

# **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`  
`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 OCaml 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 : 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`

# 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)

# 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

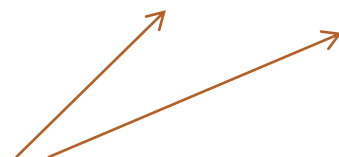
Example rules:

- (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`



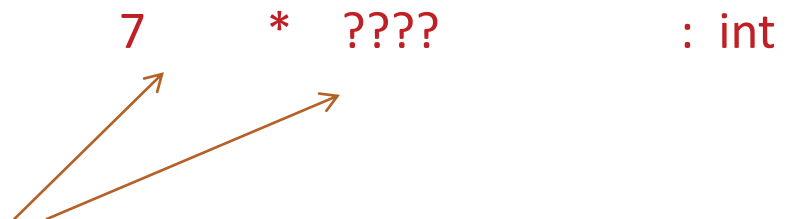
# Type Checking Rules

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 `????`





# Type Checking Rules

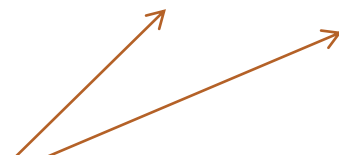
Example rules:

- (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:

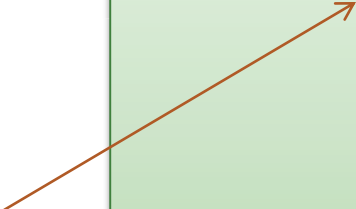
```
$ ocaml
      Objective Caml Version 3.12.0
#
```

# 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

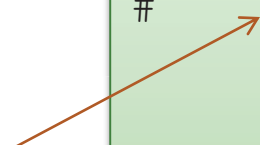


(“;;” can also end a top-level phrase in a file, but I’m going to avoid using it there because then some of you will confuse it with a “;” ....)

# 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
#
```



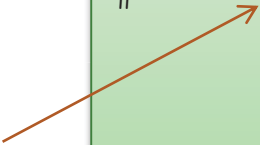
press  
return  
and you  
find out  
the type  
and the  
value

# 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

- More rules:

(7) `true : bool`

(8) `false : bool`

(9) if `e1 : bool`  
and `e2 : t` and `e3 : t` (for some type `t`)  
then `if e1 then e2 else e3 : t`

- Using the rules:

`if ??? then ??? else ??? : int`

# Type Checking Rules

- More rules:

(7) `true : bool`

(8) `false : bool`

(9) if `e1 : bool`  
and `e2 : t` and `e3 : t` (for some type `t`)  
then `if e1 then e2 else e3 : t`

- Using the rules:

`if true then ???? else ???? : int`

# Type Checking Rules

- More rules:

(7) `true : bool`

(8) `false : bool`

(9) if `e1 : bool`  
and `e2 : t` and `e3 : t` (for some type `t`)  
then `if e1 then e2 else e3 : t`

- Using the rules:

`if true then 7 else ???? : int`

# Type Checking Rules

- More rules:

(7) `true : bool`

(8) `false : bool`

(9) if `e1 : bool`  
and `e2 : t` and `e3 : t` (for some type `t`)  
then `if e1 then e2 else e3 : t`

- Using the rules:

`if true then 7 else 8 : int`

# Type Checking Rules

- More rules:

(7) `true : bool`

(8) `false : bool`

(9) if `e1 : bool`  
and `e2 : t` and `e3 : t` (for some type `t`)  
then `if e1 then e2 else e3 : t`

- Violating the rules

`if false then "1" else 2 : ????`



types don't agree -- one is a string and one is an int

# Type Checking Rules

- Violating the rules:

```
# if true then "1" else 2;;
```

```
Error: This expression has type int but an  
expression was expected of type string
```

```
#
```

# 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*

# **TYPE ERRORS**

# Type Checking Rules

Type errors for if statements can be confusing sometimes. Recall:

```
let rec concatn s n =  
  if n <= 0 then  
    ...  
  else  
    s ^ (concatn s (n-1))
```



# Type Checking Rules

Type errors for if statements can be confusing sometimes. Recall:

```
let rec concatn s n =  
  if n <= 0 then  
    ...  
  else  
    s ^ (concatn s (n-1))
```

ocamlbuild says:

**Error: This expression has type int but an expression was expected of type string**

# Type Checking Rules

Type errors for if statements can be confusing sometimes. Recall:

```
let rec concatn s n =  
  if n <= 0 then  
    ...  
  else  
    s ^ (concatn s (n-1))
```

ocamlbuild says:

**Error: This expression has type int but an expression was expected of type string**

merlin inside emacs points to the error above and gives a second error:


**Error: This expression has type string but an expression was expected of type int**

# Type Checking Rules

Type errors for if statements can be confusing sometimes.

Example. We create a string from  $s$ , concatenating it  $n$  times:

```
let rec concatn s n =  
  if n <= 0 then  
    ...  
  else  
    s ^ (concatn s (n-1))
```



ocamlbuild says:

**Error: This expression has type int but an expression was expected of type string**

merlin inside emacs points to the error above and gives a second error:

**Error: This expression has type string but an expression was expected of type int**

# Type Checking Rules

Type errors for if statements can be confusing sometimes.

Example. We create a string from *s*, concatenating it *n* times:

they don't  
*agree!*

```
let rec concatn s n =  
  if n <= 0 then  
    0  
  else  
    s ^ (concatn s (n-1))
```

???

ocamlbuild says:

**Error: This expression has type int but an expression was expected of type string**

merlin inside emacs points to the error above and gives a second error:

**Error: This expression has type string but an expression was expected of type int**

# Type Checking Rules

Type errors for if statements can be confusing sometimes.

Example. We create a string from  $s$ , concatenating it  $n$  times:

they don't  
agree!

```
let rec concatn s n =  
  if n <= 0 then  
    0  
  else  
    s ^ (concatn s (n-1))
```


???

The type checker points to *some* place where there is *disagreement*.

Moral: *Sometimes you need to look in an earlier branch for the error*  
even though the type checker points to a later branch.  
The type checker doesn't know what the user wants.

## A Tactic: Add Typing Annotations

```
let rec concatn (s:string) (n:int) : string =  
  if n <= 0 then  
    0  
  else  
    s ^ (concatn s (n-1))
```



**Error: This expression has type int but an expression was expected of type string**

**ONWARD**

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!

# **OCAML BASICS: LET DECLARATIONS**

# Abstraction

- Good programmers identify repeated patterns in their code and factor out the repetition into meaningful components
- In OCaml, 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 OCaml, 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
```



## 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
```

# Abstraction & Abbreviation

Two “kinds” of let:

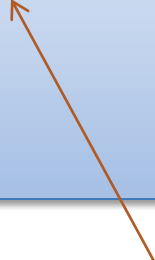
```
if tuesday() then
  let x = 2 + 3 in
  x + x
else
  0
```



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

The scope of `x` (ie: the places `x` may be used) does not extend outside the enclosing “in”

```
let x = 2 + 3
let y = x + 17 / x
```



`let ...` without “in” is a top-level *declaration*

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

You can only omit the “in” in a top-level declaration

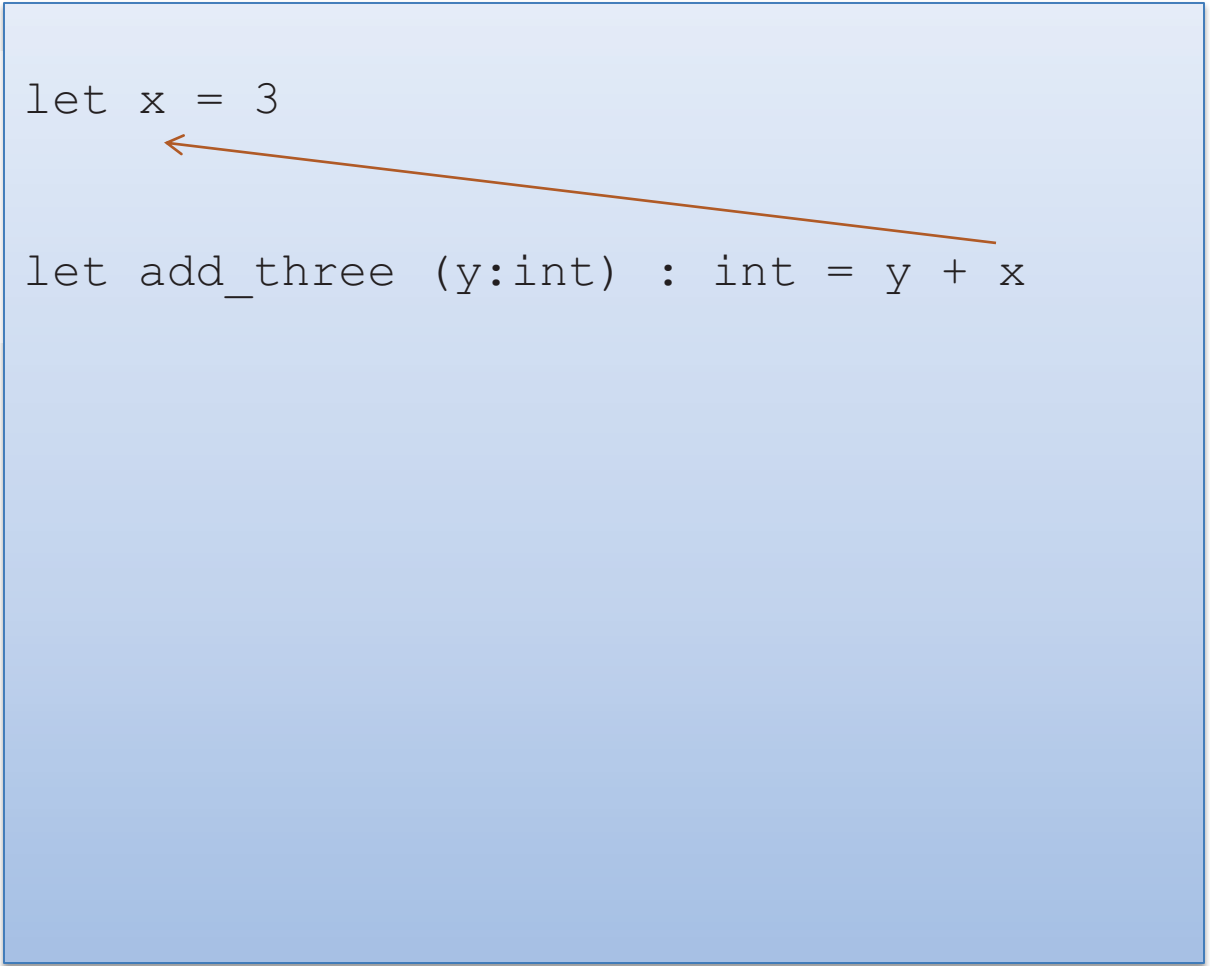
# Binding Variables to Values

During execution, we say an OCaml variable is *bound* to a value.

*The value to which a variable is bound to never changes!*

```
let x = 3
```

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



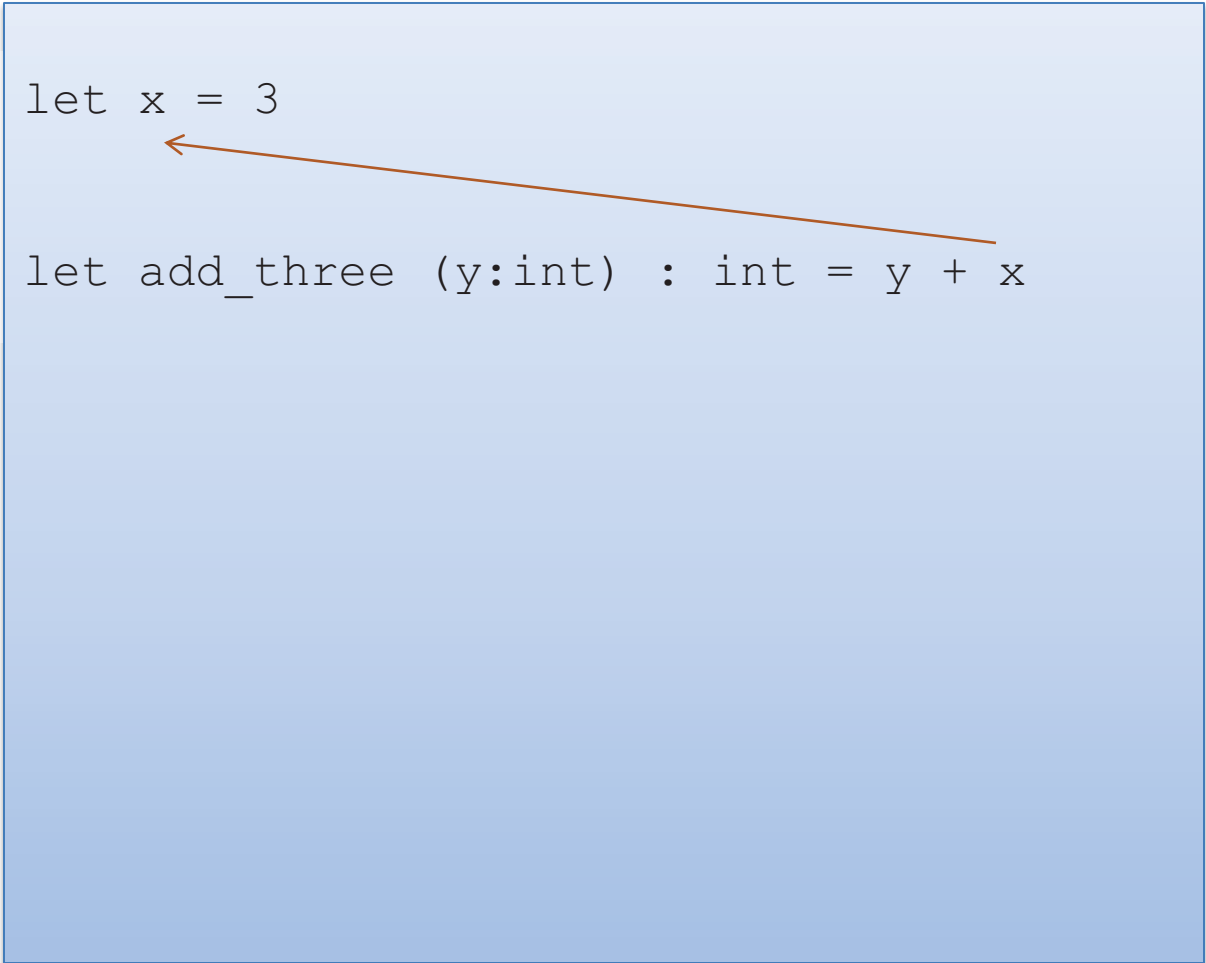
# Binding Variables to Values

During execution, we say an OCaml variable is *bound* to a 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

During execution, we say an OCaml variable is *bound* to a value.

*The value to which a variable is bound to never changes!*

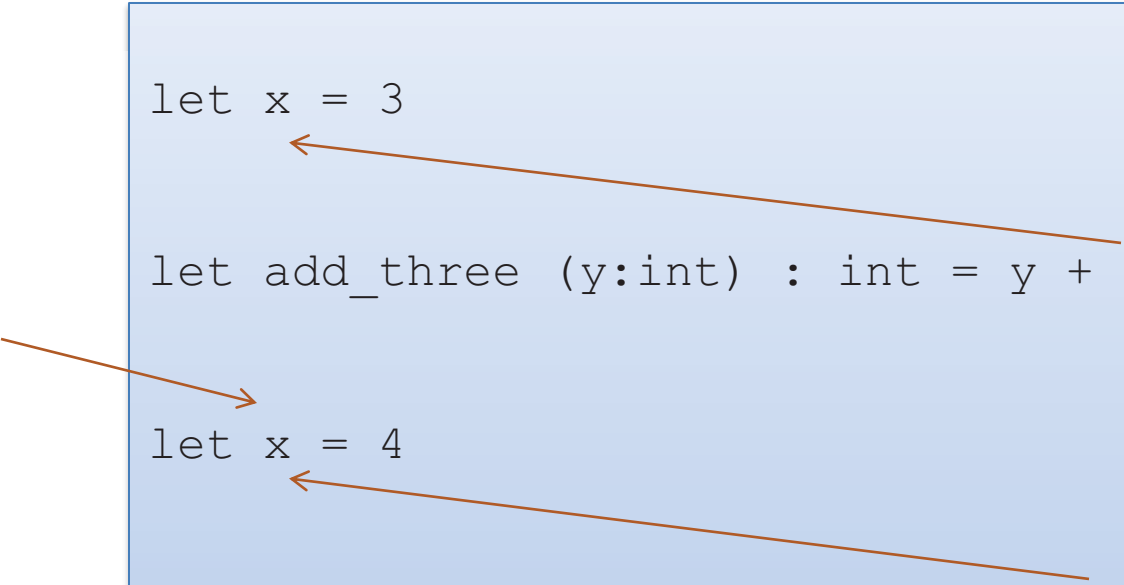
a distinct  
variable that  
"happens to  
be spelled the  
same"

```
let x = 3
```

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

```
let x = 4
```

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



# 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
```

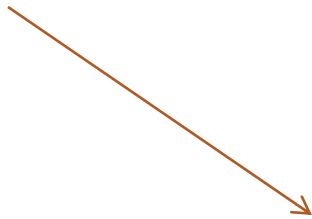
```
let add_seven (y:int) : int =  
  add_three (add_four y)
```

# Binding Variables to Values

A use of a variable always refers to its *closest* (in terms of syntactic distance) enclosing declaration. Hence, we say OCaml is a *statically scoped* (or *lexically scoped*) language

```
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)
```

we can use  
add\_three  
without worrying  
about the second  
definition of x



# How does OCaml execute a let expression?

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



# How does OCaml execute a let expression?

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

-->

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

# How does OCaml execute a let expression?

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

-->

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

-->

```
3 * 3
```

substitute  
3 for x



# How does OCaml execute a let expression?

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

-->

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

-->

```
3 * 3
```

-->

```
9
```

substitute  
3 for x



# How does OCaml execute a let expression?

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

-->

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

-->

```
3 * 3
```

-->

```
9
```

substitute  
3 for x

**Note:** I write  
 $e1 \rightarrow e2$   
when  $e1$  evaluates  
to  $e2$  in one step

# Meta-comment

OCaml expression

OCaml expression

let x = 2 in x + 3    -->    2 + 3

I defined the language in terms of itself:  
By reduction of one OCaml expression to another

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 to tell you  
how OCaml works.*

## 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
```

-->

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

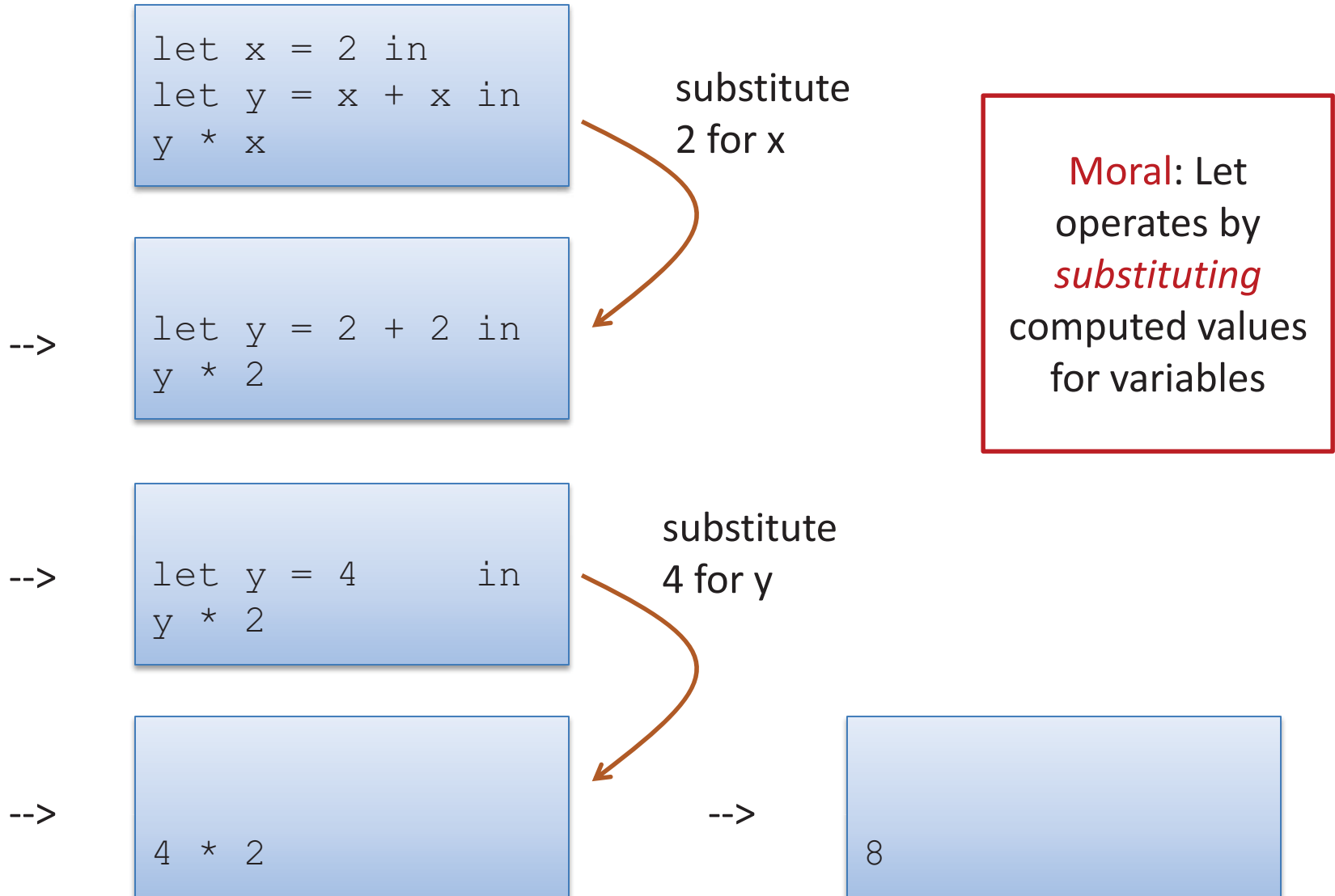
substitute  
4 for y



-->

```
4 * 2
```

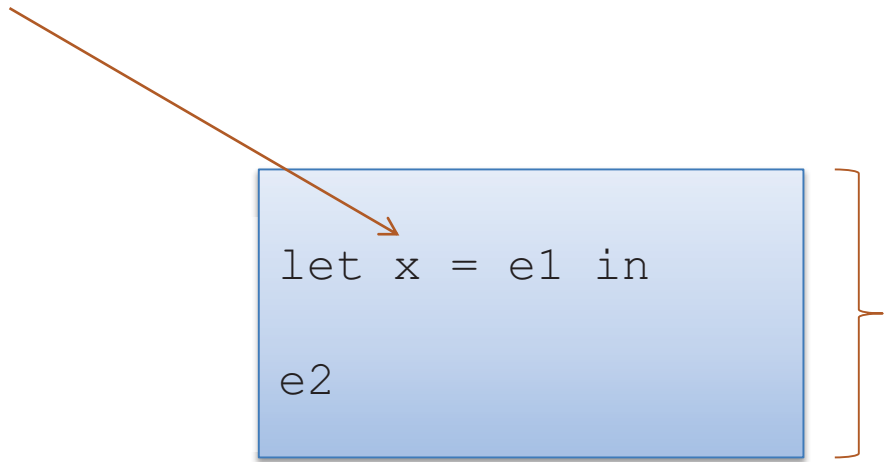
# Another Example



# **OCAML BASICS: TYPE CHECKING AGAIN**

## Back to Let Expressions ... Typing

x granted type of e1 for use in e2

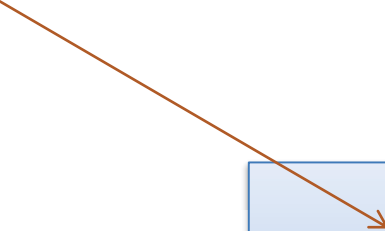


```
let x = e1 in  
e2
```

overall expression  
takes on the type of e2

# 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

# **OCAML BASICS: FUNCTIONS**

# Defining functions

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

# Defining functions

let keyword

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

function name

argument name

type of argument

type of result

expression  
that computes  
value produced  
by function

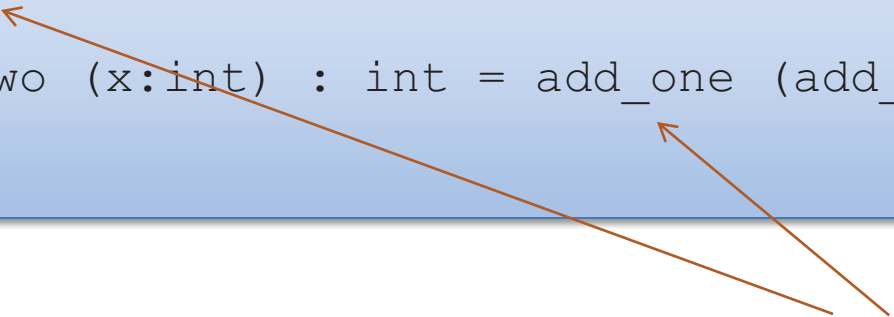
Note: recursive functions begin with **let rec**



# 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.  
OCaml 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  $f : T1 \rightarrow T2$   
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  $f : T1 \rightarrow T2$   
and an argument  $e : T1$   
then  $f e : T2$

$A \rightarrow B \rightarrow C$

same as:

$A \rightarrow (B \rightarrow C)$

Example:

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

```
3 + 4 : int
```

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

# Rule for type-checking functions

General Rule:

If a function  $f : T1 \rightarrow T2$   
and an argument  $e : T1$   
then  $f e : T2$

$A \rightarrow B \rightarrow C$

same as:

$A \rightarrow (B \rightarrow C)$

Example:

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

```
3 + 4 : int
```

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



# Rule for type-checking functions

General Rule:

If a function  $f : T1 \rightarrow T2$   
and an argument  $e : T1$   
then  $f e : T2$

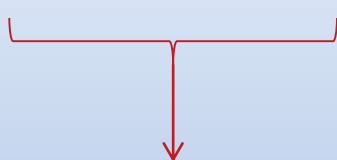
$A \rightarrow B \rightarrow C$

same as:

$A \rightarrow (B \rightarrow C)$

Example:

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



# Rule for type-checking functions

General Rule:

If a function  $f : T1 \rightarrow T2$   
and an argument  $e : T1$   
then  $f e : T2$

$A \rightarrow B \rightarrow C$

same as:


$A \rightarrow (B \rightarrow C)$

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  $f : T1 \rightarrow T2$   
and an argument  $e : T1$   
then  $f e : T2$

$A \rightarrow B \rightarrow C$

same as:

$A \rightarrow (B \rightarrow C)$

Example:


`add : int -> int -> int`

`3 + 4 : int`

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

`add (3 + 4) 7 : int`

extra parens  
not necessary



# 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:

$$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F$$

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

$$f\ a1 : B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \quad (\text{if } a1 : A)$$

$$f\ a1\ a2 : C \rightarrow D \rightarrow E \rightarrow F \quad (\text{if } a2 : B)$$

$$f\ a1\ a2\ a3 : D \rightarrow E \rightarrow F \quad (\text{if } a3 : C)$$

$$f\ a1\ a2\ a3\ a4\ a5 : F \quad (\text{if } a4 : D \text{ and } a5 : E)$$

# Reading Assignments

- [Lecture Notes 01: OCaml Programming Basics](#)
- [Lecture Notes 02: Type Checking](#)
- Optional: Book “[Real World OCaml](#)”
  - Chapter 1
    - Section: [OCaml as a Calculator](#)
    - Section: [Functions and Type Inference](#)