In OCaml, we don't write the parentheses or the commas:

```
f x y z         sum 3 4 5
```

But we do have to worry about *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.)

OCAML 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 : 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

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

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)

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

(2) "abc" : string (and similarly for a ... ")

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

FYI: This is a ***formal proof*** that the expression is well-typed!

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

... tring_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 : 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

- Another perspective:

???? * ???? : 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 : 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

- Another perspective:

7 * ???? : 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 : 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

- Another perspective:

7 * (add_one 17) : int

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 OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
        Objective Caml Version 3.12.0
#
```

Type Checking Rules

- You can always start up the OCaml 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 OCaml 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 OCaml 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"
#
```

press return and you find out the type and the value

Type Checking Rules

- You can always start up the OCaml 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:

```
# "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

Type Checking Rules

- Violating the rules:

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

- A possible fix:

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

- *One of the keys to becoming a good ML programmer is to understand type error messages.*

OVERALL SUMMARY:
A SHORT INTRODUCTION TO FUNCTIONAL PROGRAMMING

OCaml

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*