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 \times (13 + 1)
The expression \( 42 \times (13 + 1) \) evaluates to the value 588.

Read like this: “the expression 42 * (13 + 1) evaluates to the value 588.”

The “\(\rightarrow\)” is there to say that it does so in 0 or more small steps.
Evaluation

42 * (13 + 1) --> 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

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

(3.14 + 12.0) * 10^6  -->*  151400000.

int_of_char 'a'  -->*  97

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

if true then 3 else 4  -->*  3

print_int 3  -->*  ()
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
The simplest OCaml expressions $e$ are:

- **values**
  - numbers, strings, bools, ...
- **id**
  - variables (x, foo, ...)
- $e_1 \text{ op } e_2$
  - operators (x+3, ...)
- **id $e_1 e_2 \ldots e_n$**
  - 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:

\[
\begin{align*}
    f(x, y, z) & \quad \text{sum}(3, 4, 5) \\
\end{align*}
\]

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

\[
\begin{align*}
    f \ x \ y \ z & \quad \text{sum} \ 3 \ 4 \ 5 \\
\end{align*}
\]

But we do have to worry about \textit{grouping}. For example,

\[
\begin{align*}
    f \ x \ y \ z & \\
    f \ x \ (y \ z) \\
\end{align*}
\]

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:

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

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

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

Using the rules:

\(2 : \text{int}\) and \(3 : \text{int}\). (By rule 1)
Example rules:

(1) \(0 : \text{int}\) \hspace{1cm} (and similarly for any other integer constant \(n\))

(2) \"abc\" : \text{string}\) \hspace{1cm} (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_of_int} e : \text{string}

Using the rules:

\(2 : \text{int}\) and \(3 : \text{int}\). \hspace{1cm} (By rule 1)
Therefore, \((2 + 3) : \text{int}\) \hspace{1cm} (By rule 3)
Type Checking Rules

Example rules:

(1) \(0 : \text{int}\) \hspace{1cm} (and similarly for any other integer constant \(n\))

(2) \"abc\" : \text{string} \hspace{1cm} (and similarly for any other string constant \"\ldots\")

(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 \(\text{string\_of\_int } e : \text{string}\)

Using the rules:

\[2 : \text{int}\] and \[3 : \text{int}\]. \hspace{1cm} (By rule 1)
Therefore, \[(2 + 3) : \text{int}\] \hspace{1cm} (By rule 3)
\[5 : \text{int}\] \hspace{1cm} (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 \(e_1 + e_2 : \text{int}\)

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

Using the rules:

\[2 : \text{int} \text{ and } 3 : \text{int}.\] (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  (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) \( 0 : \text{int} \) (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 \( e_1 * e_2 : \text{int} \)

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

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

Another perspective:

\[ 7 \ast \_\_\_\_ : \text{int} \]

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 \(e_1 \, ^\wedge \, e_2 : \text{string}\)

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

Another perspective:

\[7 \, \times \, (\text{add}_\text{one} \, 17) : \text{int}\]

rule (4) for typing expressions says I can put any expression with type \text{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
#
```
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
$ 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) \(0 : \text{int}\) (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 \(e_1 \times e_2 : \text{int}\)

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

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

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 Checking Rules

Violating the rules:

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

A possible fix:

```ml
# "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 \( e_1 : \text{bool} \)
    and \( e_2 : t \) and \( e_3 : t \) (for some type \( t \))
    then if \( e_1 \) then \( e_2 \) else \( e_3 \) : \( t \)

• Using the rules:

if true then question else question : int
Type Checking Rules

- More rules:

  (7) \( \text{true} : \text{bool} \)

  (8) \( \text{false} : \text{bool} \)

  (9) \( \text{if } e_1 : \text{bool} \)  
      \( \text{and } e_2 : t \)  
      \( \text{and } e_3 : t \)  
      \( \text{(for some type } t \)  
      \( \text{then } \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \)

- Using the rules:

  \( \text{if true then 7 else } ???? : \text{int} \)
Type Checking Rules

- More rules:

  (7) \[ \text{true} : \text{bool} \]

  (8) \[ \text{false} : \text{bool} \]

  (9) \[ \text{if } \text{e1} : \text{bool} \]
      \[ \text{and } \text{e2} : \text{t} \text{ and } \text{e3} : \text{t} \text{ (for some type } \text{t}) \]
      \[ \text{then } \text{if } \text{e1} \text{ then } \text{e2} \text{ else } \text{e3} : \text{t} \]

- Using the rules:

  \[ \text{if true then 7 else 8} : \text{int} \]
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:

```ocaml
# 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:

```ml
# 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?)
“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, 1978

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

```ocaml
let rec concatn s n =  
  if n <= 0 then  
    ...  
  else  
    s ^ (concatn s (n-1))
```
Type errors for if statements can be confusing sometimes. Recall:

```ocaml
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:

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

ocamllbuild 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 errors for if statements can be confusing sometimes. Example. We create a string from \( s \), concatenating it \( n \) times:

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

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

`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 errors for if statements can be confusing sometimes. Example. We create a string from s, concatenating it n times:

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

they don't agree!

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 errors for if statements can be confusing sometimes. Example. We create a string from s, concatenating it n times:

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

they don't agree!

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.
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 O’Caml, the most basic technique for factoring your code is to use let expressions.
- Instead of writing this expression:

\[(2 + 3) \times (2 + 3)\]
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:
  
  $$(2 + 3) \times (2 + 3)$$
  
- We write this one:
  
  ```ocaml
  let x = 2 + 3 in
  x * x
  ```
let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
A Few More Let Expressions

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

let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
Abstraction & Abbreviation

Two “kinds” of let:

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

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

```ocaml
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!*

```ocaml
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!*
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

let x = 4
let add_four (y:int) : int = y + x
```
Since the 2 variables (both happened to be named x) are actually different, unconnected things, we can rename them

```ocaml
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)
```

rename x to zzz if you want to, replacing its uses
A use of a variable always refers to it’s *closest* (in terms of syntactic distance) enclosing declaration. Hence, we say OCaml is a *statically scoped* (or *lexically scoped*) language.

```ocaml
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 does OCaml execute a let expression?

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

27
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

Note: I write $e_1 \rightarrow e_2$ when $e_1$ evaluates to $e_2$ in one step. Substitute 3 for $x$. 

substitute 3 for $x$
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

34
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

-->
let y = 2 + 2 in
y * 2

-->
let y = 4     in
y * 2

-->
4 * 2
```

substitute 2 for x

substitute 4 for y

36
Another Example

\[
\text{let } x = 2 \text{ in } \\
\text{let } y = x + x \text{ in } y * x
\]

substitute 2 for \( x \)

\[
\text{let } y = 2 + 2 \text{ in } \\
y * 2
\]

.. substitute 4 for \( y \)

\[
\text{let } y = 4 \text{ in } \\
y * 2
\]

Moral: Let operates by \textit{substituting} computed values for variables

\[
4 * 2
\]

\[
8
\]
OCAML BASICS:
TYPE CHECKING AGAIN
Back to Let Expressions ... Typing

- x granted type of e1 for use in e2
- overall expression takes on the type of e2

```
let x = e1 in
e2
```
x granted type of e1 for use in e2

let x = e1 in

overall expression takes on the type of e2

\[
\text{let } x = 3 + 4 \text{ in }
\]

overall expression has type string

\[
\text{string_of_int } x
\]

x has type int for use inside the let body
OCAML BASICS:
FUNCTIONS
let add_one (x:int) : int = 1 + x
Defining functions

let keyword

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

- function name: `add_one`
- type of argument: `int`
- argument name: `x`
- type of result: `int`
- expression that computes value produced by function: `1 + x`

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

Nonrecursive functions:

```ocaml
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:

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

With a local definition:

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

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

Good style: types on top-level definitions.
Some functions:

```ocaml
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
```

Types for functions:

```ocaml
add_one : int -> int
add_two : int -> int
add : int -> int -> int
```
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \rightarrow T_2 \)
and an argument \( e : T_1 \)
then \( f e : T_2 \)

Example:

\[
\text{add_one} : \text{int} \rightarrow \text{int} \\
3 + 4 : \text{int} \\
\text{add_one} (3 + 4) : \text{int}
\]
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:

\[
\text{let } add \ (x:\text{int}) \ (y:\text{int}) : \text{int} = x + y
\]

Type:

\[
\text{add : int } \rightarrow \text{ int } \rightarrow \text{ int}
\]

Same as:

\[
\text{add : int } \rightarrow \ (\text{int } \rightarrow \text{ int})
\]
Rule for type-checking functions

General Rule:

If a function $f : T_1 \rightarrow T_2$ and an argument $e : T_1$ then $f\ e : T_2$

Example:

$add : int \rightarrow int \rightarrow int$

$3 + 4 : int$

$add\ (3 + 4) : ???$
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \to T_2 \) and an argument \( e : T_1 \) then \( f \ e : T_2 \)

Example:

\[
\begin{align*}
\text{add : int} & \to (\text{int} \to \text{int}) \\
3 + 4 & : \text{int} \\
\text{add (3 + 4)} & : 
\end{align*}
\]
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f \ e : T_2 \)

Example:

\[ \text{add} : \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \]
\[ 3 + 4 : \text{int} \]
\[ \text{add} (3 + 4) : \text{int} \rightarrow \text{int} \]
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f \, e : T_2 \)

Example:

\[
\begin{align*}
\text{add} & : \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
3 + 4 & : \text{int} \\
\text{add} \,(3 + 4) & : \text{int} \rightarrow \text{int} \\
(\text{add} \,(3 + 4))\,7 & : \text{int}
\end{align*}
\]
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f \; e : T_2 \)

Example:

\[
\begin{align*}
add & : \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
3 + 4 & : \text{int} \\
add \; (3 + 4) & : \text{int} \rightarrow \text{int} \\
add \; (3 + 4) \; 7 & : \text{int}
\end{align*}
\]

extra parens not necessary
Rule for type-checking functions

Example:

```ocaml
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 : ??
```
Example:

```ocaml
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

\[
\begin{align*}
f\ a1 : & \ B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \quad \text{(if } a1 : A \text{)} \\
f\ a1\ a2 : & \ C \rightarrow D \rightarrow E \rightarrow F \quad \text{(if } a2 : B \text{)} \\
f\ a1\ a2\ a3 : & \ D \rightarrow E \rightarrow F \quad \text{(if } a3 : C \text{)} \\
f\ a1\ a2\ a3\ a4\ a5 : & \ F \quad \text{(if } a4 : D \text{ and } a5 : E \text{)}
\end{align*}
\]
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