



# 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?

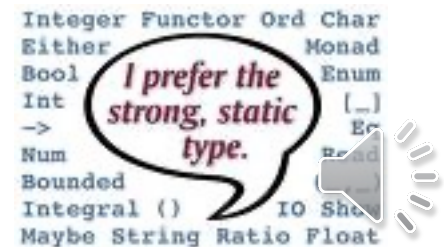
**Statically typed:** debugging and testing aid

- 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



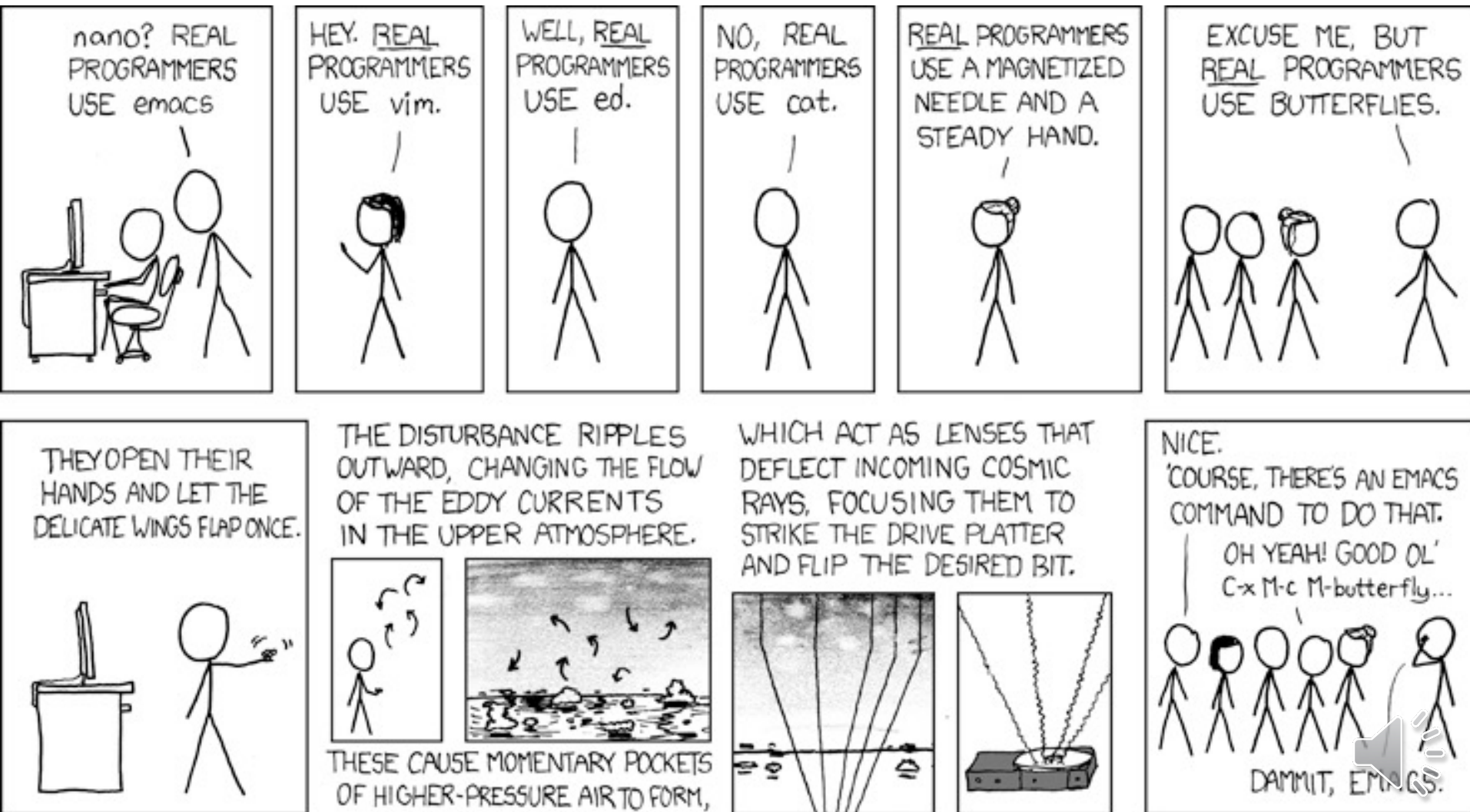
# Editing OCaml Programs

- 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

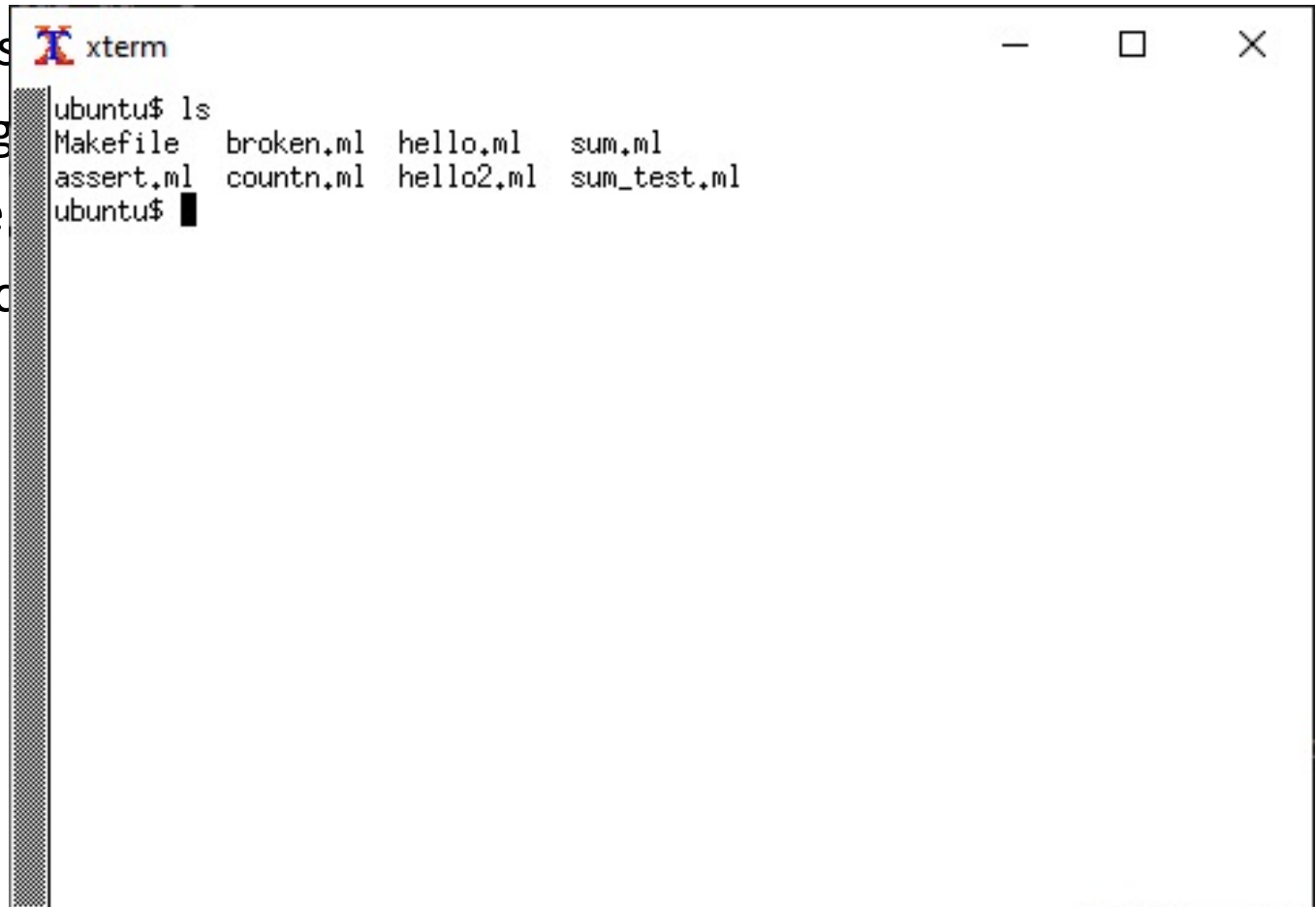


# **AN INTRODUCTORY EXAMPLE (OR TWO)**



# OCaml Compiler and Interpreter

- Demo:
  - emacs
  - ml files
  - writing
  - simple
  - ocamlc

A screenshot of an xterm window titled 'xterm'. The window shows a terminal session on a Ubuntu system. The user has entered the command 'ls' to list the contents of the current directory. The output shows a 'Makefile' and several OCaml files: 'broken.ml', 'hello.ml', 'sum.ml', 'assert.ml', 'countn.ml', 'hello2.ml', and 'sum\_test.ml'. The prompt 'ubuntu\$' is visible at the end of the command line.

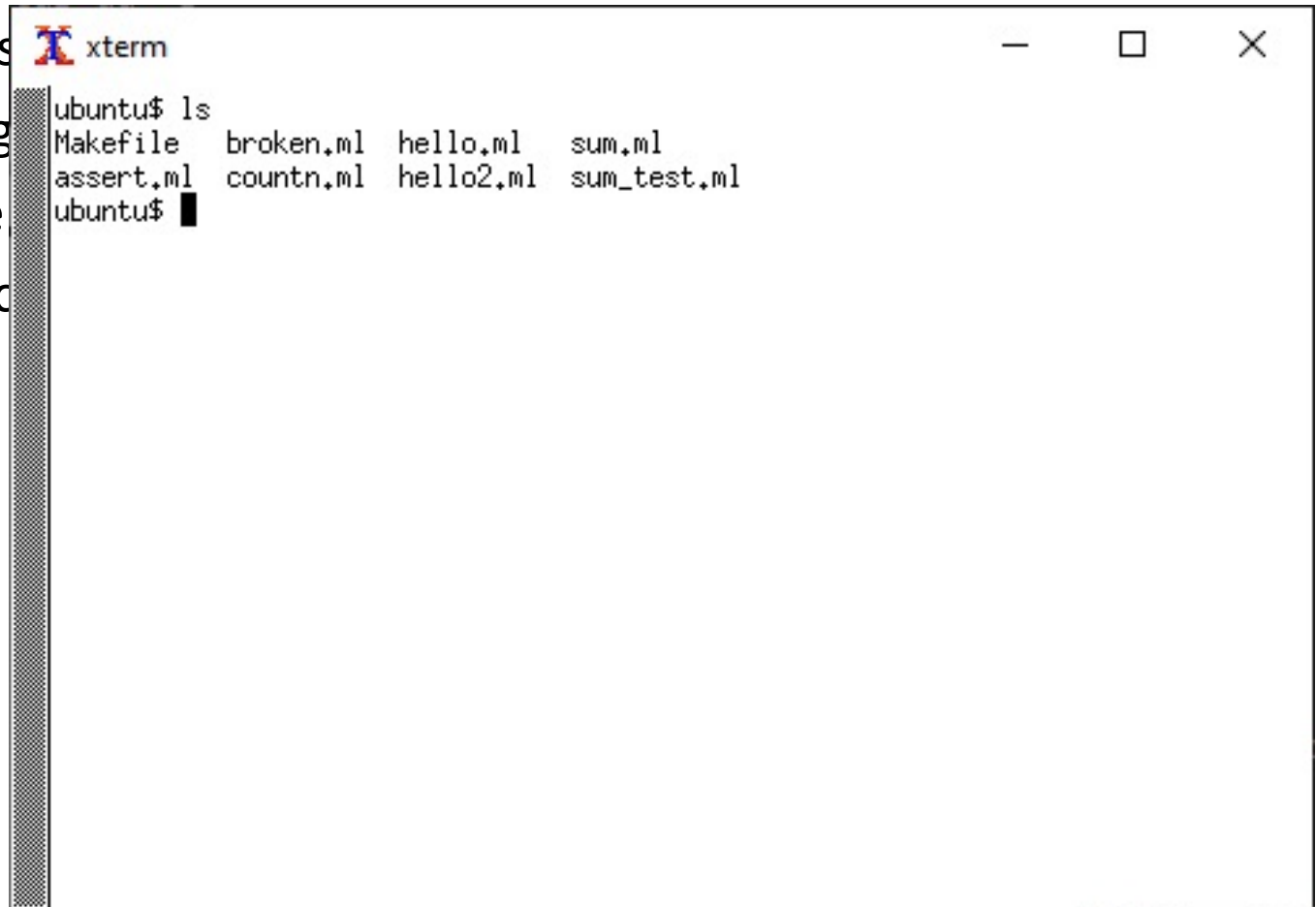
```
xterm
ubuntu$ ls
Makefile  broken.ml  hello.ml  sum.ml
assert.ml countn.ml  hello2.ml sum_test.ml
ubuntu$
```





# OCaml Compiler and Interpreter

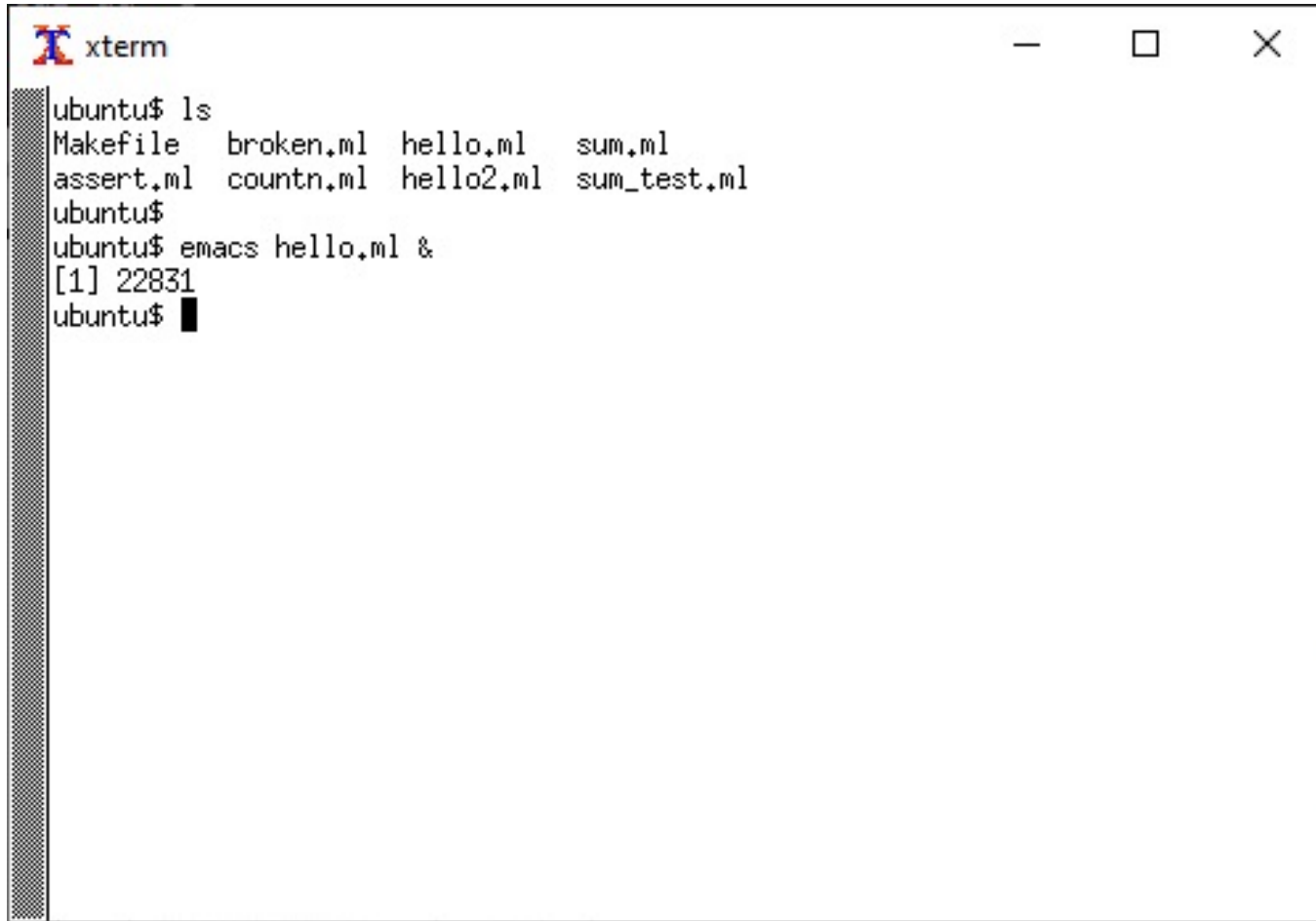
- Demo:
  - emacs
  - ml files
  - writing
  - simple
  - ocamlc

A screenshot of an xterm window titled 'xterm'. The window shows a terminal session on a system named 'ubuntu'. The user has entered the command 'ls', and the output lists the following files: 'Makefile', 'broken.ml', 'hello.ml', 'sum.ml', 'assert.ml', 'countn.ml', 'hello2.ml', and 'sum\_test.ml'. The prompt 'ubuntu\$' is visible at the end of the line.

```
xterm
ubuntu$ ls
Makefile  broken.ml  hello.ml   sum.ml
assert.ml countn.ml  hello2.ml  sum_test.ml
ubuntu$
```



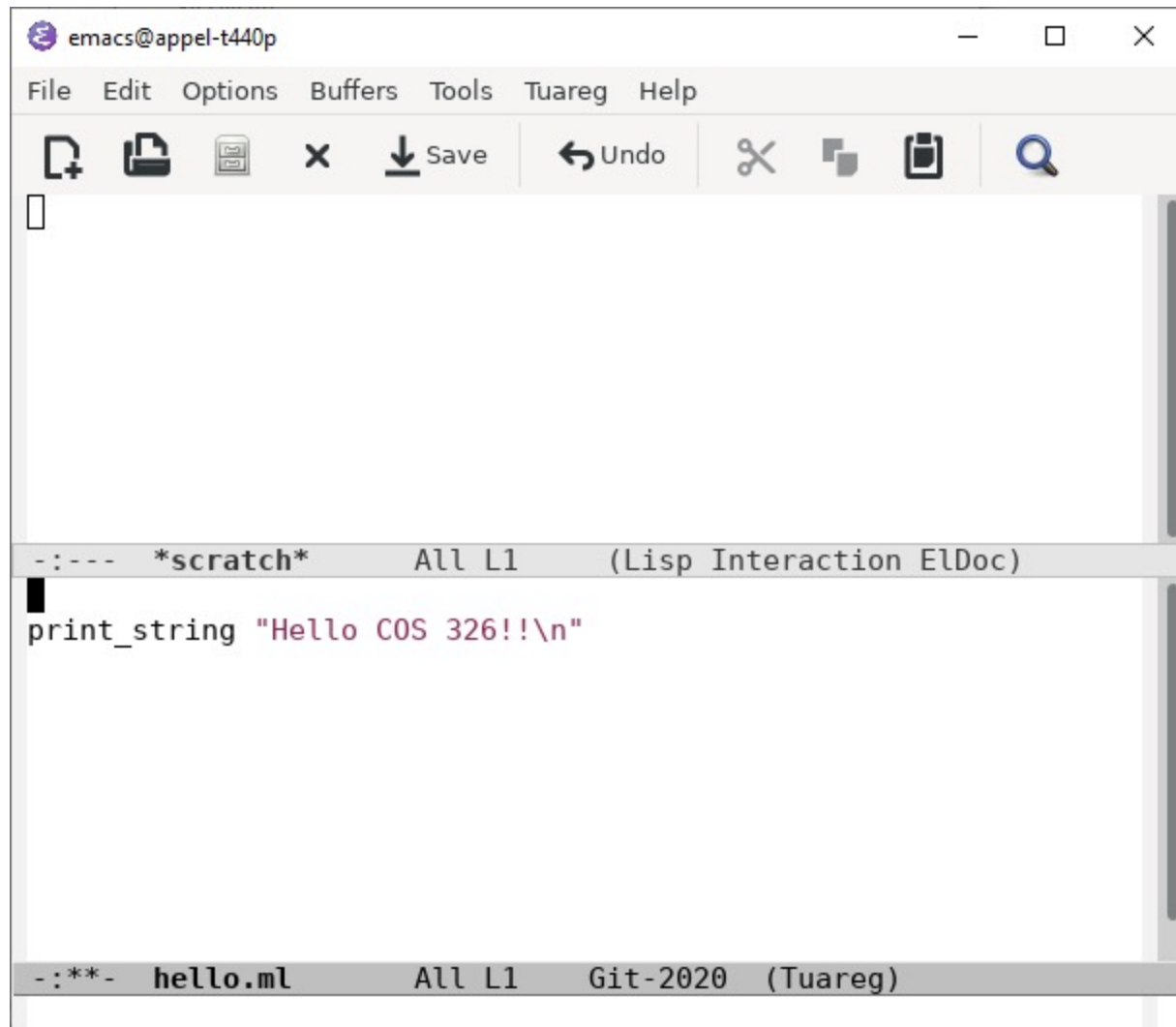
# OCaml demo



```
xterm
ubuntu$ ls
Makefile  broken.ml  hello.ml  sum.ml
assert.ml  countn.ml  hello2.ml  sum_test.ml
ubuntu$
ubuntu$ emacs hello.ml &
[1] 22831
ubuntu$
```



# OCaml demo

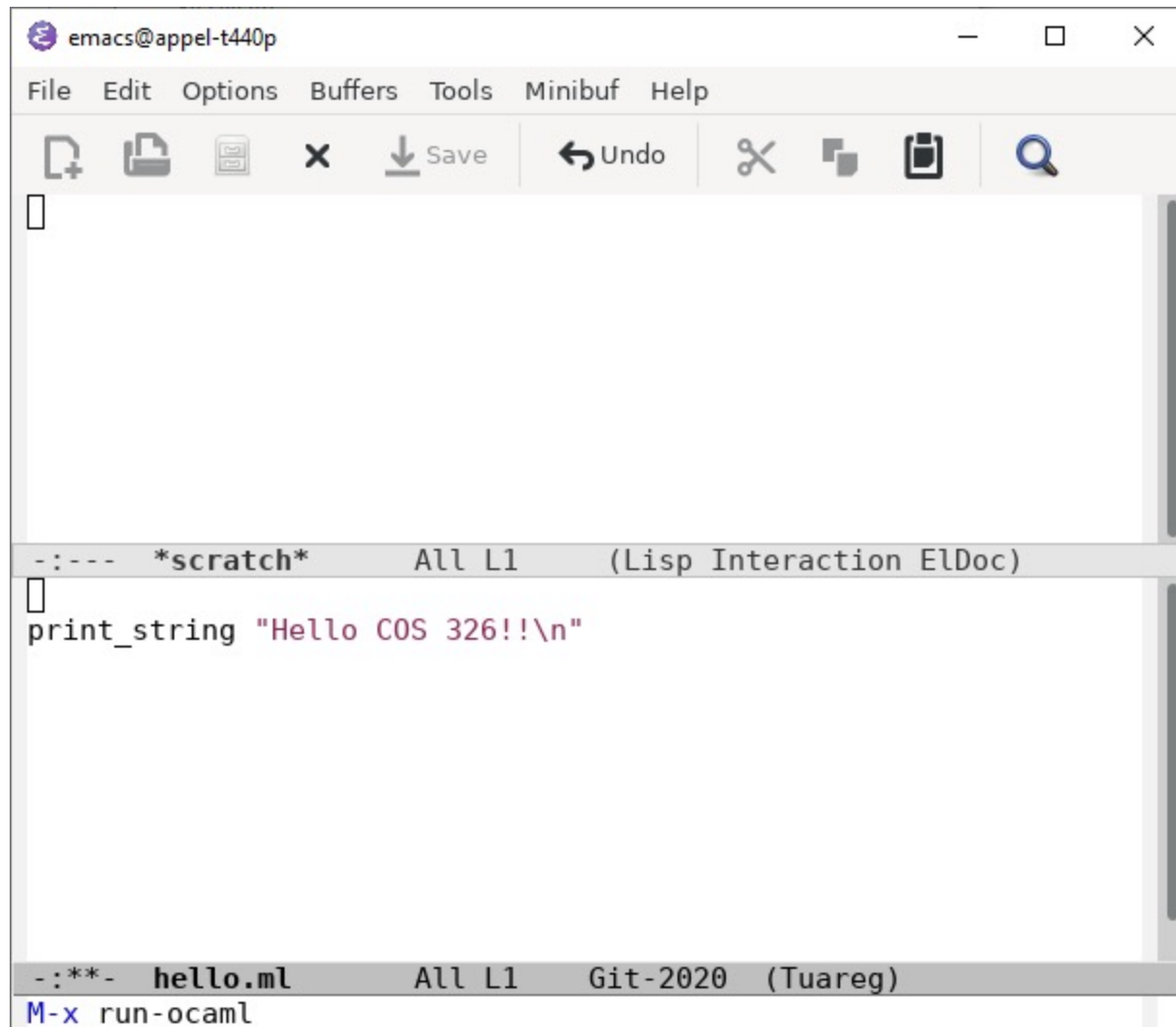


The screenshot shows the Emacs editor interface. The title bar indicates the user is 'emacs@appel-t440p'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Tuareg', and 'Help'. The toolbar contains icons for opening files, saving, undo, redo, and search. The main editing area is empty. The bottom buffer, titled '\*scratch\*', shows the code 'print\_string \"Hello COS 326!!\\n\"' in a Lisp interaction window. The status bar at the bottom indicates the file 'hello.ml' is open, with 'All L1' and 'Git-2020 (Tuareg)'.

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



# OCaml demo

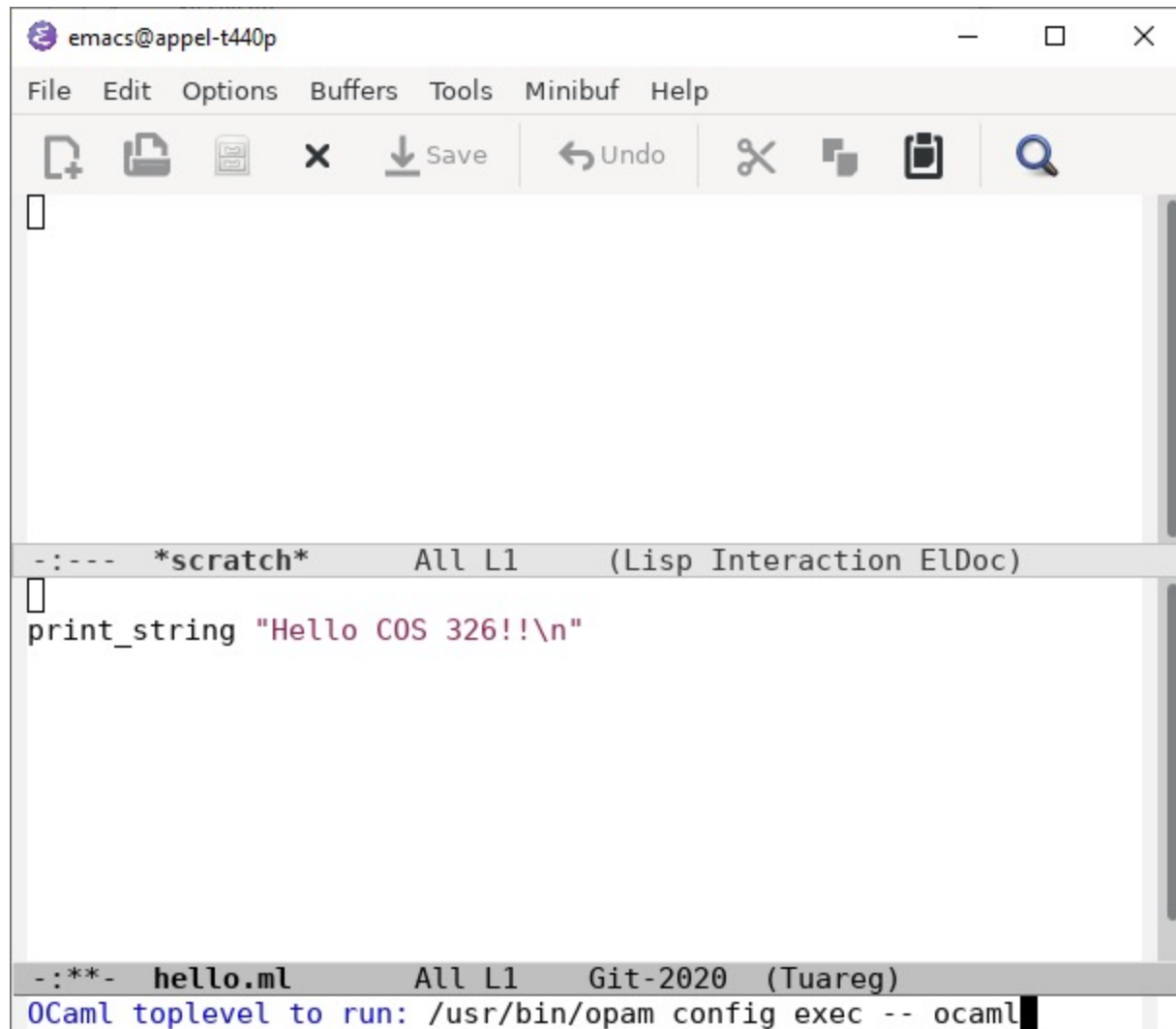


The screenshot shows an Emacs window titled 'emacs@appel-t440p'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Minibuf', and 'Help'. The toolbar contains icons for opening files, saving, undo, redo, and search. The main editing area is empty. Below it is a buffer titled '---- \*scratch\* All L1 (Lisp Interaction ElDoc)' containing the OCaml code: `print_string "Hello COS 326!!\n"`. At the bottom is another buffer titled '--- hello.ml All L1 Git-2020 (Tuareg)' containing the command `M-x run-ocaml`.

```
emacs@appel-t440p
File Edit Options Buffers Tools Minibuf Help
[Icons: Open, Save, Undo, Redo, Search]
[Empty editing area]
---- *scratch* All L1 (Lisp Interaction ElDoc)
print_string "Hello COS 326!!\n"
--- hello.ml All L1 Git-2020 (Tuareg)
M-x run-ocaml
```



# OCaml demo

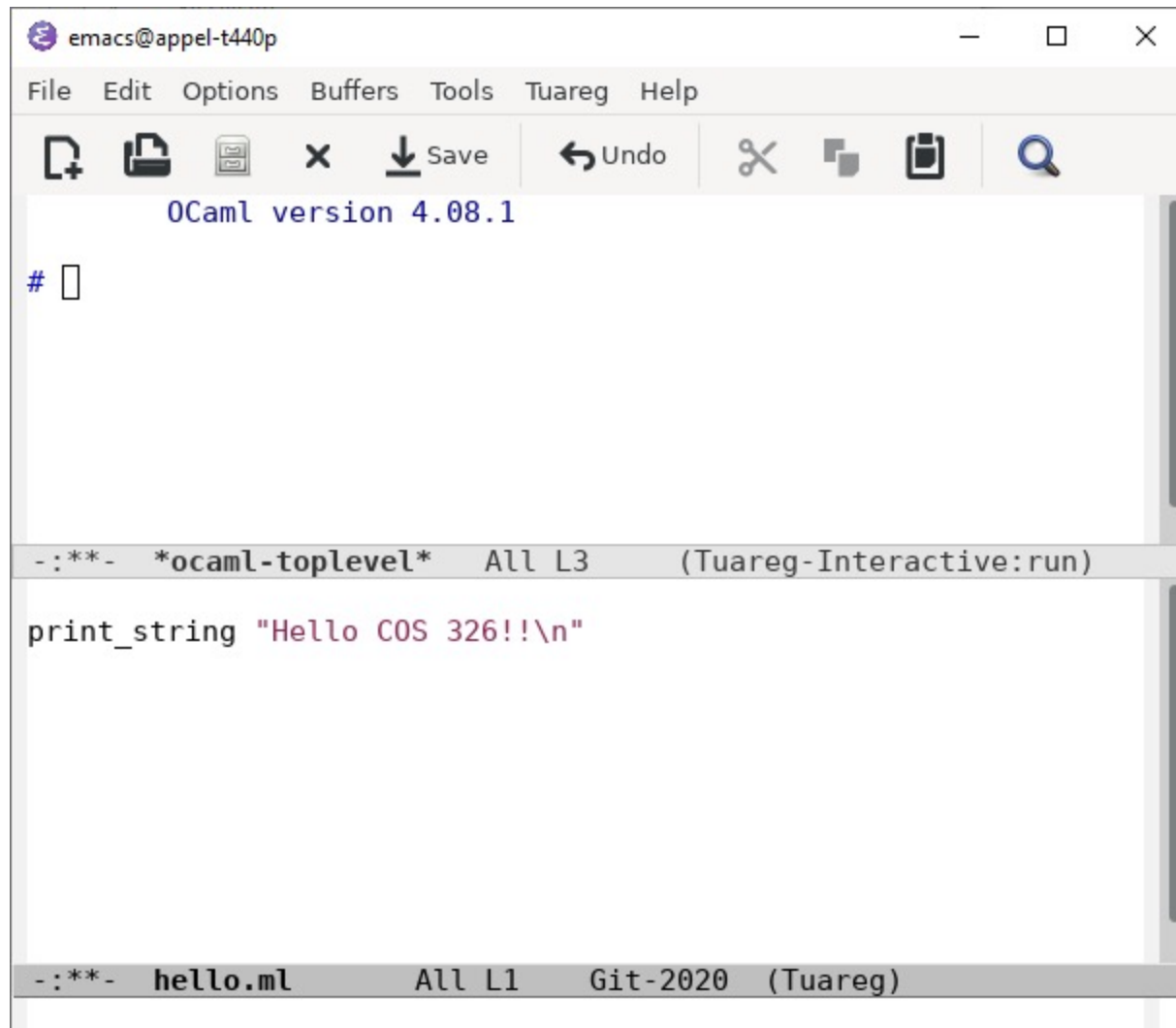


The screenshot shows an Emacs editor window titled 'emacs@appel-t440p'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Minibuf', and 'Help'. The toolbar contains icons for opening, saving, undo, redo, and search. The main editing area is empty. Below it is a buffer window titled '\*scratch\*' with the text 'All L1 (Lisp Interaction ElDoc)'. The bottom-most buffer window is titled 'hello.ml' with the text 'All L1 Git-2020 (Tuareg)'. The REPL output in the bottom window shows the command 'OCaml toplevel to run: /usr/bin/opam config exec -- ocaml' followed by a prompt and the command 'print\_string "Hello COS 326!!\n"' which has been executed.

```
emacs@appel-t440p
File Edit Options Buffers Tools Minibuf Help
[Icons: Open, Save, Undo, Redo, Search]
[Empty Buffer]
-:--- *scratch* All L1 (Lisp Interaction ElDoc)
[Empty Buffer]
-:***- hello.ml All L1 Git-2020 (Tuareg)
OCaml toplevel to run: /usr/bin/opam config exec -- ocaml
[Empty Buffer]
[Empty Buffer]
print_string "Hello COS 326!!\n"
```



# OCaml demo



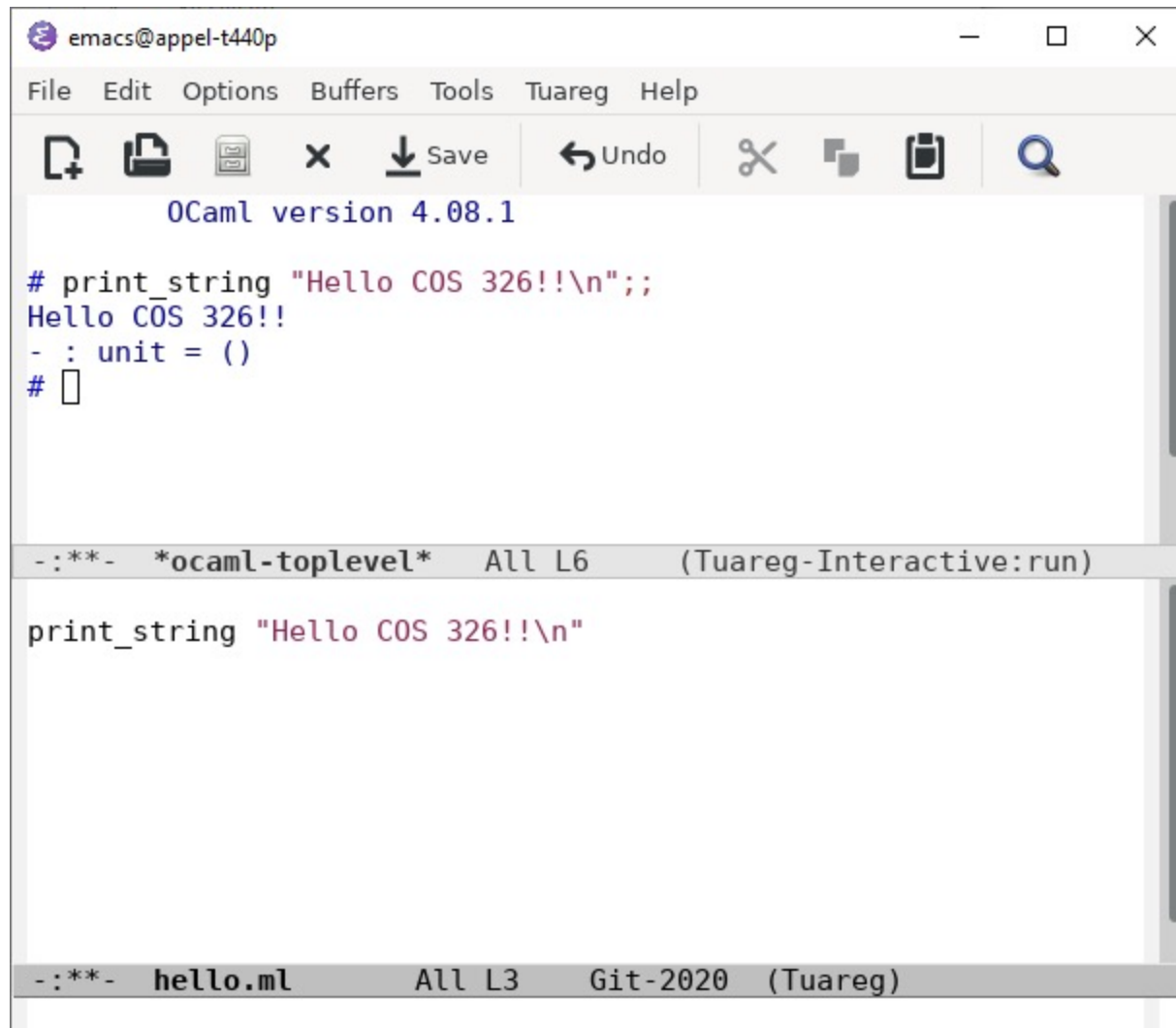
The screenshot shows the Emacs editor window titled "emacs@appel-t440p". The menu bar includes "File", "Edit", "Options", "Buffers", "Tools", "Tuareg", and "Help". The toolbar contains icons for opening, saving, undo, redo, and search. The main text area displays "OCaml version 4.08.1" and a prompt "# ". Below the text area, there are two buffer status lines. The first line is "-:\*\*\*- \*ocaml-toplevel\* All L3 (Tuareg-Interactive:run)" and the second line is "-:\*\*\*- hello.ml All L1 Git-2020 (Tuareg)". The code "print\_string \"Hello COS 326!!\\n\"" is visible in the buffer.

```
emacs@appel-t440p
File Edit Options Buffers Tools Tuareg Help
[Icons: Open, Save, Undo, Redo, Search]
OCaml version 4.08.1
# 
-:***- *ocaml-toplevel* All L3 (Tuareg-Interactive:run)
print_string "Hello COS 326!!\n"
-:***- hello.ml All L1 Git-2020 (Tuareg)
```





# OCaml demo



The screenshot shows an Emacs window titled 'emacs@appel-t440p'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Tuareg', and 'Help'. The toolbar contains icons for opening, saving, undo, redo, and search. The main text area displays OCaml code: 'OCaml version 4.08.1', a comment '# print\_string "Hello COS 326!!\n";;', the function call 'Hello COS 326!!', and a prompt '- : unit = ()' followed by a cursor. Below the text area is a status bar with two entries: '- :\*\*- \*ocaml-toplevel\* All L6 (Tuareg-Interactive:run)' and '- :\*\*- hello.ml All L3 Git-2020 (Tuareg)'. The second entry is highlighted, showing the file 'hello.ml' at line 3.

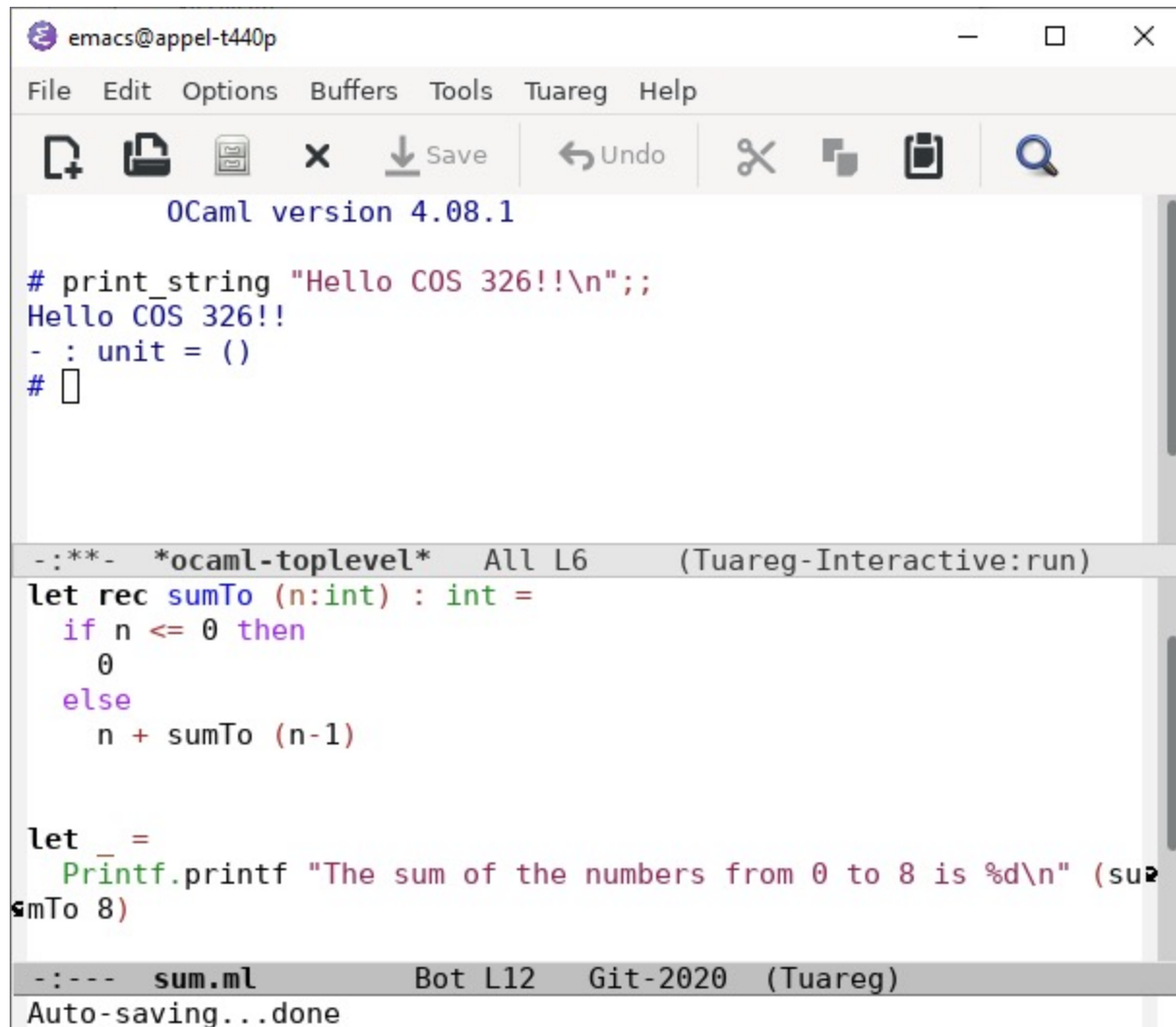
```
emacs@appel-t440p
File Edit Options Buffers Tools Tuareg Help
[Icons: Open, Save, Undo, Redo, Search]
OCaml version 4.08.1
# print_string "Hello COS 326!!\n";;
Hello COS 326!!
- : unit = ()
# █

- :**- *ocaml-toplevel* All L6 (Tuareg-Interactive:run)
print_string "Hello COS 326!!\n"

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



# OCaml demo



The screenshot shows an Emacs editor window titled 'emacs@appel-t440p'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Tuareg', and 'Help'. The toolbar contains icons for opening, saving, undo, redo, and search. The main text area displays OCaml code with syntax highlighting. Below the code is a Tuareg-Interactive:run buffer showing the execution of the code. At the bottom, a status bar indicates the current file is 'sum.ml' and it is being auto-saved.

```

OCaml version 4.08.1

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

-:***- *ocaml-toplevel* All L6 (Tuareg-Interactive:run)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)

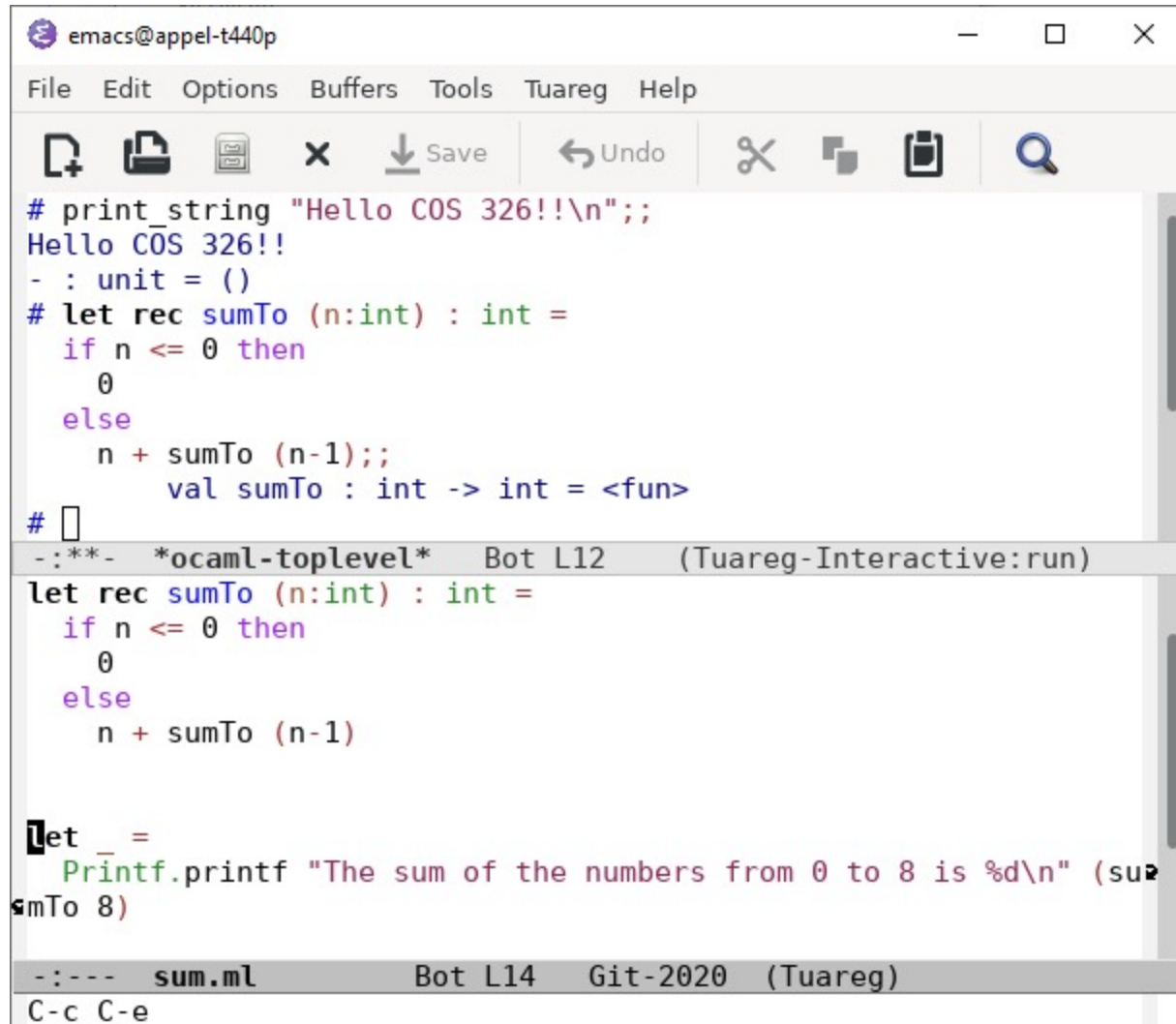
let _ =
  Printf.printf "The sum of the numbers from 0 to 8 is %d\n" (sumTo 8)

-:--- sum.ml Bot L12 Git-2020 (Tuareg)
Auto-saving...done

```



# OCaml demo



The screenshot shows an Emacs editor window titled 'emacs@appel-t440p'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Tuareg', and 'Help'. The toolbar contains icons for opening, saving, undo, redo, and search. The main text area contains OCaml code:

```
# 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>
#
```

Below the code, the Tuareg interface shows the execution of the code in a buffer named 'ocaml-toplevel'. The prompt is '-:\*\*\*-'. The code being executed is:

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

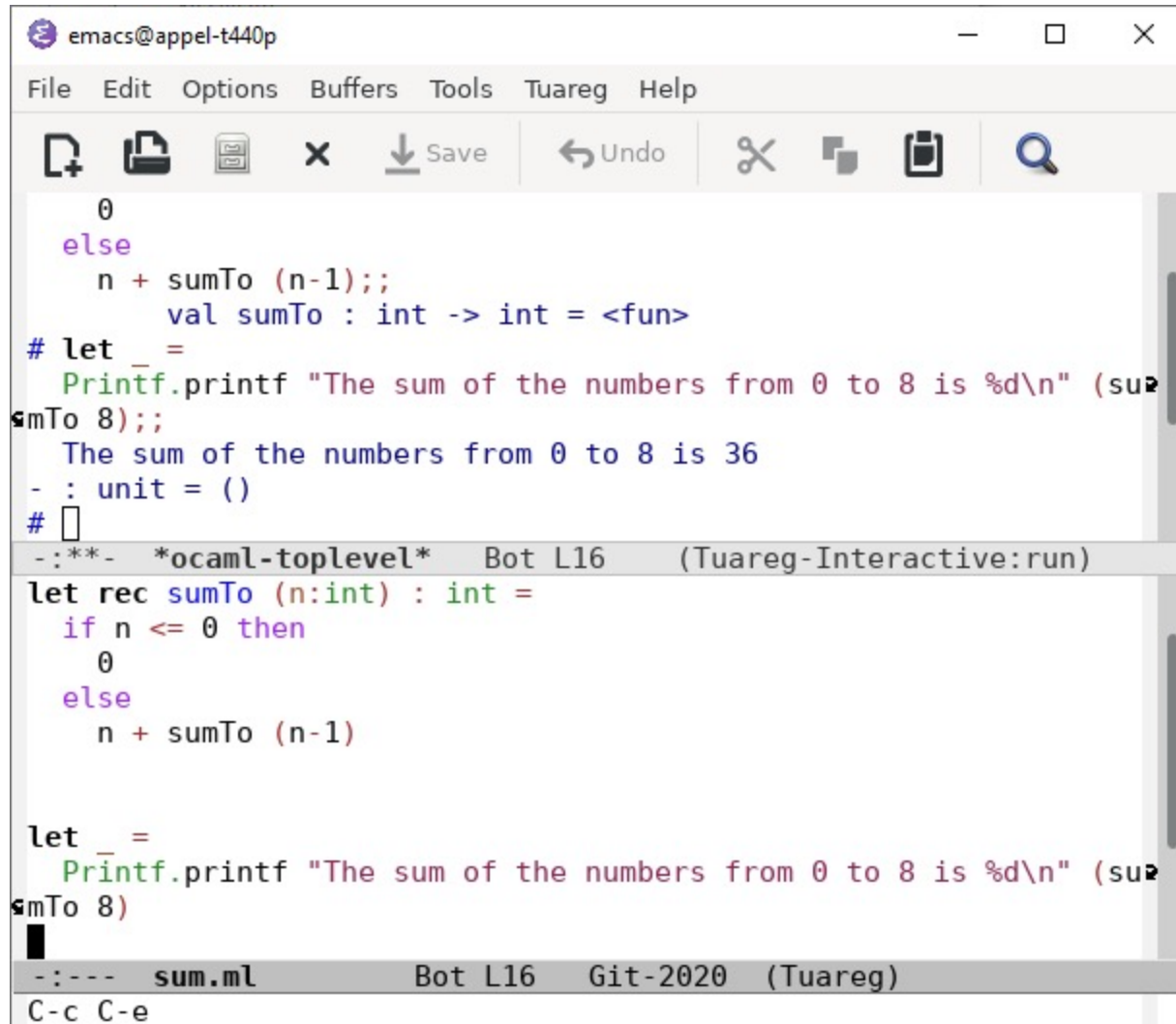
The output of the execution is:

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

The bottom status bar shows the file name 'sum.ml', the buffer name 'Bot L14', the version 'Git-2020', and the mode '(Tuareg)'. The keyboard shortcuts 'C-c C-e' are also displayed.



# OCaml demo



The screenshot shows an Emacs window titled 'emacs@appel-t440p'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Tuareg', and 'Help'. The toolbar contains icons for opening, saving, undo, redo, and search. The main text area displays OCaml code for a recursive function 'sumTo' and its execution. The code defines 'sumTo' as a recursive function that calculates the sum of numbers from 0 to 'n'. It then uses 'Printf.printf' to print the result of 'sumTo 8'. The output shows 'The sum of the numbers from 0 to 8 is 36'. Below the code, the Tuareg interface shows the execution of the code, with the prompt 'ocaml-toplevel\*' and the output 'The sum of the numbers from 0 to 8 is 36'. The status bar at the bottom indicates the file 'sum.ml' and the mode 'C-c C-e'.

```

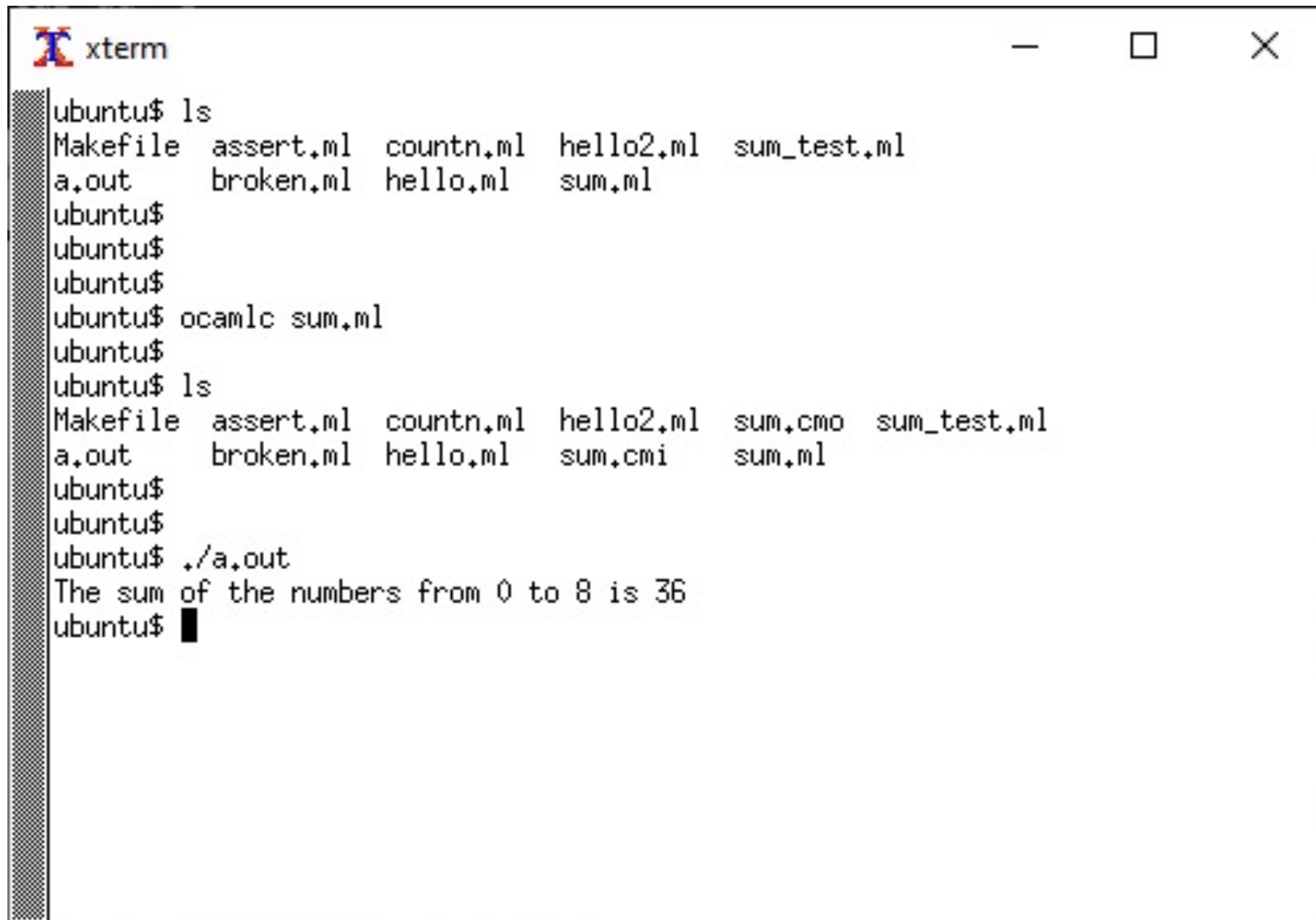
0
else
  n + sumTo (n-1);;
  val sumTo : int -> int = <fun>
# let _ =
  Printf.printf "The sum of the numbers from 0 to 8 is %d\n" (sumTo 8);;
The sum of the numbers from 0 to 8 is 36
- : unit = ()
#
-:***- *ocaml-toplevel* Bot L16 (Tuareg-Interactive:run)
let rec sumTo (n:int) : int =
  if n <= 0 then
    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 L16 Git-2020 (Tuareg)
C-c C-e

```



# OCaml demo

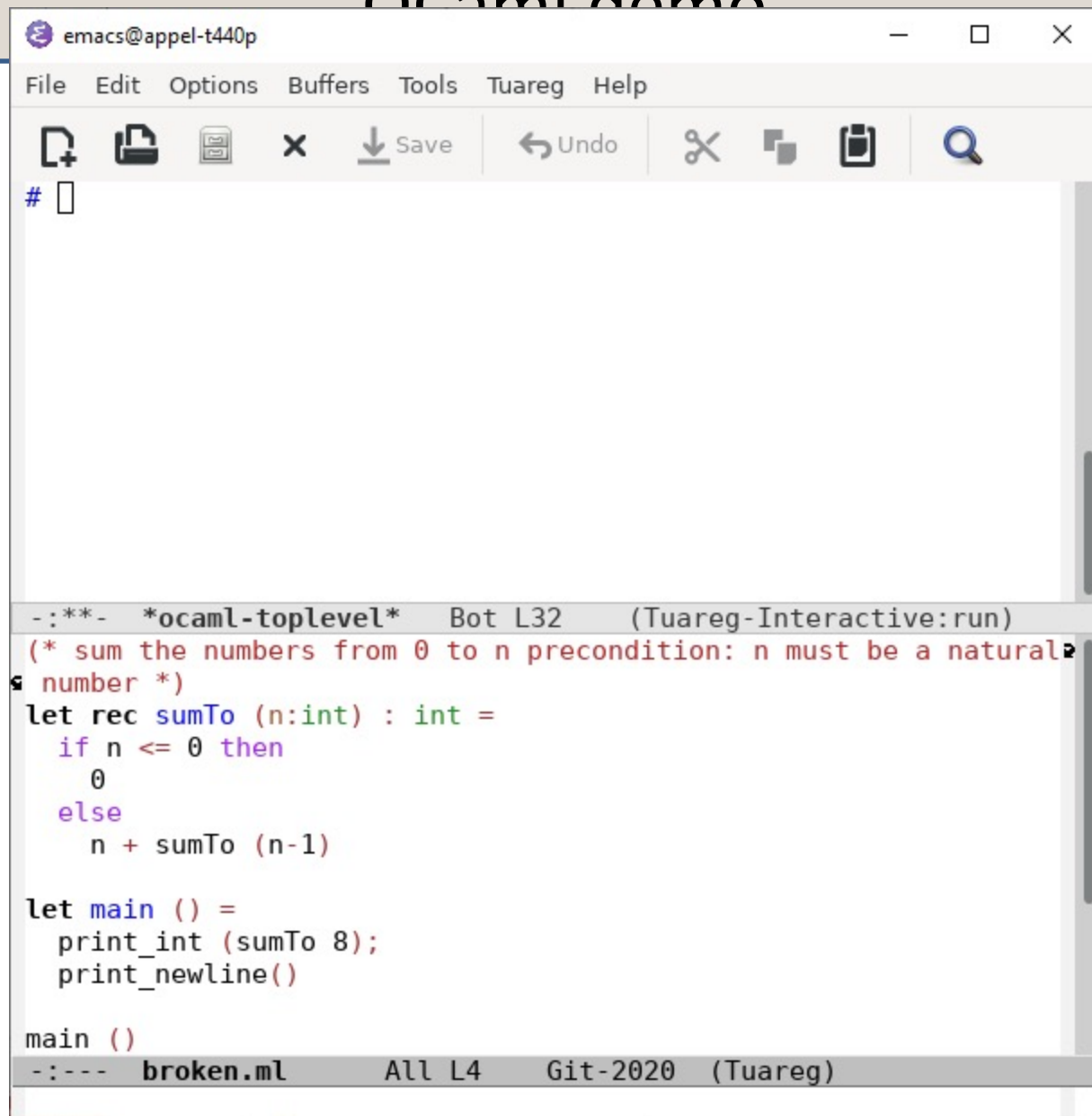


The image shows a terminal window titled 'xterm' with standard window controls (minimize, maximize, close). The terminal displays the following sequence of commands and output:

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



## OCaml demo



The screenshot shows an Emacs window titled 'emacs@appel-t440p'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Tuareg', and 'Help'. The toolbar contains icons for opening, saving, undo, redo, and search. The main text area shows a file named 'broken.ml' with the following OCaml code:

```
# []

-:***- *ocaml-toplevel* Bot L32 (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()

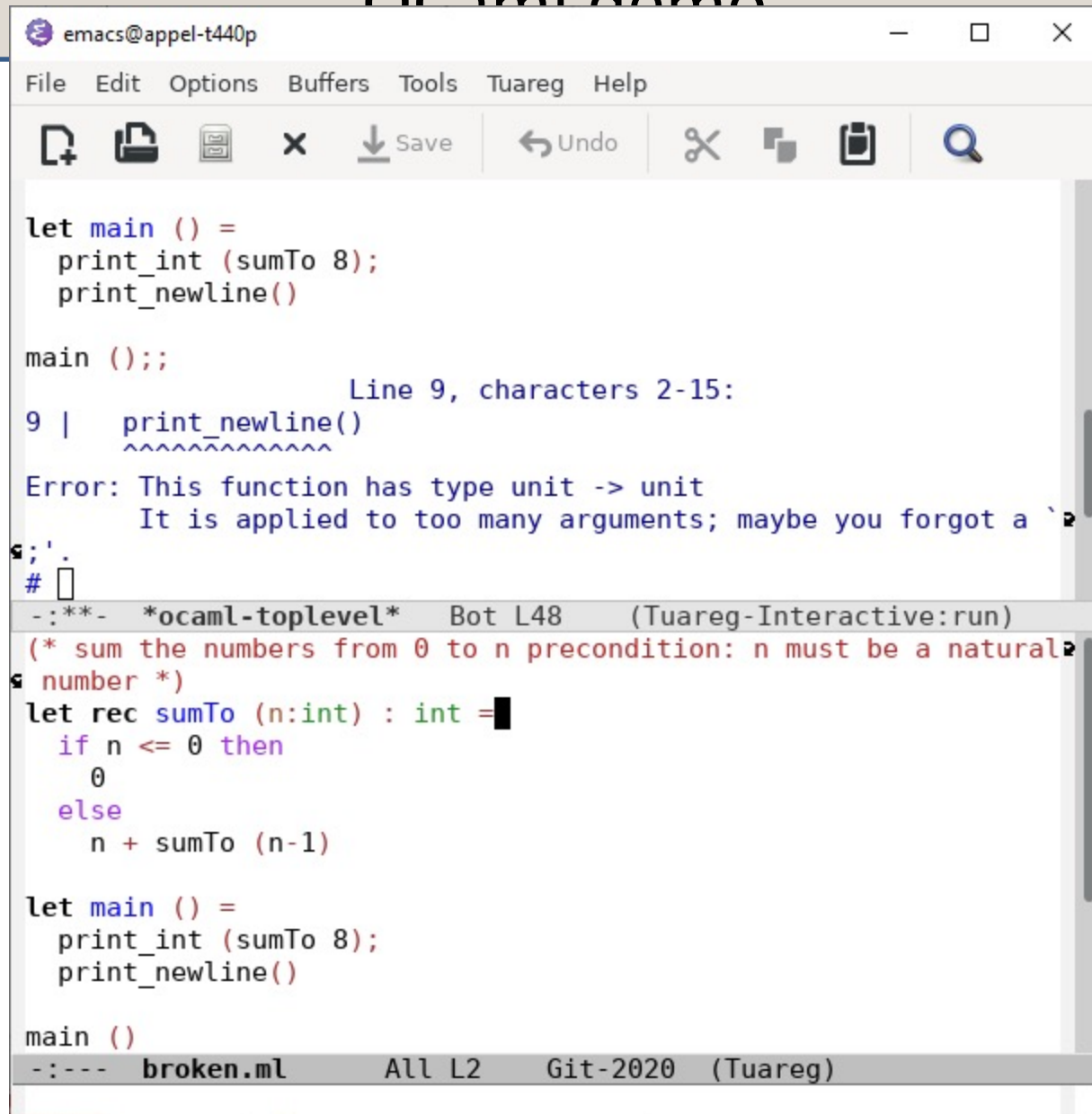
main ()

-:--- broken.ml All L4 Git-2020 (Tuareg)
```





## OCaml demo



The screenshot shows an Emacs window titled 'emacs@appel-t440p'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Tuareg', and 'Help'. The toolbar contains icons for opening, saving, undo, redo, and search. The main text area displays OCaml code with a syntax error. The code defines a 'main' function that calls 'print\_int' and 'print\_newline'. An error message indicates that 'print\_newline' is being called with too many arguments. Below the error, a snippet of the 'sumTo' function is visible, showing a recursive definition. The status bar at the bottom shows the current file is 'broken.ml' and the buffer is 'All L2'.

```

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()

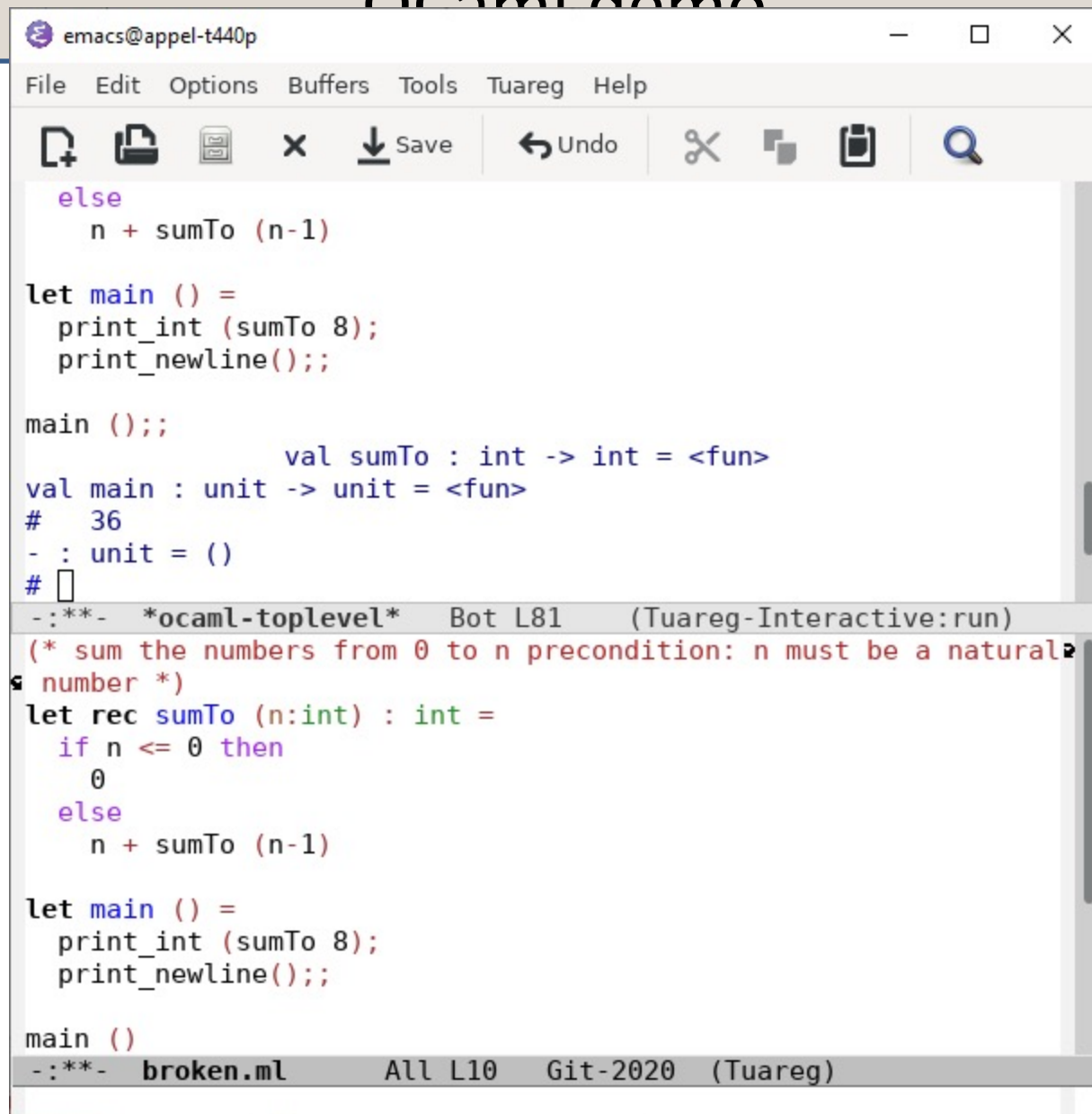
main ()

-:--- broken.ml All L2 Git-2020 (Tuareg)

```



## OCaml demo



The screenshot shows an Emacs window titled 'emacs@appel-t440p'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Tuareg', and 'Help'. The toolbar contains icons for opening files, saving, undo, redo, and search. The main text area displays OCaml code for a recursive sum function and its main entry point. The bottom status bar shows two buffers: '\*ocaml-toplevel\*' (Bot L81, Tuareg-Interactive:run) and 'broken.ml' (All L10, Git-2020, Tuareg). The REPL output shows the function signature and a call to 'sumTo 8'.

```

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)

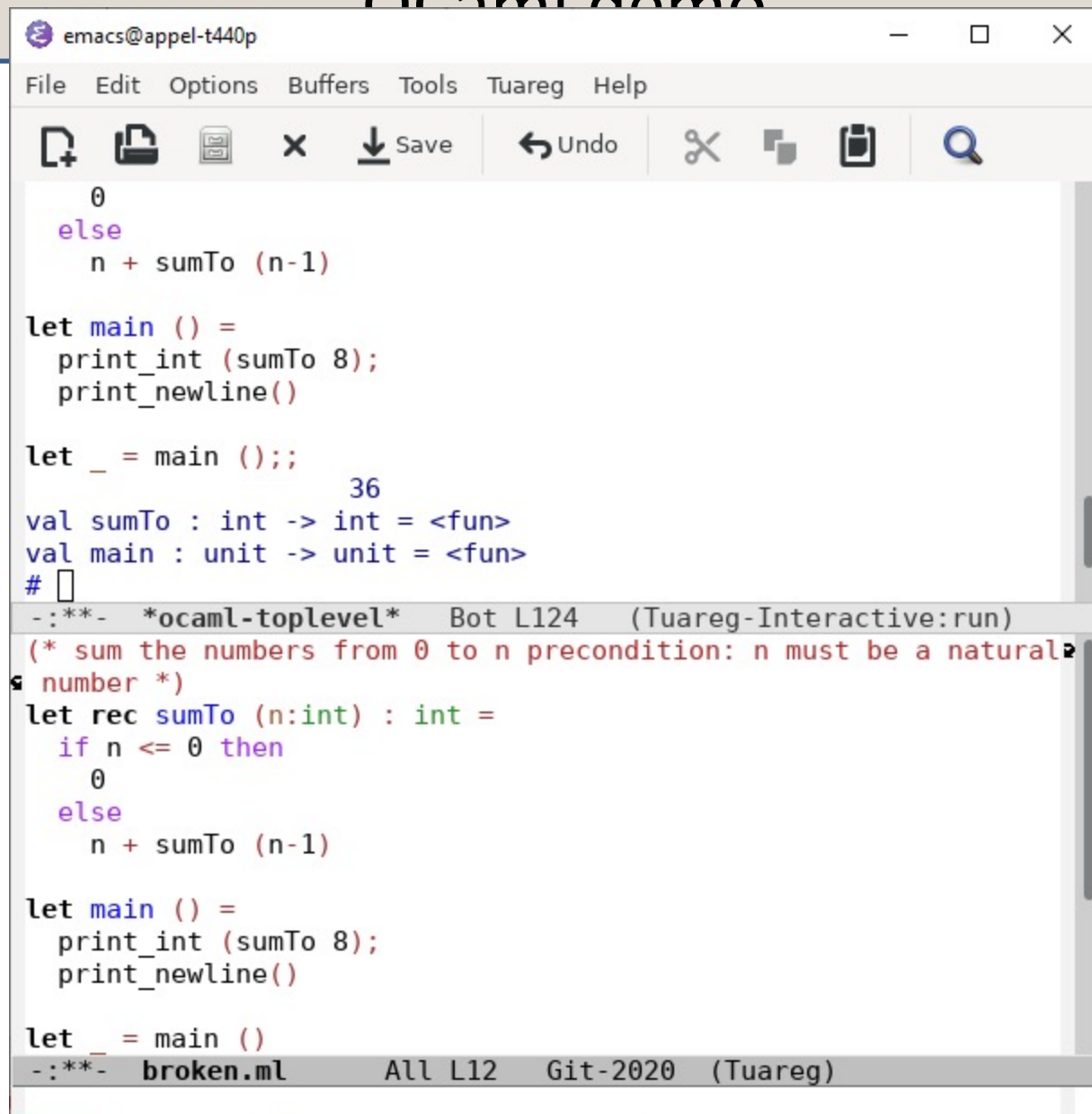
let main () =
  print_int (sumTo 8);
  print_newline();;

main ()
- : ** - broken.ml All L10 Git-2020 (Tuareg)

```



## OCaml demo



The screenshot shows an Emacs window titled 'emacs@appel-t440p'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Tuareg', and 'Help'. The toolbar contains icons for opening files, saving, undo, redo, and search. The main buffer displays OCaml code for a recursive sum function. The bottom buffer, 'Tuareg-Interactive:run', shows the execution of the code, including a comment about the precondition for the sum function and the definition of the recursive function.

```

0
else
  n + sumTo (n-1)

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

let _ = main ();;
36
val sumTo : int -> int = <fun>
val main : unit -> unit = <fun>
#

-:***- *ocaml-toplevel* Bot L124 (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)

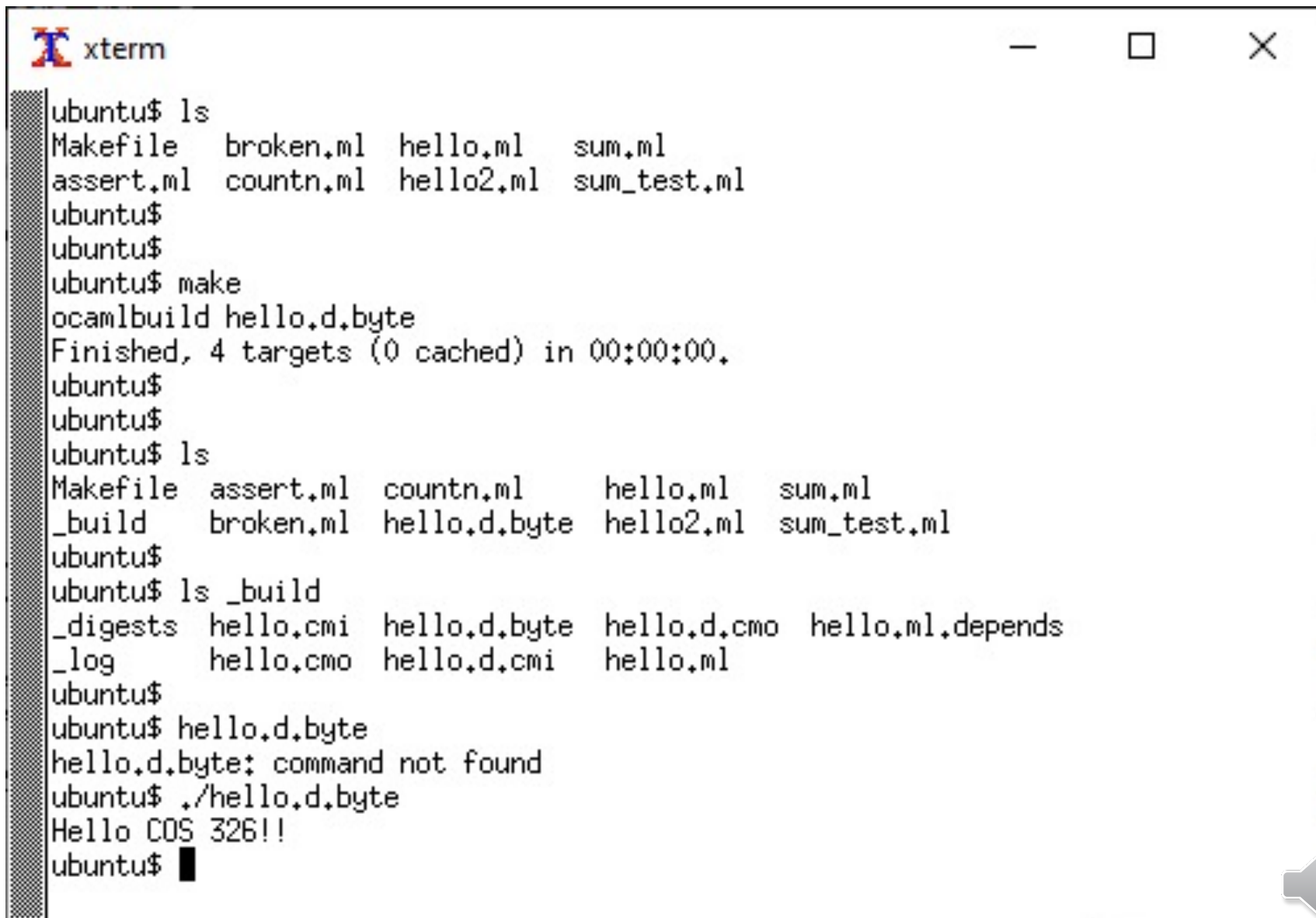
let main () =
  print_int (sumTo 8);
  print_newline()

let _ = main ()
-:***- broken.ml All L12 Git-2020 (Tuareg)

```



# OCaml demo

A screenshot of an xterm window titled 'xterm' with standard window controls (minimize, maximize, close). The terminal shows a series of commands and their outputs in a Ubuntu environment. The commands include 'ls', 'make', and running the generated 'hello.d.byte' file. The output shows the successful compilation of OCaml files and the execution of a 'Hello COS 326!!' message.

```
xterm
ubuntu$ ls
Makefile  broken.ml  hello.ml  sum.ml
assert.ml  countrn.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  countrn.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"
```



# A First OCaml Program

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:

```
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

sumTo8.ml:

a comment  
(\* ... \*)

```
(* 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 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



# **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:

<http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.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)  $\text{-->}$ \* 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

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

<code>42 * (13 + 1)</code>	<code>--&gt;*</code>	<code>588</code>
<code>(3.14 +. 12.0) *. 10e6</code>	<code>--&gt;*</code>	<code>151400000.</code>
<code>int_of_char 'a'</code>	<code>--&gt;*</code>	<code>97</code>
<code>"moo" ^ "cow"</code>	<code>--&gt;*</code>	<code>"moocow"</code>
<code>if true then 3 else 4</code>	<code>--&gt;*</code>	<code>3</code>
<code>print_int 3</code>	<code>--&gt;*</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 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, ...)*
- $e_1 \text{ op } e_2$  *operators (x+3, ...)*
- id  $e_1 e_2 \dots e_n$  *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` same as: `((f x) y) z`  
 not the same as: `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 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



# 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

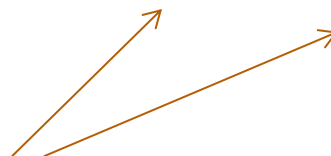
Example rules:

- (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

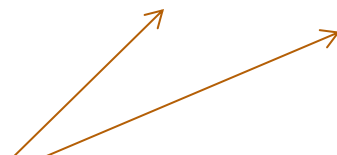
Example rules:

- (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

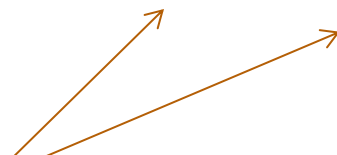
Example rules:

- (1) `0 : 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 * (add_one 17) : 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
#
```



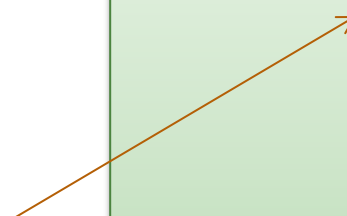


# Type Checking Rules

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



# 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`  
`1 : int`  
`1 + "hello" : ??`

(By rule 2)

(By rule 1)

(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 / (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*

