LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.

I WONDER IF THE CYCLES WILL CONTINUE FOREVER.

A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.

THESE ARE YOUR FATHER'S PARENTHESES

ELEGANT WEAPONS FOR A MORE... CIVILIZED AGE.

COS 326 Functional Programming:
An elegant weapon for the modern age
Andrew Appel
Princeton University

or mother's
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."

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."

Greatest technological understatement of the 20th century?

Alonzo Church, 1903-1995
Princeton Professor, 1929-1967
A few designers of functional programming languages

Alonzo Church: λ-calculus, 1934

John McCarthy
(PhD Princeton 1951)
LISP, 1958

Guy Steele & Gerry Sussman:
Scheme, 1975
A few designers of functional programming languages

Alonzo Church: λ-calculus, 1934

Robin Milner
ML, 1978

Appel & MacQueen: SML/NJ, 1988

Xavier Leroy: Ocaml, 1990’s
Vastly Abbreviated FP Genealogy

LCF Theorem Prover (70s) → Edinburgh ML

Miranda (80s) → Haskell (90s - now)

Standard ML (90s - now) → Caml (80s-now) → OCaml (90s - now)

Scala (00s - now) → F# (now)

LISP (1960-now) → Scheme (70s-now) → Racket (00s-now)

Coq (80s - now)

lazy

call-by-value

typed, polymorphic

typed

dependently typed

untyped

standard
**Vastly Abbreviated FP Genealogy**

- **LCF Theorem Prover (70s)**
  - Edinburgh ML
    - Miranda (80s)
    - Haskell (90s - now)
  - Standard ML (90s - now)
    - OCaml (90s - now)
    - Scala (00s - now)
    - F# (now)
  - Coq (80s - now)
- **LISP (50s - now)**
  - Scheme (70s - now)
  - Racket (00s - now)

- **Typed, Polymorphic**
- **Lazy**
- **Call-by-Value**
- **Untyped**
- **Dependently Typed**
Functional Languages: Who’s using them?

map-reduce in their data centers

map - reduce in their data centers

Scala for correctness, maintainability, flexibility

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

Microsoft

F# in Visual Studio

Haskell for specifying equity derivatives

Bluespec

Haskell to synthesize hardware

mathematicians

Coq (re)proof of 4-color theorem

Facebook

Barclays

Haskell for specifying equity derivatives

Twitter

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.haskell.org/haskellwiki/Haskell_in_industry
COURSE LOGISTICS
Course Staff

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

Joomy Korkut
Preceptor
office: CS 214
email: joomy@cs
Resources

• coursehome:
  – http://www.cs.princeton.edu/~cos326

• Lecture schedule and readings:
  – $(coursehome)/lectures.php

• Assignments:
  – $(coursehome)/assignments.php

• Precepts
  – useful if you want to do well on exams and homeworks

• Install OCaml: $(coursehome)/resources.php
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.
http://realworldocaml.org/
Exams

Midterm
• take-home during midterm week

Final
• during exam period in January
• 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”
Figure out how to download and install the latest version of OCaml on your machine by the time precept begins tomorrow. (or, how to use OCaml by ssh to Princeton University servers)

Resources Page:

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

Hint:

ocaml.org
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.
- We may need this experiment to be replicated a few more times to gain confidence in the result.
- Instagram and Fortnite distract your classmates.
A Functional Introduction
Thinking Functionally

In Java or C, you get (most) work done by *changing* something

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

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

In ML, you get (most) work done by *producing something new*

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

you *analyze* existing data (like pair) and you *produce* new data (y,x)
This simple switch in perspective can change the way you think about programming and problem solving.
Thinking Functionally

pure, functional code:

```plaintext
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:

```plaintext
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
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
Many options: pick your own poison

- Emacs
  - what I’ll be using in class.
  - good but not great support for OCaml.
  - I like it because it's what I'm used to
  - (extensions written in elisp – a functional language!)

- OCaml IDE
  - integrated development environment written in OCaml.
  - haven’t used it, so can’t comment.

- Eclipse
  - I’ve put up a link to an OCaml plugin
  - I haven't tried it but others recommend it

- Sublime, atom
  - A lot of students seem to gravitate to this
nano? REAL PROGRAMMERS USE emacs

HEY. REAL PROGRAMMERS USE vim.

WELL, REAL PROGRAMMERS USE ed.

NO, REAL PROGRAMMERS USE cat.

REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.

EXCUSE ME, BUT REAL PROGRAMMERS USE BUTTERFLIES.

THEY OPEN THEIR HANDS AND LET THE DELICATE WINGS FLAP ONCE.

THE DISTURBANCE RIPPLES OUTWARD, CHANGING THE FLOW OF THE EDDY CURRENTS IN THE UPPER ATMOSPHERE.

WHICH ACT AS LENSES THAT DEFLECT INCOMING COSMIC RAYS, FOCUSING THEM TO STRIKE THE DRIVE PLATTER AND FLIP THE DESIRED BIT.

NICE. ‘COURSE, THERE’S AN EMACS COMMAND TO DO THAT.

‘OH YEAH! GOOD OL’ C-x M-c M-butterfly...'

DAMMIT, EMACS.
AN INTRODUCTORY EXAMPLE (OR TWO)
OCaml Compiler and Interpreter

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

print_string "Hello COS 326!!\n";;
A First OCaml Program

hello.ml:

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

- a function
- its string argument enclosed in " . . . "
- no parens. normally call a function \( f \) like this:
  ```ocaml
  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:

```
$ 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)
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

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()
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()
(* 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.

```ocaml
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 *)

sumTo8.ml:
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()
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., 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)
Evaluation

\[ 42 \times (13 + 1) \rightarrow 588 \]

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

(3.14 + 12.0) * 10e6  -->*  151400000.

int_of_char 'a'  -->*  97

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

if true then 3 else 4  -->*  3

print_int 3  -->*  ()
1 + "hello" -->*  ???
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:

1. **values**
   - numbers, strings, bools, ...
2. **id**
   - variables (x, foo, ...)
3. **$e_1$ op $e_2$**
   - operators (x+3, ...)
4. **id $e_1$ $e_2$ ... $e_n$**
   - function call (foo 3 42)
5. **let id = $e_1$ in $e_2$**
   - local variable decl.
6. **if $e_1$ then $e_2$ else $e_3$**
   - a conditional
7. **(e)**
   - a parenthesized expression
8. **(e : t)**
   - an expression with its type
A note on parentheses

In most languages, arguments are parenthesized & separated by commas:

\[ f(x, y, z) \quad \text{sum}(3, 4, 5) \]

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

\[ f \ x \ y \ z \quad \text{sum} \ 3 \ 4 \ 5 \]

But we do have to worry about *grouping*. For example,

\[ f \ x \ y \ z \quad f \ x \ (y \ z) \]

The first one passes three arguments to \( f \ (x, y, \text{and } z) \)
The second passes two arguments to \( f \ (x, \text{and the result of applying the function } y \text{ to } z) \).
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:

\[
\begin{align*}
2 & : \text{int} & \quad \text{"hello" : string} \\
2 + 2 & : \text{int} & \quad \text{"I say " ^ "hello" : string}
\end{align*}
\]
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) \(0 : \text{int}\) (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 \(e_1 ^ e_2 : \text{string}\)

6. If \(e : \text{int}\) then \(\text{string}_\text{of}_\text{int} e : \text{string}\)
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 \(3 : \text{int}\).  
(By rule 1)
Type Checking Rules

Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "...")

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

(4) if e1 : int and e2 : int
    then e1 * e2 : int

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

(6) if e : int
    then string_of_int e : string

Using the rules:

2 : int and 3 : int. (By rule 1)
Therefore, (2 + 3) : int (By rule 3)
Example rules:

(1) 0 : int  (and similarly for any other integer constant n)

(2) "abc" : string  (and similarly for any other string constant "...")

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

(4) if e1 : int and e2 : int  
   then e1 * e2 : int

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

(6) if e : int  
   then string_of_int e : string

Using the rules:

2 : int and 3 : int.  (By rule 1)  
Therefore, (2 + 3) : int  (By rule 3)  
5 : int  (By rule 1)
Type Checking Rules

Example rules:

1. \( 0 : \text{int} \)  
   (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 \( e : \text{int} \)  
   then \( \text{string_of_int} e : \text{string} \)

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

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)

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

FYI: This is a formal proof that the expression is well-typed!
Type Checking Rules

Example rules:

(1) 0 : int  
(2) "abc" : string  
(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) \(0 : \text{int}\) (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 \(e_1 \ ^{\ W} e_2 : \text{string}\)

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

Another perspective:

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

rule (4) for typing expressions says I can put any expression with type \text{int} in place of the ????
Type Checking Rules

Example rules:

(1) 0 : 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 ????
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 : int  
    (and similarly for any other integer constant n)

(2) "abc" : string  
    (and similarly for any other string constant "...")

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

(4) if e1 : int and e2 : int 
    then e1 * e2 : int

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

(6) if e : int 
    then string_of_int e : string

Violating the rules:

"hello" : string  
    (By rule 2)

1 : int  
    (By rule 1)

1 + "hello" : ??  
    (NO TYPE! Rule 3 does not apply!)
Type Checking Rules

Violating the rules:

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

The type error message tells you the type that was expected and the type that it inferred for your subexpression.

By the way, this was one of the nonsensical expressions that did not evaluate to a value.

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

"Well typed programs do not go wrong"

Robin Milner, 1978
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++, Pascal
Well typed programs do not go wrong

Robin Milner

Turing Award, 1991

“For three distinct and complete achievements:

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

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

3. **CCS**, a general theory of concurrency.

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

“**Well typed programs do not go wrong**”

Robin Milner, 1978
OVERALL SUMMARY:
A SHORT INTRODUCTION TO FUNCTIONAL PROGRAMMING
OCaml

OCaml is a *functional* programming language

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

OCaml is a *typed* programming language

- the *type* of an expression *correctly predicts* the kind of *value* the expression will generate when it is executed
- types help us *understand* and *write* our programs
- the type system is *sound*; the language is *safe*