Introduction to OCaml

Speaker: David Walker
COS 326
Princeton University
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
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:
  – useful for trying something out.
  – "ocaml" at the prompt.
  – *but use the compiler most of the time*

• 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 your professors use
    • good but not great support for OCaml.
    • we like it because we’re used to it
    • (extensions written in elisp – a functional language!)
  – Visual Studio
    • haven’t used it much, but pretty popular, I believe
  – Eclipse
    • we’ve put up a link to an OCaml plugin
    • we haven't tried it but others recommend it
  – Sublime, atom
    • A lot of students seem to gravitate to this
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.
WELL, THERE’s A “M-: C-c M-p”...

DAMMIT, EMACS.

OH YEAH! GOOD OL’ C-x M-c M-butterfly...
AN INTRODUCTORY EXAMPLE (OR TWO)
OCaml Compiler and Interpreter

• Demo:
  – emacs
  – ml files
  – writing
  – simple
  – ocamlc
OCaml Compiler and Interpreter

• Demo:
  - emacs
  - ml files
  - writing
  - simple
  - ocamlc
OCaml demo

```sh
ubunto$ ls
Makefile broken.ml hello.ml sum.ml
assert.ml countn.ml hello2.ml sum_test.ml
ubunto$
ubunto$ emacs hello.ml &
[1] 22831
ubunto$  
```
OCaml demo

```
-:--- *scratch*   All L1   (Lisp Interaction ElDoc)

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

```
-:***- hello.ml   All L1   Git-2020   (Tuareg)
```
OCaml demo

```ocaml
print_string "Hello COS 326!!\n"
```
OCaml demo

```ocaml
print_string "Hello COS 326!!!\n"
```
OCaml demo

```
# OCaml version 4.08.1

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

```
# print_string "Hello COS 326!!\n";;
Hello COS 326!!
- : unit = ()
#

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

```
-:***- *ocaml-toplevel*   All L6   (Tuareg-Interactive:run)

print_string "Hello COS 326!!\n"

-:***- hello.ml   All L3   Git-2020   (Tuareg)
```
OCaml demo

```
OCaml version 4.08.1

# print_string "Hello COS 326!!\n";;
Hello COS 326!!
- : unit = ()
#

let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)

let _ =
  Printf.printf "The sum of the numbers from 0 to 8 is %d\n" (sumTo 8)
```

```
Auto-saving...done
```
OCaml demo

```ocaml
# print_string "Hello COS 326!!\n";;
Hello COS 326!!
- : unit = ()
# let rec sumTo (n:int) : int =
   if n <= 0 then
     0
   else
     n + sumTo (n-1);;
   val sumTo : int -> int = <fun>

#

-: **- *ocaml-toplevel* Bot L12 (Tuareg-Interactive:run)
let rec sumTo (n:int) : int =
   if n <= 0 then
     0
   else
     n + sumTo (n-1)

let _ =
  Printf.printf "The sum of the numbers from 0 to 8 is %d\n" (sumTo 8)

-::--- sum.ml Bot L14 Git-2020 (Tuareg)
C-c C-e
```
OCaml demo

```ocaml
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)

let _ =
  Printf.printf "The sum of the numbers from 0 to 8 is %d\n" (sumTo 8);;
  The sum of the numbers from 0 to 8 is 36

let _ =
  Printf.printf "The sum of the numbers from 0 to 8 is %d\n" (sumTo 8);
```

OCaml demo

```
ubuntu$ ls
Makefile assert.ml countn.ml hello2.ml sum_test.ml
a.out broken.ml hello.ml sum.ml
ubuntu$
ubuntu$
ubuntu$
ubuntu$ ls
Makefile assert.ml countn.ml hello2.ml sum.cmo sum_test.ml
a.out broken.ml hello.ml sum.cmi sum.ml
ubuntu$
ubuntu$
ubuntu$
ubuntu$ ./a.out
The sum of the numbers from 0 to 8 is 36
ubuntu$
```
# OCaml demo

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

let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)

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

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

main ();;

Line 9, characters 2-15:
9 | print_newline()
   ^^^^^^^^^^^^^
Error: This function has type unit -> unit
It is applied to too many arguments; maybe you forgot a `;`.

#

---  *ocaml-toplevel*  Bot L48  (Tuareg-Interactive:run)
(* sum the numbers from 0 to n precondition: n must be a natural number *)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)

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

main ()
---  broken.ml  All L2  Git-2020  (Tuareg)
```ocaml
else
  n + sumTo (n-1)

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

main ();;

val sumTo : int -> int = <fun>
val main : unit -> unit = <fun>
#
  36
- : unit = ()
#
-:**- *ocaml-toplevel*  Bot L81  (Tuareg-Interactive:run)
(* sum the numbers from 0 to n precondition: n must be a natural number *)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)

let main () =
  print_int (sumTo 8);
  print_newline();;
main ()
-:**- broken.ml  All L10  Git-2020 (Tuareg)
```
OCaml demo

```ocaml
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)

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

let _ = main ()
```

```
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)

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

let _ = main ()
```
Ubuntu$ ls
Makefile broken.ml hello.ml sum.ml
assert.ml counrn.ml hello2.ml sum_test.ml
Ubuntu$
Ubuntu$
Ubuntu$ make
ocamlbuild hello.d.byte
Finished, 4 targets (0 cached) in 00:00:00.
Ubuntu$
Ubuntu$
Ubuntu$ ls
Makefile assert.ml counrn.ml hello.ml sum.ml
_build broken.ml hello.d.byte hello2.ml sum_test.ml
Ubuntu$
Ubuntu$ ls _build
_digests hello.cmi hello.d.byte hello.d.cmo hello.ml.depends
_log hello.cmo hello.d.cmi hello.ml
Ubuntu$
Ubuntu$ hello.d.byte
hello.d.byte: command not found
Ubuntu$ ./hello.d.byte
Hello COS 326!!
Ubuntu$
A First OCaml Program

hello.ml:

```
print_string "Hello COS 326!!\n"
```
hello.ml:

print_string "Hello COS 326!!\n"

a function
its string argument enclosed in "..."
no parens. normally call a function f like this:

f arg

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

(parens are used for grouping, precedence only when necessary)
A First OCaml Program

hello.ml:

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

compiling and running hello.ml:

```bash
$ 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:

```ocaml
text
```

interpreting and playing with hello.ml:

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

A First OCaml Program

hello.ml:

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

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:

```ocaml
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

sumTo8.ml:

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

let _ =
  print_int (sumTo 8);
  print_newline()
(* sum the numbers from 0 to n
  precondition: n must be a natural number
*)

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

let _ =
  print_int (sumTo 8);
  print_newline()
sumTo8.ml:

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

let _ =
  print_int (sumTo 8);
  print_newline()
(* sum the numbers from 0 to n
   precondition: n must be a natural number
 *)

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

let _ =
  print_int (sumTo 8);
  print_newline()
(* sum the numbers from 0 to n
  precondition: n must be a natural number
*)

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

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

sumTo8.ml:

(* 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()
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()
OCAML BASICS:
EXPRESSIONS, VALUES, SIMPLE 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

<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, -1.0, 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
42 * (13 + 1)
42 * (13 + 1) \rightarrow* 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
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 --- ie, I’m telling you something about the *operational semantics* of OCaml

More on semantics later.
Evaluation

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


int_of_char 'a' -->* 97

"moo" ^ "cow" -->* "moocow"

if true then 3 else 4 -->* 3

print_int 3 -->* ()
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 talk on Javascript:
https://www.destroyallsoftware.com/talks/wat
OCAML BASICS:
CORE EXPRESSION SYNTAX
The simplest OCaml expressions $e$ are:

- **values**
- **id**
- $e_1 \text{ op } e_2$
- $\text{id } e_1 \text{ e}_2 \ldots \text{ e}_n$
- **let** $\text{id } = \text{e}_1 \text{ in } \text{e}_2$
- **if** $\text{e}_1 \text{ then } \text{e}_2 \text{ else } \text{e}_3$
- (e)
- (e : t)

\text{numbers, strings, bools, ...}
\text{variables (x, foo, ...)}
\text{operators (x+3, ...)}
\text{function call (foo 3 42)}
\text{local variable decl.}
\text{a conditional}
\text{a parenthesized expression}
\text{an expression with its type}
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,

\[ f \ x \ y \ z \quad \text{same as:} \quad ((f \ x) \ y) \ z \]
\[ \quad \text{not the same as:} \quad f \ x \ (y \ z) \]

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:

- \( 2 : \text{int} \)
- \( "\text{hello}" : \text{string} \)
- \( 2 + 2 : \text{int} \)
- \( \text{"I say "} ^ {\text{"hello"}} : \text{string} \)
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 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:

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

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

(6) if $e : \text{int}$  
    then $\text{string_of_int } e : \text{string}$
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 \(e_1 \uparrow e_2 : \text{string}\)

(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) "\(abc\)" : 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 \(3 : \text{int}\).  
(\(\text{By rule } 1\))

Therefore, \((2 + 3) : \text{int}\)  
(\(\text{By rule } 3\))
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 \(\text{string_of_int}(e) : \text{string}\)

Using the rules:

\[2 : \text{int} \text{ and } 3 : \text{int}.\] (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. "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 \(e_1 : \text{string}\) and \(e_2 : \text{string}\) then \(e_1 ^ e_2 : \text{string}\)

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

Using the rules:

\[
\begin{align*}
2 : \text{int} \quad & \text{and} \quad 3 : \text{int}. \quad \text{(By rule 1)} \\
\text{Therefore,} \quad (2 + 3) : \text{int} \quad \text{(By rule 3)} \\
5 : \text{int} \quad & \text{ } \quad \text{(By rule 1)} \\
\text{Therefore,} \quad (2 + 3) \ast 5 : \text{int} \quad \text{(By rule 4 and our previous work)}
\end{align*}
\]
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 ????
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. $0 : \text{int}$ (and similarly for any other integer constant $n$)
2. "$\text{abc}" : \text{string}$ (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 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
   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;;
```

(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
   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
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. $0 : \text{int}$ (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 string_of_int $e : \text{string}$

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:

```plaintext
# "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
One of the keys to becoming a good ML programmer is to understand type error messages.
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?
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 / (run_turing_machine(); 0);;
```

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
OVERALL SUMMARY:
A SHORT INTRODUCTION TO
FUNCTIONAL PROGRAMMING
OCaml

OCaml is a *functional* programming language

- express control flow and iteration by defining *functions*

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*