CS 320 : Functional Programming in Ocaml

(based on slides from David Walker, Princeton, Lukasz Ziarek, Buffalo and myself.)

Marco Gaboardi

MSC 116

gaboardi@bu.edu

What is a Functional Language

A functional language:

- defines programs in a way similar to the one we useto define mathematical functions,
- avoids the use of mutable states (states that can change) in describing what a program should do.

In a functional language, the information is maintained by the computation.

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

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)
5 : 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 s ")

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

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

tring_of_int e : string

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

- Using the rules:

2 : int and 3 : int. (By rule 1)
Therefore, (2 + 3) : int (By rule 3)
5 : int (By rule 1)
Therefore, (2 + 3) * 5 : int (By rule 4 and our previous work)

Notation

We will often use lower case english letters, `e,f,g,` etc to denote an arbitrary expression. If we write

`f e`

We mean that we are considering an expression built up as the application of an expression named `f` to another expression named `e`

Notation

When we write

```
f g e
```

We mean that we are considering an expression built up as the application of an expression named `f` to another expression named `g`, and the resulting expression is further applied to an expression named `e`.

This comes from the fact that application associates to the left, so we read the expression above as:

```
((f g) e)
```

TopHat Q1-Q2

Learning Goals for today

- More Type Checking
- Type Safety
- Let-abstraction
- Defining functions
- Function Types

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

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

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.*

TopHat Q3-Q7

Abstraction

What is the single most important mathematical concept ever developed in human history?

What is the single most important mathematical concept ever developed in human history?

An answer: The mathematical variable

What is the single most important mathematical concept ever developed in human history?

An answer: The mathematical variable

(runner up: natural numbers/induction)

Why is the mathematical variable so important?

The mathematician says:

"Let x be some integer, we define a polynomial over x ..."

Why is the mathematical variable so important?

The mathematician says:

"Let x be some integer, we define a polynomial over x ..."

What is going on here? The mathematician has separated a *definition* (of x) from its *use* (in the polynomial).

This is the most primitive kind of *abstraction* (x is *some* integer)

Abstraction is the key to controlling complexity and without it, modern mathematics, science, and computation would not exist.

It allows *reuse* of ideas, theorems ... functions and programs!

Let-abstraction (or let-binding)

Abstraction

- Good programmers identify repeated patterns in their code and factor out the repetition into meaningful components
- In O'Caml, the most basic technique for factoring your code is to use let expressions
- Instead of writing this expression:


```
(2 + 3) * (2 + 3)
```

Abstraction & Abbreviation

- Good programmers identify repeated patterns in their code and factor out the repetition into meaning components
- In O'Caml, the most basic technique for factoring your code is to use let expressions
- Instead of writing this expression:


```
(2 + 3) * (2 + 3)
```


- We write this one:

```
let x = 2 + 3 in
x * x
```

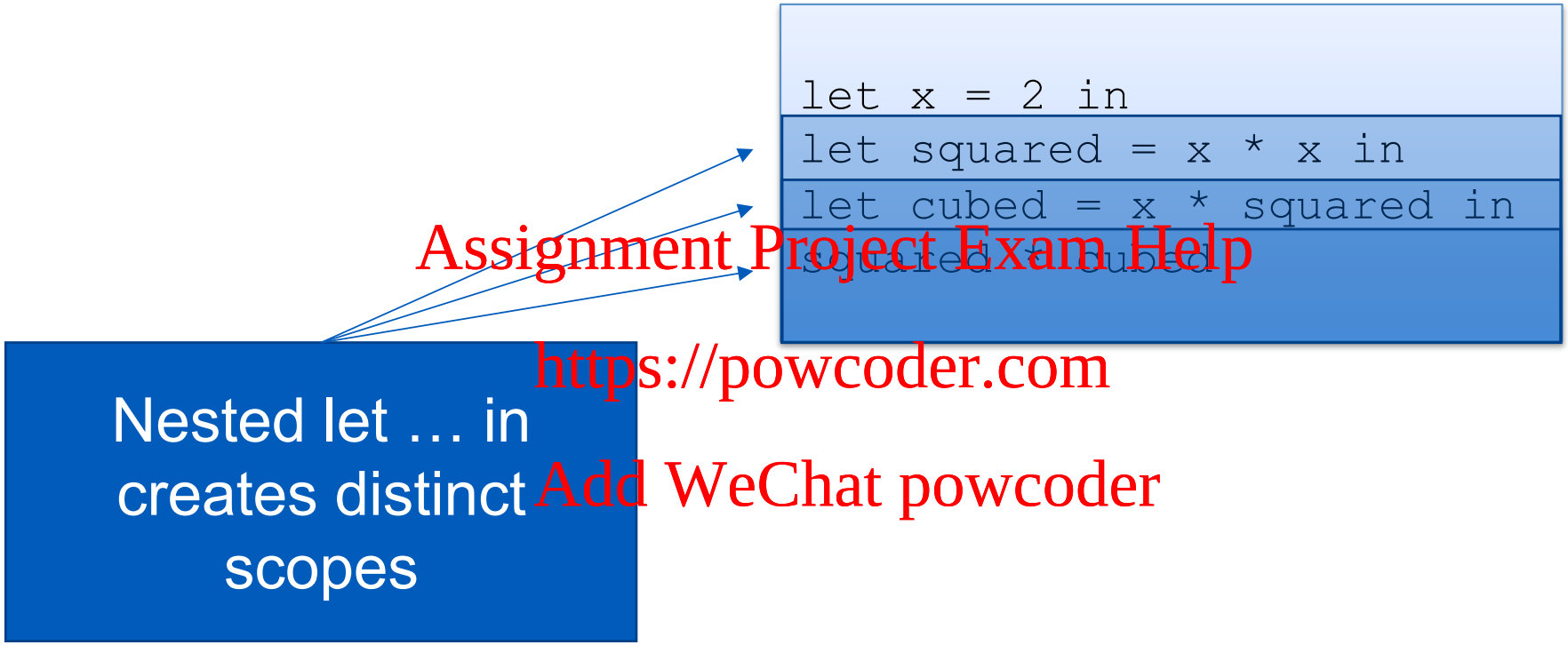
A Few More Let Expressions

```
let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
```



```
let a = "a" in
let b = "b" in
let as = a ^ a ^ a in
let bs = b ^ b ^ b in
as ^ bs
```

A Few More Let Expressions



Abstraction & Abbreviation

- Two kinds of let:

```
if tuesday() then
     let x = 2 + 3 in
     x + x
else
     0
```

```
let x = 2 + 3

let y = x + 17 / x
```


let … **in** … is an *expression* that can appear inside any other *expression*

The scope of x does not extend outside the enclosing "in"

let … without "in" is a top-level *declaration*

Variables x and y may be exported; used by other modules

(Don't need ;; if another let comes next; do need it if the next top-level declaration is an expression)

Binding Variables to Values

- Each OCaml variable is *bound* to 1 value
- *The value to which a variable is bound to never changes*!

```
let x = 3

let add_three (y:int) : int = y + x
```

Binding Variables to Values

- Each OCaml variable is *bound* to 1 value
- *The value to which a variable is bound to never changes*!

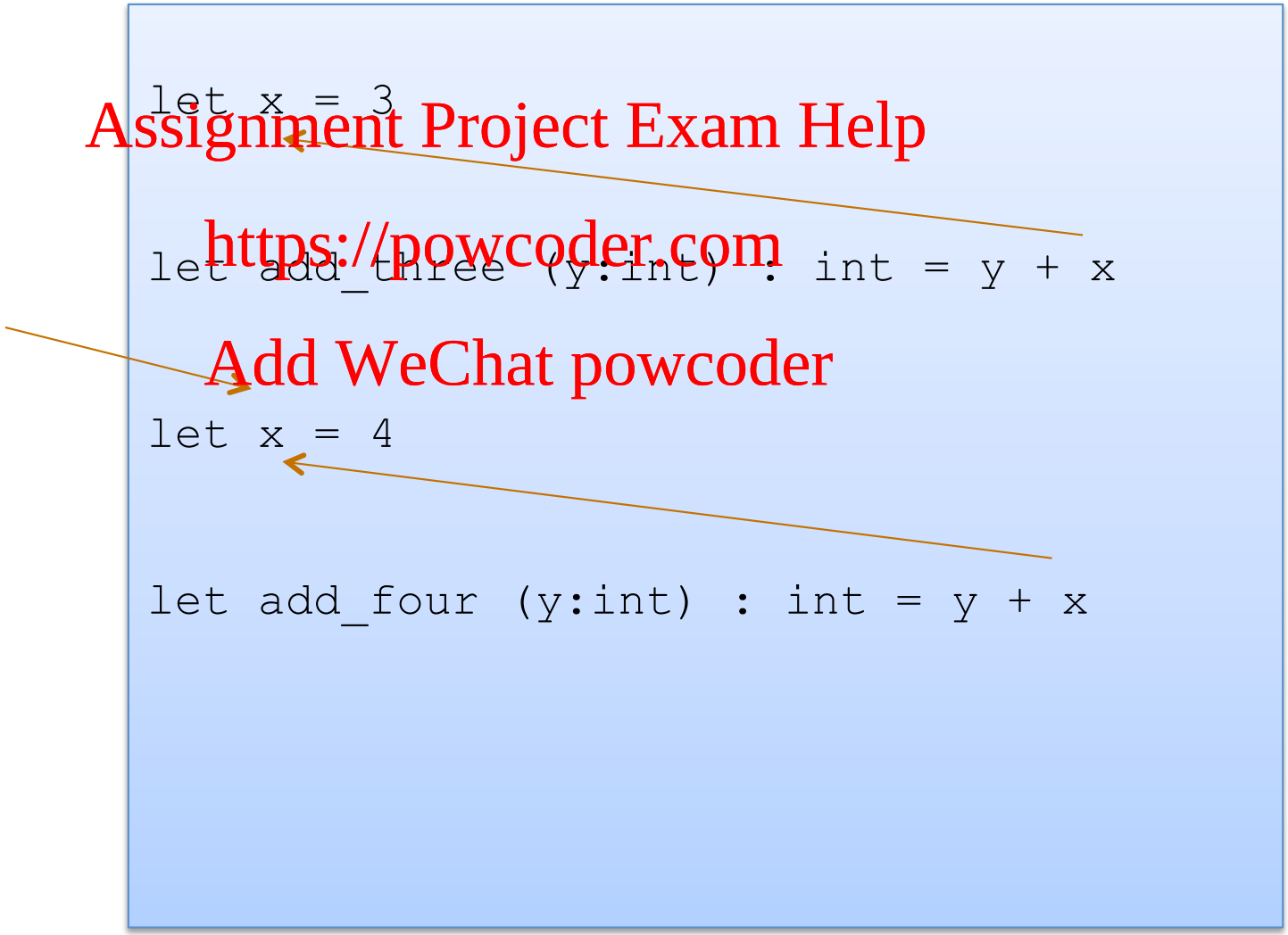
```
let x = 3
let add_three (y:int) : int = y + x
```


It does not matter what I write next. add_three will always add 3!

Binding Variables to Values

- Each OCaml variable is bound to 1 value
- *The value a variable is bound to never changes*!

a distinct variable that "happens to be spelled the same"



Binding Variables to Values

- Since the 2 variables (both happened to be named x) are actually different, unconnected things, we can rename them

rename x
to zzz
if you want
to, replacing
its uses

```
let x = 3

let add_three (y:int) : int = y + x

let zzz = 4

let add_four (y:int) : int = y + zzz

let add_seven (y:int) : int =
  add_three (add_four y)
```

Binding Variables to Values

- Each OCaml variable is bound to 1 value
- OCaml is a statically scoped (or lexically scoped) language

we can use add_three without worrying about the second definition of x

```
let x = 3

let add_three (y:int) : int = y + x

let x = 4

let add_four (y:int) : int = y + x

let add_seven (y:int) : int =
  add_three (add_four y)
```

How do let expressions operate?

```
let x = 2 + 1 in x * x
```

How do let expressions operate?

```
let x = 2 + 1 in x * x
```

-->


```
let x = 3 in x * x
```

How do let expressions operate?

```
let x = 2 + 1 in x * x
```

-->


```
let x = 3 in x * x
```


substitute
3 for x

-->


```
3 * 3
```

How do let expressions operate?

```
let x = 2 + 1 in x * x
```

-->


```
let x = 3 in x * x
```


substitute
3 for x

-->


```
3 * 3
```

-->

```
9
```

How do let expressions operate?

```
let x = 2 + 1 in x * x
```

-->


```
let x = 3 in x * x
```


substitute
3 for x

-->


```
3 * 3
```

-->

```
9
```

Note: I write
e1 --> e2
when e1 evaluates
to e2 in one step

Did you see what I did there?

Did you see what I did there?

I defined the language in terms of itself:

let x = 2 in x + 3 -> 2 + 3

I'm trying to train you to think at a high level of abstraction.

I didn't have to mention low-level abstractions like assembly code or registers or memory layout

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```


-->

```
let y = 4     in
y * 2
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```


substitute
4 for y

-->

```
let y = 4     in
y * 2
```

-->

```
4 * 2
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```


-->

```
let y = 4     in
y * 2
```

substitute
4 for y

-->

```
4 * 2
```

-->

```
8
```

Moral: Let operates by *substituting* computed values for variables

What would happen in an imperative language?

C program:

```
x = 2;
x += x;
return x*2;
```

substitute
2 for x

-->

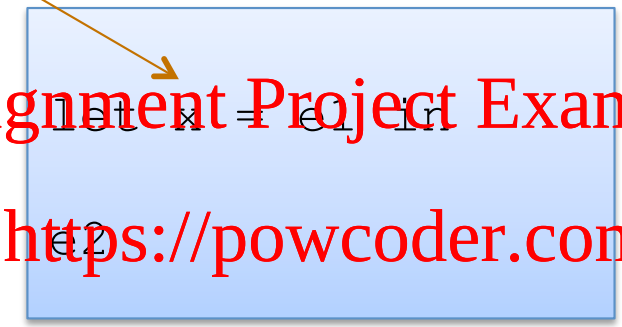
```
x += 2   ???
return x*2;
```

substituting
computed values
for variables

This principle works in
functional languages, not
so well in imperative
languages

Back to Let Expressions ... Typing

x granted type of e1 for use in e2



overall expression takes on the type of e2

Back to Let Expressions ... Typing

x granted type of e1 for use in e2

```
let x = e1 in
e2
```

overall expression takes on the type of e2

x has type int for use inside the let body

```
let x = 3 + 4 in
string_of_int x
```

overall expression has type string

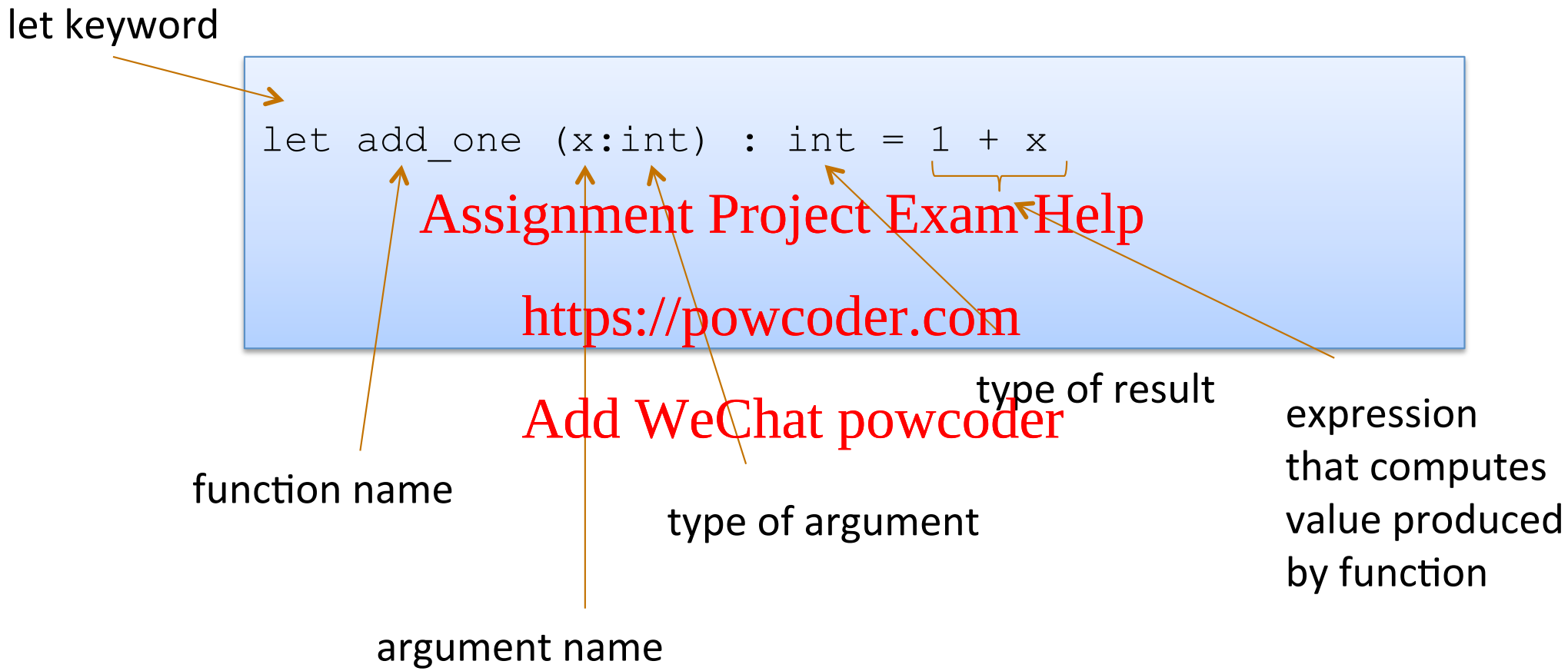
TopHat Q8-Q13

Functions

Defining functions

```
let add_one (x:int) : int = 1 + x
```

Defining functions



Note: recursive functions with begin with "**let rec**"

Defining functions

- Nonrecursive functions:

```
let add_one (x:int) : int = 1 + x

let add_two (x:int) : int = add_one (add_one x)
```


definition of add_one must come before use

Defining functions

- Nonrecursive functions:

```
let add_one (x:int) : int = 1 + x

let add_two (x:int) : int = add_one (add_one x)
```


- With a local definition:

```
let add_two' (x:int) : int =
  let add_one x = 1 + x in
  add_one (add_one x)
```

local function definition
hidden from clients

I left off the types.
O'Caml figures them out

Good style: types on
top-level definitions

Types for Functions

Some functions:

```
let add_one (x:int) : int = 1 + x

let add_two (x:int) : int = add_one (add_one x)

let add (x:int) (y:int) : int = x + y
```


function with two arguments

Types for functions:

```
add_one : int -> int

add_two : int -> int

add : int -> int -> int
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

Example:

```
add_one : int -> int

3 + 4 : int

add_one (3 + 4) : int
```

Rule for type-checking functions

- Recall the type of add:

Definition:

```
let add (x:int) (y:int) : int =
  x + y
```


Type:

```
add : int -> int -> int
```

Rule for type-checking functions

- Recall the type of add:

Definition:

```
let add (x:int) (y:int) : int =
  x + y
```


Type:

```
add : int -> int -> int
```

Same as:

```
add : int -> (int -> int)
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> int -> int

3 + 4 : int

add (3 + 4) : ???
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> (int -> int)

3 + 4 : int

add (3 + 4) :
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> (int -> int)

3 + 4 : int

add (3 + 4) : int -> int
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> int -> int

3 + 4 : int

add (3 + 4) : int -> int

(add (3 + 4)) 7 : int
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> int -> int

3 + 4 : int

add (3 + 4) : int -> int

add (3 + 4) 7 : int
```

Rule for type-checking functions

Example:

```
let munge (b:bool) (x:int) : ?? =
  if not b then
    string_of_int x
  else
    "hello"
;;

let y = 17;;
```



```
munge (y > 17) : ??

munge true (f (munge false 3)) : ??
  f : ??

munge true (g munge) : ??
  g : ??
```

Rule for type-checking functions

Example:

```
let munge (b:bool) (x:int) : ?? =
  if not b then
    string of int x
  else
    "hello"
;;

let y = 17;;
```



```
munge (y > 17) : ??

munge true (f (munge false 3)) : ??
  f : string -> int

munge true (g munge) : ??
  g : (bool -> int -> string) -> int
```

One key thing to remember

- If you have a function f with a type like this:

A -> B -> C -> D -> E -> F

- Then each time you add an argument, you can get the type of the result by knocking off the first type in the series

f a1 : B -> C -> D -> E -> F (if a1 : A)

f a1 a2 : C -> D -> E -> F (if a2 : B)

f a1 a2 a3 : D -> E -> F (if a3 : C)

f a1 a2 a3 a4 a5 : F (if a4 : D and a5 : E)

TopHat Q14-Q19

Summary

- More Type Checking
- Type Safety
- Let-abstraction
- Defining functions
- Function Types