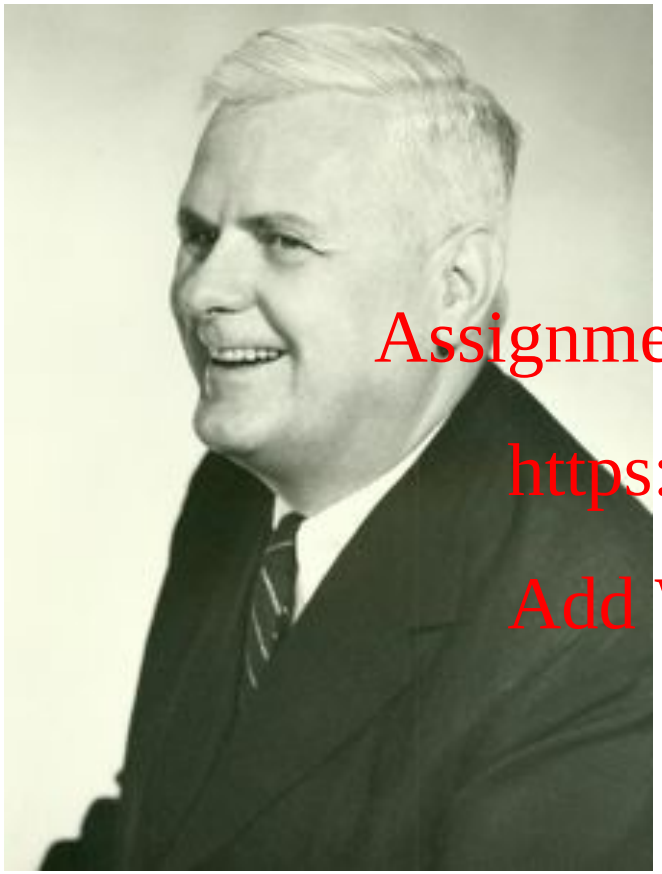
Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

INTRODUCTION TO OCAML

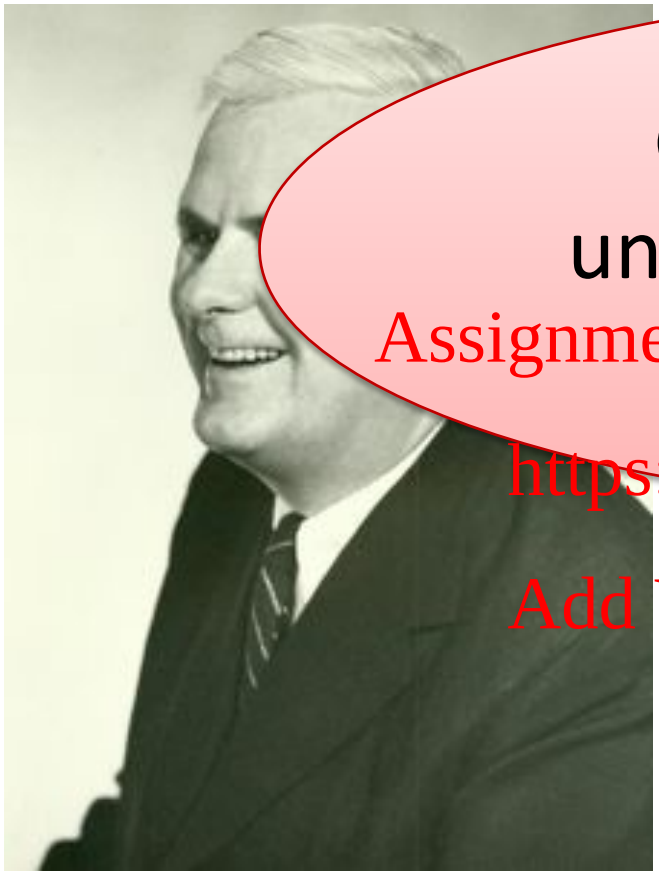


Alonzo Church, 1903-1995
Princeton Professor, 1929-1967

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.*"



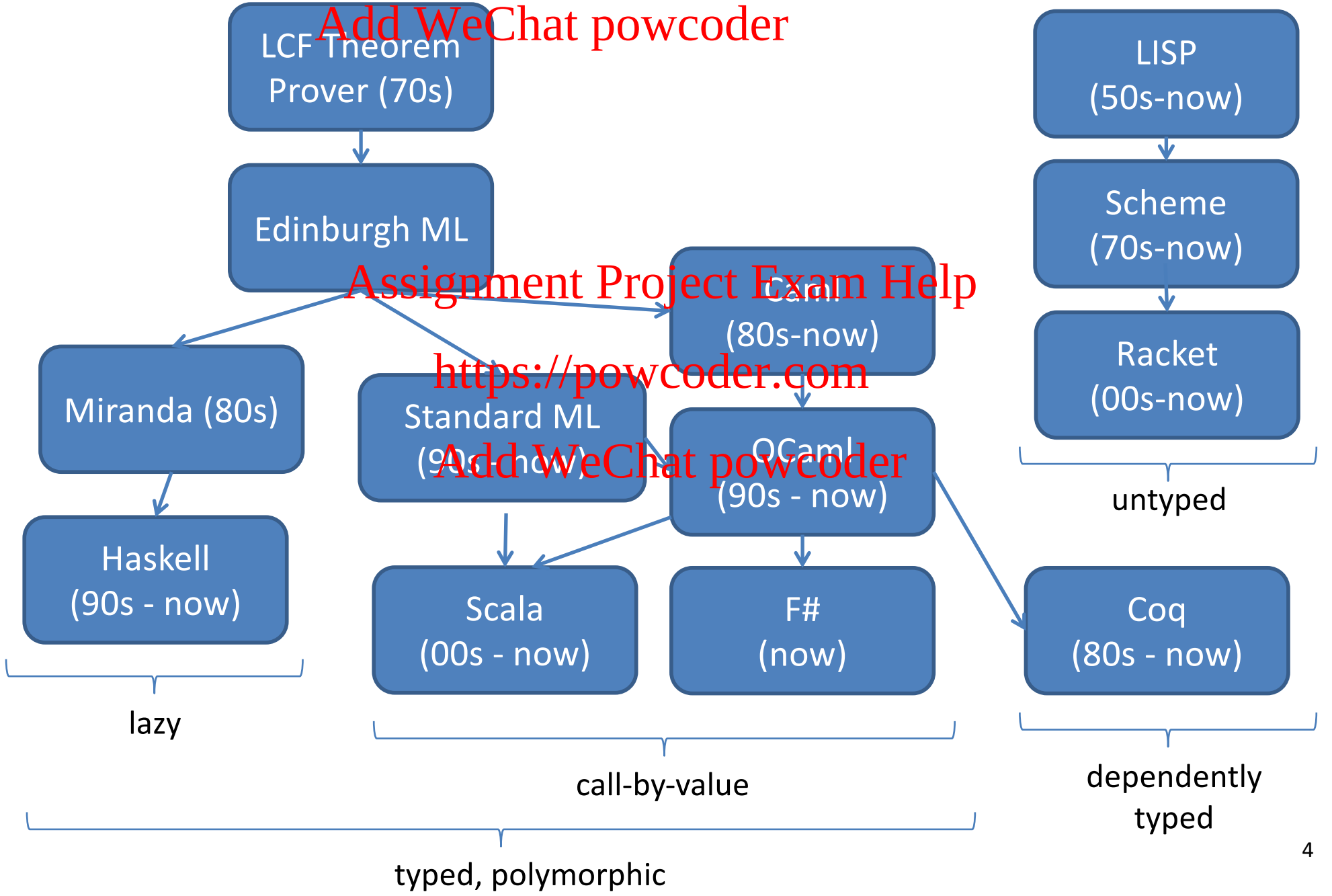
Greatest technological understatement of the 20th century?

He said:

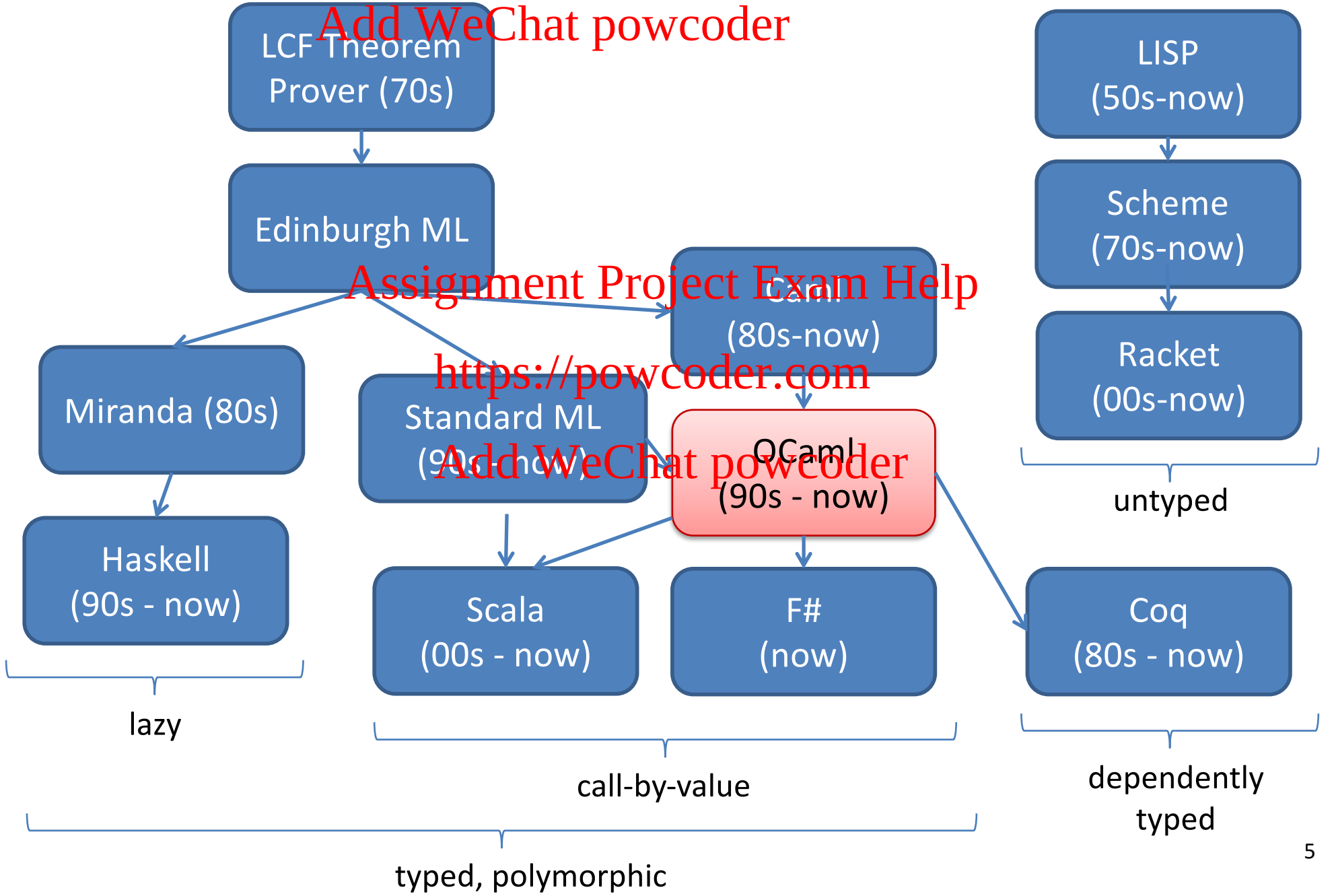
"*There may, indeed, be other applications of the system than its use as a logic.*"

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

Vastly Abbreviated FP Geneology



Vastly Abbreviated FP Geneology



Functional Languages: Who's using them?

map-reduce in their data centers

Scala for
correctness, maintainability, flexibility

Erlang for
concurrency,
Haskell for
managing PHP

F# in Visual Studio

Haskell to
synthesize hardware

Coq (re)proof of
4-color theorem

Haskell
for specifying
equity derivatives

www.artima.com/scalazine/articles/twitter_on_scala.html
www.janestreet.com/technology/
msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/visual-fsharp
ai.google/research/pubs/pub62
www.haskell.org/haskellwiki/Haskell_in_industry
blogs.wsj.com/cio/2014/02/24/facebook-whatsapp-messaging-service-written-in-exotic-erlang/

Functional Languages: Join the crowd

- Elements of functional programming are showing up all over
  - F# in Microsoft Visual Studio
  - Scala combines ML (a functional language) with Objects
    - runs on the JVM
  - C# includes "delegates"
    - delegates == functions
  - Python includes "lambdas"
    - lambdas == more functions
  - Javascript
    - find tutorials online about using functional programming techniques to write more elegant code
  - C++ libraries for map-reduce
    - enabled functional parallelism at Google
  - Java has generics and GC
  - ...

A Functional Review

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)

Thinking Functionally

pure, functional code:

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

imperative code:

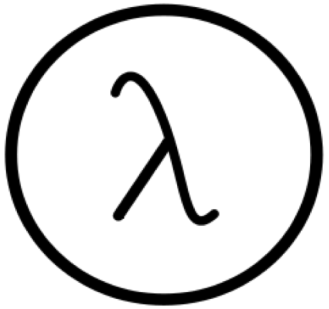
```
temp = pair.x;
pair.x = pair.y;
pair.y = temp;
```


- *outputs are everything!*
- *output is function of input*
- *data properties are stable*
- *repeatable*
- *parallelism apparent*
- *easier to test*
- *easier to compose*

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

OCaml Feature Overview

Small core based on the *lambda calculus*.



- Control is based on (recursive) functions.
- Instead of for-loops, while-loops, do-loops, iterators, etc.
  - can be defined as library functions
- Makes it easy to define semantics

Supports *first-class, lexically-scoped, higher-order* procedures

- a.k.a. first-class functions or closures or lambdas
- first-class: functions are data values like any other data value
  - like numbers, they can be stored, defined anonymously, ...
- lexically-scoped: meaning of variables determined statically
- higher-order: functions as arguments and results
  - programs passed to programs; generated from programs

These features also found in Racket, Haskell, SML, F#, Clojure, ...

OCaml Feature Overview

Statically typed: debugging and testing aid

- compiler catches many silly errors before you can run the code
  - A type is worth a thousand tests (start at 6:20):
    - https://www.youtube.com/watch?v=q1Yi-WM7XqQ
- Java is also strongly, statically typed

- Scheme, Python, Javascript, etc. are all strongly, *dynamically typed* – type errors are discovered while the code is running

Strongly typed: compiler enforces type abstraction

- cannot cast an integer to a record, function, string, etc.
- 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

Running OCaml

- OCaml comes with compilers:
  - "ocamlc" – fast bytecode compiler
  - "ocamlopt" – optimizing, native code compiler
  - "ocamlbuild" – a nice wrapper that computes dependencies (not used in this course)
- And an interactive, top-level shell (interpreter):
  - useful for trying something out, and for debugging small programs
  - "ocaml" at the prompt
  - *You will likely use this most of the time*
- And many other tools
  - e.g., debugger, dependency generator, profiler, etc.
- We will use it via VCL. You may want to try to install it yourself.
  - See OCaml.org

Editing OCaml Programs

- Many options:
  - Emacs
    - what I'll be using in class, and what is available on lab computers
    - good support for OCaml with the Tuareg Emacs Mode
    - powerful text editor, can run OCaml inside it
    - (extensions written in elisp – a functional language!)
  - OCaml IDE
    - integrated development environment written in OCaml
    - I haven't used it, so can't comment.
  - Eclipse
    - I haven't tried it but others recommend it. At least one TA can help with this.
  - Sublime, atom
    - I haven't tried it, but often used by students.

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

AN INTRODUCTORY EXAMPLE (OR TWO)

Ocaml Compiler and Interpreter

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

A First OCaml Program

hello.ml:

```
print_string "Hello CSI 3120!!\n";;
```

A First OCaml Program

hello.ml:

```
print_string "Hello CSI 3120!!\n"
```


a function

its string argument enclosed in `" . . . "`

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

no parens. normally call a function f like this:

```
f arg
```

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

A First OCaml Program

hello.ml:

```
print_string "Hello CSI 3120!!\n"
```


interpreting and playing with hello.ml:


```
$ ocaml
        OCaml Version 4.05.0
# 3 + 1;;
- : int = 4
#
```

the OCaml interpreter

Current versions of Ocaml on website are 4.05 to 4.08 (depends on OS)

A First OCaml Program

hello.ml:

```
print_string "Hello CSI 3120!!\n"
```


interpreting and playing with hello.ml:

```
$ ocaml
        OCaml Version 4.05.0
# 3 + 1;;
- : int = 4
# #use "hello.ml";;
hello CSI 3120!!
- : unit = ()
#
```

the OCaml interpreter

Current versions of Ocaml on website are 4.05 to 4.08 (depends on OS)

A First OCaml Program

hello.ml:

```
print_string "Hello CSI 3120!!\n"
```


interpreting and playing with hello.ml:

```
$ ocaml
        OCaml Version 4.05.0
# 3 + 1;;
- : int = 4
# #use "hello.ml";;
hello CSI 3120!!
- : unit = ()
# #quit;;
$
```

the OCaml interpreter

Current versions of Ocaml on website are 4.05 to 4.08 (depends on OS)

A First OCaml Program

hello.ml:

```
print_string "Hello CSI 3120!!\n"
```


compiling and running hello.ml:


```
$ ocamlc hello.ml
$
```

the OCaml compiler

A First OCaml Program

hello.ml:

```
print_string "Hello CSI 3120!!\n"
```


compiling and running hello.ml:


```
$ ocamlc hello.ml
$ ocaml
        OCaml Version 4.05.0
#
```

the OCaml compiler

A First OCaml Program

hello.ml:

```
print_string "Hello CSI 3120!!\n"
```


compiling and running hello.ml:


```
$ ocamlc hello.ml
$ ocaml
        OCaml Version 4.05.0
# #load "hello.cmo";;
Hello CSI 3120!!
#
```

the OCaml compiler

A Second OCaml Program

a comment
(* ... *)

sumTo.ml:

```
(* sum the numbers from 0 to n
*)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)
let _ =
  print_int (sumTo 8);
  print_newline()
```

A Second OCaml Program

the name of the function being defined

sumTo.ml:

```
(* sum the numbers from 0 to n
*)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    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

sumTo.ml:

```
(* sum the numbers from 0 to n
*)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)
let _ =
  print_int (sumTo 8);
  print_newline()
```


result type int

argument named n with type int

A Second OCaml Program

sumTo.ml:

```
(* sum the numbers from 0 to n
*)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)
let _ =
  print_int (sumTo 8);
  print_newline()
```


a conditional statement in OCaml

A Second OCaml Program

Each branch of the conditional statement constructs a result

sumTo.ml:

```
(* sum the numbers from 0 to n
*)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    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

sumTo.ml:

```
(* sum the numbers from 0 to n
*)
let rec sumTo (n:int) : int =
  if n <= 0 then
    0
  else
    n + sumTo (n-1)
let _ =
  print_int (sumTo 8);
  print_newline()
```


print the result of calling sumTo on 8

print a new line

A Second OCaml Program (New Version)

newSumTo.ml:

Additional 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 (New Version)

deconstruct the value n
using pattern matching

newSumTo.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

newSumTo.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

newSumTo.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()
```


as before, construct the result 0

as before, construct a result using a recursive call to sumTo

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

OCAML BASICS: EXPRESSIONS, VALUES, SIMPLE TYPES

Terminology: Expressions, Values, Types

- Expressions are computations
  - 2 + 3 is a computation

- Values are the results of computations
  - 5 is a value

- Types describe collections of values and the computations that generate those values
  - int is a type

  - values of type int include
    - 0, 1, 2, 3, …, max_int
    - -1, -2, …, min_int

Some simple types, values, expressions

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

For more primitive types and functions over them,
see the OCaml Reference Manual here:

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html

Not every expression has a value

Expression:

```
42 * (13 + 1)          evaluates to  588
(3.14 +. 12.0) *. 10e6      ↦        151400000.
int_of_char 'a'             ↦        97
"moo" ^ "cow"               ↦        "moocow"
if true then 3 else 4       ↦        3
print_int 3                 ↦        ()

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

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

OCAML BASICS:
CORE EXPRESSION SYNTAX

Core Expression Syntax

The simplest OCaml expressions are:

- values — *numbers, strings, bools, ...*
- id — *variables (x, foo, ...)*
- e_1 op e_2 — *operators (x+3, ...)*
- id e_1 e_2 ... e_n — *function call (foo 3 42)*
- **let** id = e_1 **in** e_2 — *local variable decl.*
- **if** e_1 **then** e_2 **else** e_3 — *a conditional*
- (e) — *a parenthesized expression*
- (e : t) — *an expression with its type*

A note on parentheses

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

```
f(x,y,z)        sum(3,4,5)
```

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


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


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


```
f x y z
f x (y z)
```

The first one passes three arguments to f (x, y, and z)

The second passes two arguments to f (x, and the result of applying the function y to z.)