

COS 326

Functional Programming

Andrew Appel
Princeton University



Alonzo Church, 1903-1995
Princeton Professor, 1929-1967

In 1936, Alonzo Church invented the lambda calculus. He called it a logic, but it was a language of pure functions -- the world's first programming language.

He said:

"There may, indeed, be other applications of the system than its use as a logic."

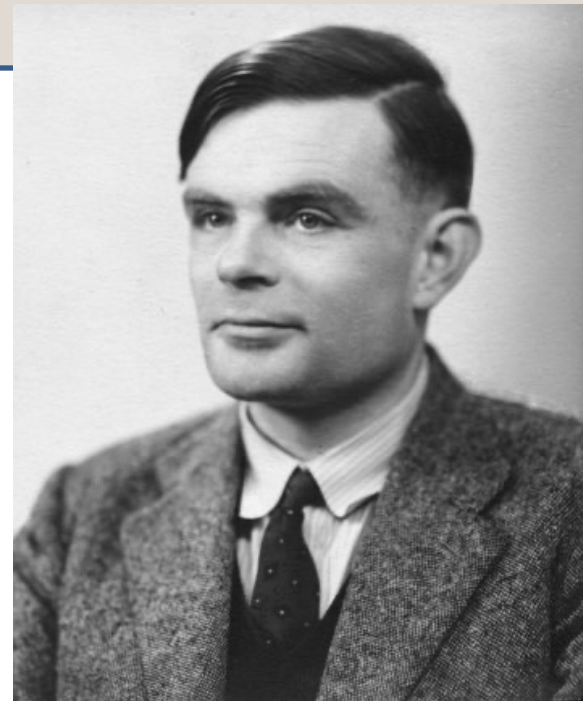
Indeed!



Alonzo Church
1934 -- developed lambda calculus



Programming Languages



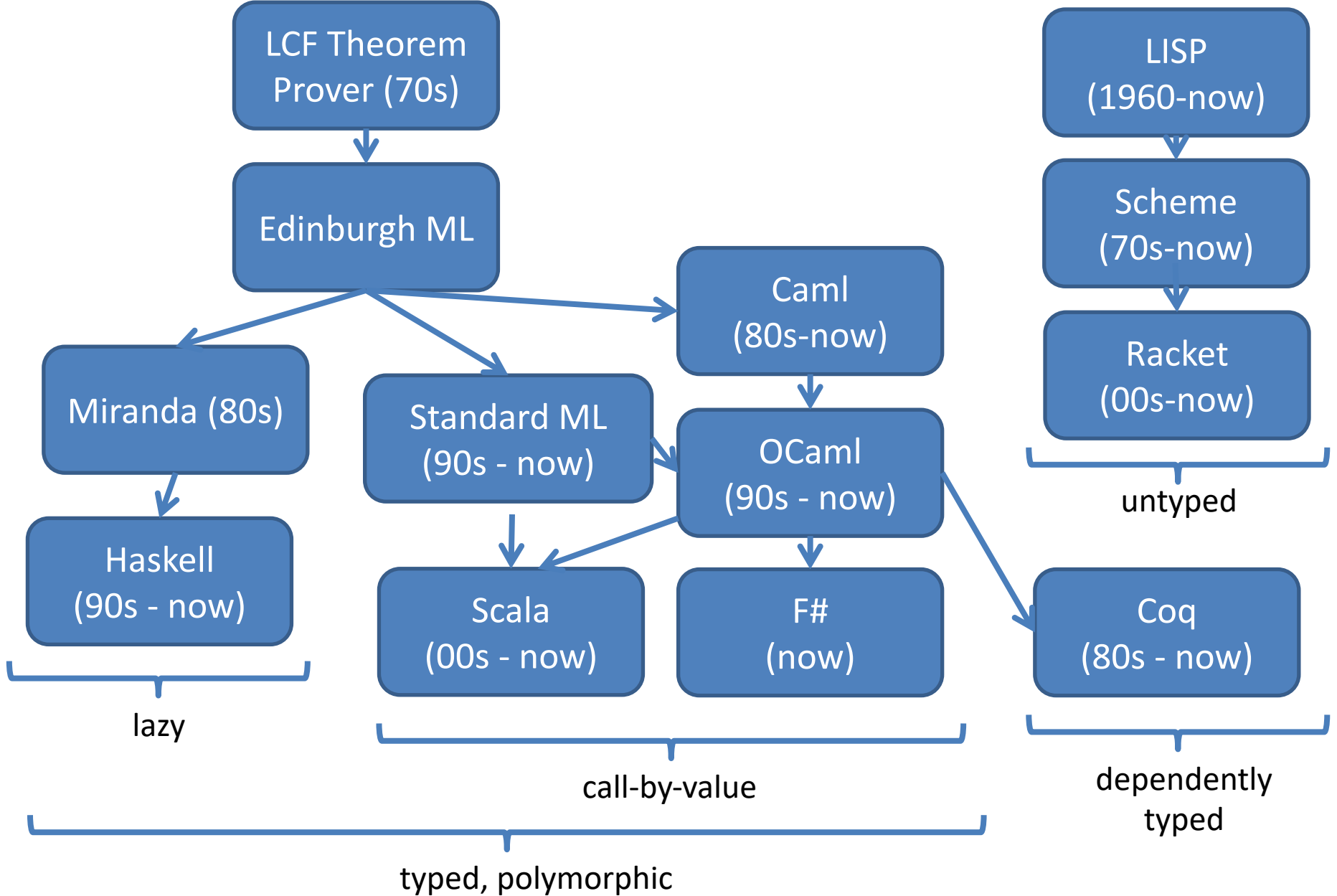
Alan Turing (PhD Princeton 1938)
1936 -- developed Turing machines



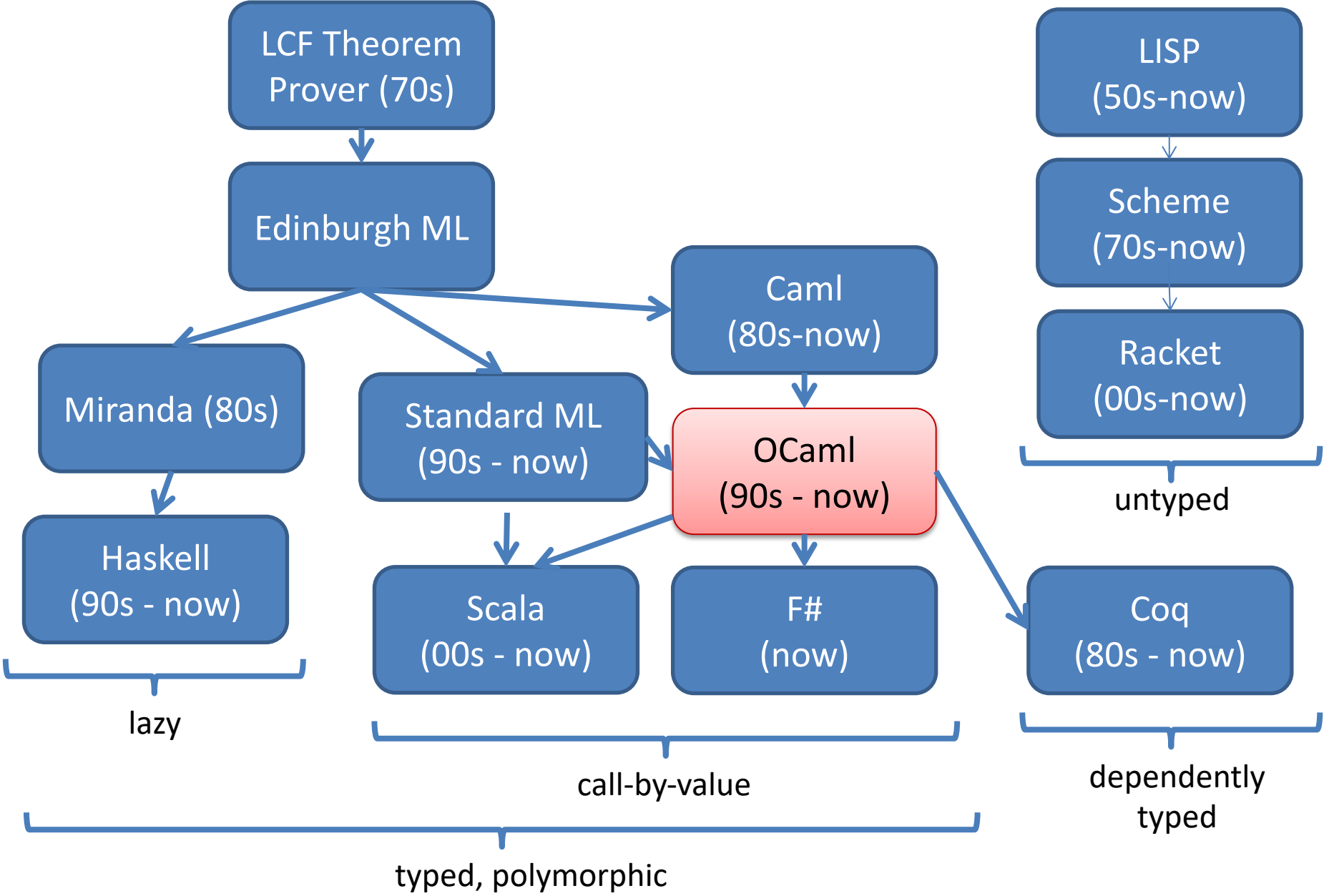
Computers

*Optional reading: **The Birth of Computer Science at Princeton in the 1930s**
by Andrew W. Appel, 2012. <http://press.princeton.edu/chapters/s9780.pdf>*

Vastly Abbreviated FP Genealogy



Vastly Abbreviated FP Genealogy



Functional Languages: Who's using them?



map-reduce in their data centers

Scala for correctness, maintainability, flexibility



Erlang for concurrency, Haskell for managing PHP, OCaml for bug-finding



F# in Visual Studio



Coq (re)proof of 4-color theorem



Haskell to synthesize hardware



Haskell for specifying equity derivatives

www.artima.com/scalazine/articles/twitter_on_scala.html

www.infoq.com/presentations/haskell-barclays

www.janestreet.com/technology/index.html#work-functionally

msdn.microsoft.com/en-us/fsharp/cc742182

research.google.com/archive/mapreduce-osdi04.pdf

www.lightbend.com/case-studies/how-apache-spark-scala-and-functional-programming-made-hard-problems-easy-at-barclays

www.haskell.org/haskellwiki/Haskell_in_industry

COURSE LOGISTICS

Course Staff



Andrew Appel
Professor
office: CS 209
email: appel@cs



Emma Farkash
Preceptor



Shaowei Zhu
Preceptor



COS 326

Functional Programming

Resources

[Topics Notes](#)[Installing and Editors](#)[Program Style Guide](#)[Standard Library](#)[Standard Modules](#)[Ocamlbuild Docs](#)[OCaml Manual](#)[OCaml Books](#)

Other Useful Stuff

[Course Info](#)

Welcome

Welcome to COS 326: Functional Programming. In this course, you will learn about the joy of functional programming: From functions to futures, map-reduce to monads, interfaces to invariants, and types to tail calls.

Getting Started

- Go to our page on [installing OCaml and VSCode](#).
- [Create a github account](#) if you don't already have one.
- Visit the [Assignments Dashboard](#) to link your PU netid to your github account.
- Go to the [Course Info](#) page and take a look at the course policies on collaboration, late

Collaboration Policy

The COS 326 collaboration policy can be found here:

<http://www.cs.princeton.edu/~cos326/info.php#collab>

Read it in full prior to beginning the first assignment.

Please ask questions whenever anything is unclear, at any time during the course.

A Typical Week

Monday

- Lecture

Tuesday

- Assignment from last week due (7 assignments total)
- Your first assignment is due Tuesday Sept 12 at 11:59pm

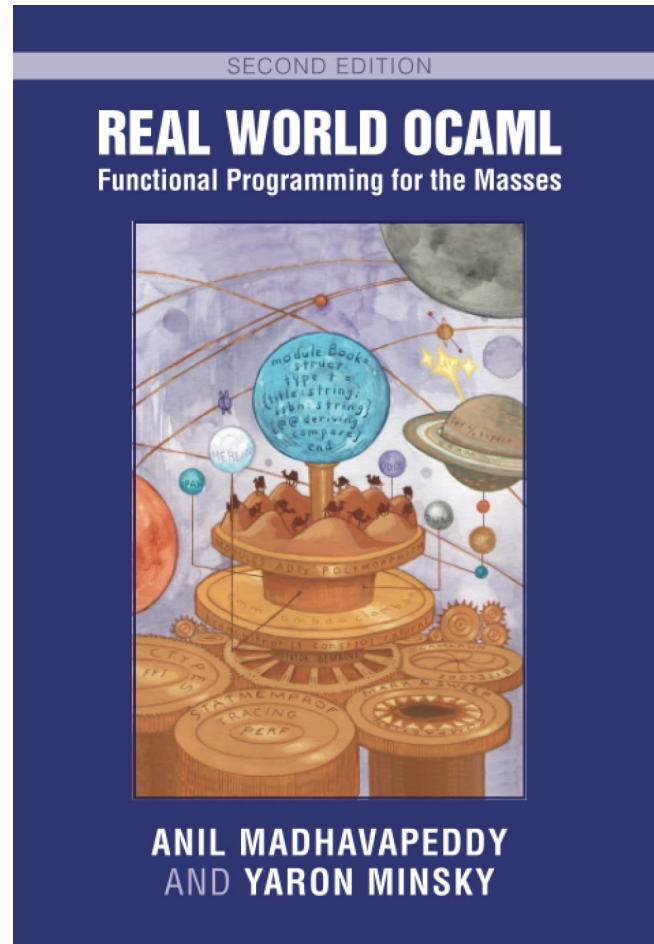
Wednesday

- Lecture
- Next assignment is available
- *start assignment* with material from lecture

Thursday/Friday

- mandatory precept reinforces lecture content
- you may have questions for your preceptor about the assignment

Course Textbook



<http://realworldocaml.org/>

Exams

Midterm

- in class during week after midterm break

Final

- during exam period in December
- make your travel plans accordingly
- I have *no control at all* over when the exam occurs, the Registrar schedules exams.
- The final is *not* “cumulative” over the whole semester, it covers just “equational reasoning”

Assignment 0

Install opam, ocaml, VS Code
[and if you use Windows: WSL2]
on your machine by the time precept begins tomorrow.

Resources Page:

<http://www.cs.princeton.edu/~cos326/resources.php>

Hint:

ocaml.org

Public Service Announcement

The Pen is Mightier than the Keyboard: Advantages of Longhand Over Laptop Note Taking

Pam Mueller (Princeton University)

Daniel Oppenheimer (UCLA)

Journal of Psychological Science, June 2014, vol 25, no 6

<http://pss.sagepub.com/content/25/6/1159.fullkeytype=ref&siteid=sppss&ijkey=CjRAwmrlURGNw>

<https://www.scientificamerican.com/article/a-learning-secret-don-t-take-notes-with-a-laptop/>

- You learn conceptual topics better by taking notes by hand.
- Instagram and Fortnite distract your classmates.

Functional Programming

Thinking Functionally

pure, functional code:

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

you *analyze* existing data (like pair)
and you *produce* new data (y,x)

imperative code:

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

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

Thinking Functionally

pure, functional code:

```
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:

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

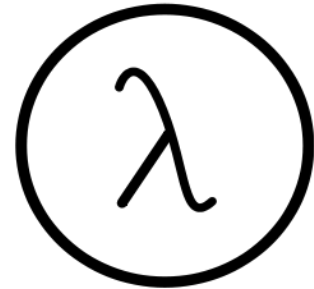
- *outputs are irrelevant!*
- *output is not function of input*
- *data properties change*
- *unrepeatable*
- *parallelism hidden*
- *harder to test*
- *harder to compose*

This simple switch in perspective can change the way you
think
about programming and problem solving.

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

Integer Functor Ord Char
 Either Monad
 Bool Enum
 Int [...] Eq
 -> Eq
 -> Eq
 Num Read
 Bounded (,_)
 Integral () IO Show
 Maybe String Ratio Float

I prefer the strong, static type.

Installing, Running OCaml

- OCaml comes with compilers:
 - "ocamlc" – fast bytecode compiler
 - "ocamlopt" – optimizing, native code compiler
 - "dune" – a build system for OCaml
- And an interactive, top-level shell:
 - useful for trying something out.
 - "ocaml" or "utop" at the prompt.
 - *but use the compiler (via dune) most of the time*
- See the course web pages for installation pointers
 - also OCaml.org

Editing OCaml Programs

- Many options:
 - We recommend VS Code, with its OCaml mode

But you can use other text editors if you want, such as:

- Emacs
 - what your professors tend to use
 - good but not great support for OCaml.
- Sublime, atom
 - Many CS326 students have used these

AN INTRODUCTORY EXAMPLE (OR TWO)

A First OCaml Program

hello.ml:

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

EXPLORER

LEC1 [WSL: UBUNTU-20....]

- assert.ml
- broken.ml
- countn.ml
- hello.ml
- hello2.ml
- Makefile
- sum_test.ml
- sum.ml

OUTLINE

TIMELINE

Visual Studio Code

Editing evolved

Start

- New File...
- Open File...
- Open Folder...
- Clone Git Repository...

Recent

test [WSL: Ubuntu-20.04] /home/a...

Walkthroughs

- Get Started with VS Code
Discover the best customizations to make VS Code yours.
- Learn the Fundamentals
Jump right into VS Code and get an overview of the must-have features.
- Boost your Productivity

More...

Show welcome page on startup

hello.ml

ml / mli icons and menu

home > appel > lec1 > hello.ml

```

1
2 print_string "Hello COS 326!!\n"
3

```



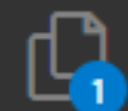
- File >
- Edit >
- Selection >
- View >
- Go >
- Run >
- Terminal >**
- Help >

- New Terminal Ctrl+Shift+`
- Split Terminal Ctrl+Shift+5

- Run Task...
- Run Build Task... Ctrl+Shift+B
- Run Active File
- Run Selected Text

- Show Running Tasks...
- Restart Running Task...
- Terminate Task...

- Configure Tasks...
- Configure Default Build Task...



hello.ml



home > appel > lec1 > hello.ml

```
1  
2 print_string "Hello COS 326!!\n"  
3
```

PROBLEMS

OUTPUT

TERMINAL



ubuntu\$

hello.ml

ml
mli

home > appel > lec1 > hello.ml

1

2 print_string "Hello COS 326!!\n"

3

PROBLEMS

OUTPUT

TERMINAL

bash

ubuntu\$ ocaml hello.ml

hello.ml

home > appel > lec1 > hello.ml

```
1  
2 print_string "Hello COS 326!!\n"  
3
```

PROBLEMS

OUTPUT

TERMINAL

bash - lec1

```
● ubuntu$ ocaml hello.ml  
Hello COS 326!!  
○ ubuntu$
```

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 "..."

a program
can be nothing
more than
just a single
expression
(but that is
uncommon)

no parens. normally call a function *f* like this:

```
f arg
```

(parens are used for grouping, precedence
only when necessary)

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();
```

a comment
(* ... *)



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)

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

the keyword "let" begins a definition; keyword "rec" indicates recursion

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()
```

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)

let _ =
  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)

let _ =
  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)

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

construct
the result 0

construct
a result
using a
recursive
call to sumTo

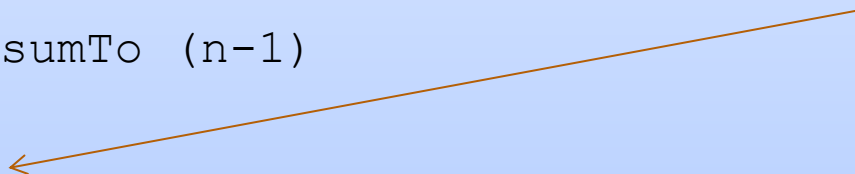
A Second OCaml Program

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()
```

print the
result of
calling
sumTo on 8



print a
new line



hello.ml

sum.ml

ml
mli

home > appel > lec1 > sum.ml > ...

```
1 (* sum the numbers from 0 to n
2   |
3   | sumTo 0 = 0
4   | sumTo 3 = 6
5   | sumTo (-1) = 0
6   *)
7 int -> int
8 let rec sumTo (n:int) : int =
9   | if n <= 0 then
10  |   0
11   | else
12  |   n + sumTo (n-1)
```

PROBLEMS

TERMINAL

bash - lec1

Hello COS 326!!

ubuntu\$

home > appel > lec1 > sum.ml > ...

```

1 (* sum the numbers from 0 +
2   sumTo 0 = 0
3   sumTo 3 = 6
4   sumTo (-1) = 0
5 *)
int -> int
6 let rec sumTo (n:int) : int =
7   if n <= 0 then
8     0
9   else
10    n + sumTo (n-1)

```

This is not part of the program, it's just VS Code reminding you the type of the "sumTo" function

PROBLEMS TERMINAL ...

bash - lec1

Hello COS 326!!

ubuntu\$

home > appel > lec1 > sum.ml > ...

```

1  (* sum the numbers from 0 +
2  |   sumTo 0 = 0
3  |   sumTo 3 = 6
4  |   sumTo (-1) = 0
5  *)
   int -> int
6  let rec sumTo (n:int) : int =
7  |   if n <= 0 then
8  |   |   0
9  |   else
10 |   |   n + sumTo (n-1)

```

Good program style: *before* each function definition, write a comment saying what it's supposed to do, perhaps with examples

PROBLEMS TERMINAL

bash - lec1

Hello COS 326!!

ubuntu\$

home > appel > lec1 > sum.ml > ...

```

6   let rec sumTo (n:int) : int =
7     | if n <= 0 then
8       | 0
9     | else
10    | n + sumTo (n-1)
11
12  let _ =
13    Printf.printf "The sum of the numbers from 0
    to 8 is %d\n" (sumTo 8)

```

PROBLEMS TERMINAL

bash - lec1

- ubuntu\$ `ocaml hello.ml`
Hello COS 326!!
- ubuntu\$

home > appel > lec1 > sum.ml > ...

```

6 let rec sumTo (n:int) : int =
7   if n <= 0 then
8     0
9   else
10    n + sumTo (n-1)
11
12 let _ =
13   Printf.printf "The sum of the numbers from 0
14   to 8 is %d\n" (sumTo 8)

```

PROBLEMS TERMINAL ...

bash - lec1 + v [] [] ^ x

```

• ubuntu$ ocaml sum.ml
The sum of the numbers from 0 to 8 is 36
○ ubuntu$ █

```

home > appel > lec1 > sum.ml > ...

```

15   let triang n = n*(n+1)/2
16
    int -> unit
17   let test n = assert (sumTo n = triang n)
18
19   let _ = test 0
20   let _ = test 1
21   let _ = test 5
22   let _ = test 10
23

```

PROBLEMS TERMINAL ...

bash - lec1

```

● ubuntu$ ocaml sum.ml
  The sum of the numbers from 0 to 8 is 36
○ ubuntu$

```

hello.ml • sum.ml

ml / mli icons and window control buttons.

home > appel > lec1 > sum.ml > ...

```
15 | let triang n = n*(n+1)/2
```

PROBLEMS TERMINAL ...

bash - lec1 + v [window control icons]

```
ubuntu$ utop
```

home > appel > lec1 > sum.ml > ...

```
15 | let triang n = n*(n+1)/2
```

ubuntu\$ utop

elcome to utop version 2.9.1 (using OCaml version 4.13.1)

Type #utop_help for help about using utop.

-(11:03:04)-< command 0 >-----{ counter: 0 }-
utop #

Arg	Arith_status	Array	ArrayLabels	Assert_failure	Atomi
-----	--------------	-------	-------------	----------------	-------

home > appel > lec1 > sum.ml > ...

```
15 let triang n = n*(n+1)/2
```

elcome to utop version 2.9.1 (using OCaml version 4.13.1)

Type #utop_help for help about using utop.

-(11:03:04)-< command 0 >-----{ counter: 0 }-

```
utop # let x = 5;;
val x : int = 5
```

-(11:03:04)-< command 1 >-----{ counter: 0 }-

```
utop #
```

Arg	Arith_status	Array	ArrayLabels	Assert_failure	Atomi
-----	--------------	-------	-------------	----------------	-------

home > appel > lec1 > sum.ml > ...

```
15 let triang n = n*(n+1)/2
```

PROBLEMS TERMINAL ... ocamlrun - lec1

-(11:03:04)-< command 0 >-----{ counter: 0 }-

utop # let x = 5;;

val x : int = 5

-(11:03:04)-< command 1 >-----{ counter: 0 }-

utop # #use "sum.ml";;

val sumTo : int -> int = <fun>

The sum of the numbers from 0 to 8 is 36

- : unit = ()

-(11:03:35)-< command 2 >-----{ counter: 0 }-

utop #

Arg	Arith_status	Array	ArrayLabels	Assert_failure	Atomi
-----	--------------	-------	-------------	----------------	-------

home > appel > lec1 > sum.ml > ...

```
15 let triang n = n*(n+1)/2
```

PROBLEMS TERMINAL ... ocamlrun - lec1

-(11:03:04)-< command 0 >-----{ counter: 0 }-

```
utop # let x = 5;;
```

val x : int = 5

-(11:03:04)-< command 1 >-----{ counter: 0 }-

```
utop # #use "sum.ml";;
```

val sumTo : int -> int = <fun>

The sum of the numbers from 0 to 8 is 36

- : unit = ()

-(11:03:35)-< command 2 >-----{ counter: 0 }-

```
utop # sumTo 6;;
```

Arg	Arith_status	Array	ArrayLabels	Assert_failure	Atomi
-----	--------------	-------	-------------	----------------	-------

home > appel > lec1 > sum.ml > ...

```
15 let triang n = n*(n+1)/2
```

PROBLEMS TERMINAL ... ocamlrun - lec1

```
val x : int = 5
-( 11:03:04 )-< command 1 >-----{ counter: 0 }-
```

```
utop # #use "sum.ml";;
```

```
val sumTo : int -> int = <fun>
```

The sum of the numbers from 0 to 8 is 36

```
- : unit = ()
```

```
-( 11:03:35 )-< command 2 >-----{ counter: 0 }-
```

```
utop # sumTo 6;;
```

```
- : int = 21
```

```
-( 11:04:14 )-< command 3 >-----{ counter: 0 }-
```

```
utop #
```

Arg	Arith_status	Array	ArrayLabels	Assert_failure	Atomi
-----	--------------	-------	-------------	----------------	-------

OCAML BASICS: EXPRESSIONS, VALUES, SIMPLE TYPES

Terminology: Expressions, Values, Types

Expressions are computations

- $2 + 3$ is a computation

Values (a subset of the expressions) 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

Some simple types, values, expressions

<u>Type:</u>	<u>Values:</u>	<u>Expressions:</u>
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:


<https://ocaml.org/api/Stdlib.html>

Evaluation

$$42 * (13 + 1)$$

Evaluation

$42 * (13 + 1) \text{ -->* } 588$




Read like this: “the expression $42 * (13 + 1)$ **evaluates to** the value 588”

The “*****” is there to say that it does so in 0 or more small steps

Evaluation

42 * (13 + 1) --> * 588



Read like this: “the expression 42 * (13 + 1) **evaluates to** the value 588”

The “*” is there to say that it does so in 0 or more small steps

Here I’m telling you how to execute an OCaml expression --- i.e., I’m telling you something about the *operational semantics* of OCaml

More on semantics later.

Evaluation

<code>42 * (13 + 1)</code>	<code>-->*</code>	<code>588</code>
<code>(3.14 +. 12.0) *. 10e6</code>	<code>-->*</code>	<code>151400000.</code>
<code>int_of_char 'a'</code>	<code>-->*</code>	<code>97</code>
<code>"moo" ^ "cow"</code>	<code>-->*</code>	<code>"moocow"</code>
<code>if true then 3 else 4</code>	<code>-->*</code>	<code>3</code>
<code>print_int 3</code>	<code>-->*</code>	<code>()</code>

Evaluation

1 + "hello" -->* ???

Evaluation

1 + "hello" -->* ???

“+” processes integers
“hello” is not an integer
evaluation is undefined!

Don't worry! This expression doesn't type check.

Aside: See this 4-min talk on Javascript:
<https://www.destroyallsoftware.com/talks/wat>

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

We write $(e : t)$ to say that *expression e has type t*. eg:

$2 : \text{int}$

$\text{"hello"} : \text{string}$

$2 + 2 : \text{int}$

$\text{"I say " ^ "hello"} : \text{string}$

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 OCaml 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:

- help us *think* about *how to construct* our programs
- help us *find stupid programming errors*
- help us track down errors quickly when we *edit our code*
- allow us to *enforce powerful invariants* about 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) $"\text{abc}" : \text{string}$ (and similarly for any other string constant "...")
- (3) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 + e2 : \text{int}$
- (4) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 * e2 : \text{int}$

Type Checking Rules

Example rules:

- (1) $0 : \text{int}$ (and similarly for any other integer constant n)
- (2) $"\text{abc}" : \text{string}$ (and similarly for any other string constant "...")
- (3) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 + e2 : \text{int}$
- (4) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 * e2 : \text{int}$
- (5) if $e1 : \text{string}$ and $e2 : \text{string}$
then $e1 \wedge e2 : \text{string}$
- (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) $"\text{abc}" : \text{string}$ (and similarly for any other string constant "...")
- (3) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 + e2 : \text{int}$
- (4) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 * e2 : \text{int}$
- (5) if $e1 : \text{string}$ and $e2 : \text{string}$
then $e1 \wedge e2 : \text{string}$
- (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) $"\text{abc}" : \text{string}$ (and similarly for any other string constant "...")
- (3) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 + e2 : \text{int}$
- (4) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 * e2 : \text{int}$
- (5) if $e1 : \text{string}$ and $e2 : \text{string}$
then $e1 \wedge e2 : \text{string}$
- (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) $0 : \text{int}$ (and similarly for any other integer constant n)
- (2) $"\text{abc}" : \text{string}$ (and similarly for any other string constant "...")
- (3) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 + e2 : \text{int}$
- (4) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 * e2 : \text{int}$
- (5) if $e1 : \text{string}$ and $e2 : \text{string}$
then $e1 \wedge e2 : \text{string}$
- (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)
 $5 : \text{int}$ (By rule 1)

Type Checking Rules

Example rules:

- (1) $0 : \text{int}$ (and similarly for any other integer constant n)
- (2) $"\text{abc}" : \text{string}$ (and similarly for any other string constant s)
- (3) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 + e2 : \text{int}$
- (5) if $e1 : \text{string}$ and $e2 : \text{string}$
then $e1 \wedge e2 : \text{string}$

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

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)
- Therefore, $(2 + 3) * 5 : \text{int}$ (By rule 4 and our previous work)

Type Checking Rules

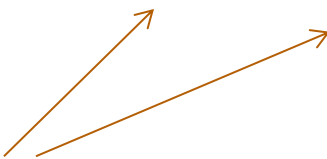
Example rules:

- (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:

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

`???? * ???? : 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`

Another perspective:

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

`7 * ????? : int`

Type Checking Rules

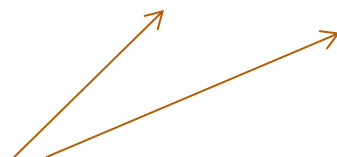
Example rules:

- (1) $0 : \text{int}$ (and similarly for any other integer constant n)
- (2) $"\text{abc}" : \text{string}$ (and similarly for any other string constant "...")
- (3) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 + e2 : \text{int}$
- (4) if $e1 : \text{int}$ and $e2 : \text{int}$
then $e1 * e2 : \text{int}$
- (5) if $e1 : \text{string}$ and $e2 : \text{string}$
then $e1 \wedge e2 : \text{string}$
- (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 ????

$7 * (\text{add_one } 17) : \text{int}$



Type Checking Rules

You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
      OCaml Version 4.13.1
#
```

Type Checking Rules

You can always start up the OCaml interpreter to find out a type of a simple expression:

or utop

```
$ ocaml
      OCaml Version 4.13.1
# 3 + 1;;
```

use “;;”
to end
a phrase
in the
top level

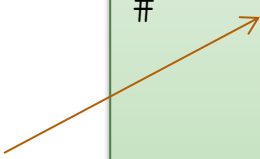
(“;;” can also end a top-level phrase in a file, but I’m going to avoid using it there because then some of you will confuse it with a “;” ...)

Type Checking Rules

You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
      OCaml Version 4.13.1
# 3 + 1;;
- : int = 4
#
```

press
return
and you
find out
the type
and the
value

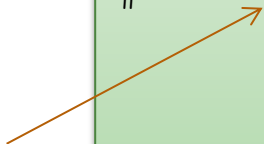


Type Checking Rules

You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
      OCaml Version 4.13.1
# 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
      OCaml Version 4.13.1
# 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:

<code>"hello" : string</code>	(By rule 2)
<code>1 : int</code>	(By rule 1)
<code>1 + "hello" : ??</code>	(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.

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:

```
# 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++

**OVERALL SUMMARY:
A SHORT INTRODUCTION TO
FUNCTIONAL PROGRAMMING**

OCaml

OCaml is a *functional* programming language

- express control flow and iteration by defining *functions*
- *not by modifying the values of variables and data structures*

Imperative: “do this”

Functional: “be this”

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*