Beginning OCaml I: Expressions

CAS CS 320: Principles of Programming Languages

January 23, 2024

Outline

Administrivia

- Homework 0 is due on *Thursday by 11:59PM*. Remember that this assignment is not graded.
- The office hours calendar has been updated with locations.
- Discussions will be held as office hours for helping you set up your personal machines.

Goals for Today

Look at a couple ways of interacting with OCaml, in particular how to organize and run OCaml code (for this course).

Look more carefully at expressions in OCaml, in particular if-expressions and let-expressions. We will begin to see how to conceptualize these things in terms of *syntax* and *semantics*.

Keywords

- Toplevel, UTop, directives
- values, expressions, evaluation, types
- primitive types and values
- syntax, dynamic and static semantics
- if-expressions, let-expressions

Practice Problem

None this week. We haven't really covered anything yet.

Preamble

A lot of people have a lot of bad things to say about OCaml...

OCaml can be a lot of fun, but it can be difficult at first. It's a different game than we're used to.

- It's a game of minimality: *how can we do this in as simple a language as possible?*
- Its a game in the functional paradigm: *"how do we describe what the output of this function?" as opposed to "what is the process for getting this value?"*

Note. You're gonna look stuff up and see familiar things like while and for loops. **Don't blindly use these things**, they don't work the same as in other languages and may cause more confusion in the long run.

Preamble

A lot of people have a lot of bad things to say about OCaml...

OCaml can be a lot of fun, but it can be difficult at first. It's a different game than we're used to.

- It's a game of minimality: *how can we do this in as simple a language as possible?*
- Its a game in the functional paradigm: *"how do we describe what the output of this function?" as opposed to "what is the process for getting this value?"*

Note. You're gonna look stuff up and see familiar things like while and for loops. **Don't blindly use these things**, they don't work the same as in other languages and may cause more confusion in the long run.

Preamble

A lot of people have a lot of bad things to say about OCaml...

OCaml can be a lot of fun, but it can be difficult at first. It's a different game than we're used to.

- It's a game of minimality: *how can we do this in as simple a language as possible?*
- Its a game in the functional paradigm: *"how do we describe what the output of this function?" as opposed to "what is the process for getting this value?"*

Note. You're gonna look stuff up and see familiar things like while and for loops. **Don't blindly use these things**, they don't work the same as in other languages and may cause more confusion in the long run.

Preamble (Continued)

That said, **this is not an introductory programming course.**

We're going to leave a lot of the learning syntax to you. We won't dwell on how comments work, or what a floating-point value is, these are things you have to pick up along the way.

Okay let's get started...

Preamble (Continued)

That said, **this is not an introductory programming course.**

We're going to leave a lot of the learning syntax to you. We won't dwell on how comments work, or what a floating-point value is, these are things you have to pick up along the way.

Okay let's get started...

Preamble (Continued)

That said, **this is not an introductory programming course.**

We're going to leave a lot of the learning syntax to you. We won't dwell on how comments work, or what a floating-point value is, these are things you have to pick up along the way.

Okay let's get started. . .

Outline

Let-Definitions

Every line of OCaml (for now) is a let-definition:

```
let x = 3
let y = "string"
(* function definition *)
let square x = x * x
(* recursive function definition *)
let rec f x = if x = 0 then 0 else x + f (x - 1)
(* We can't just print, we assign to wildcard *)
let _ = print_endline("Hello world")
```

Let-definitions assign a name to a value which is the result of evaluating and expression (even if the value is a function, more on this later).

Note. The `let` keyword (and the way comments work) make OCaml newline agnostic.

Let-Definitions

Every line of OCaml (for now) is a let-definition:

```
let x = 3
let y = "string"
(* function definition *)
let square x = x * x
(* recursive function definition *)
let rec f x = if x = 0 then 0 else x + f (x - 1)
(* We can't just print, we assign to wildcard *)
let _ = print_endline("Hello world")
```

Let-definitions assign a name to a value which is the result of evaluating and expression (even if the value is a function, more on this later).

Note. The `let` keyword (and the way comments work) make OCaml newline agnostic.

Let-Definitions

Every line of OCaml (for now) is a let-definition:

```
let x = 3
let y = "string"
(* function definition *)
let square x = x * x
(* recursive function definition *)
let rec f x = if x = 0 then 0 else x + f (x - 1)
(* We can't just print, we assign to wildcard *)
let _ = print_endline("Hello world")
```

Let-definitions assign a name to a value which is the result of evaluating and expression (even if the value is a function, more on this later).

Note. The `let` keyword (and the way comments work) make OCaml newline agnostic.

State

Unlike almost every programming language you've likely used so far, there is no state. In Python we can write:

```
x = 10
def f(y):
    x = 12 # the value of x is reassigned
    return y
assert(x == 10)
assert(f(10) == 10)
assert(x == 12)
```

In OCaml, we can't *reassign* variables:

```
let x = 10
let f y =
  let x = 12 in (* a local variable is defined *)
  y
let _ = assert (x = 10)
let _ = assert (f 10 = 10)
let _ = assert (x = 10)
```

State

Unlike almost every programming language you've likely used so far, there is no state. In Python we can write:

```
x = 10
def f(y):
    x = 12 # the value of x is reassigned
    return y
assert(x == 10)
assert(f(10) == 10)
assert(x == 12)
```

In OCaml, we can't *reassign* variables:

```
let x = 10
let f y =
  let x = 12 in (* a local variable is defined *)
  y
let _ = assert (x = 10)
let _ = assert (f 10 = 10)
let _ = assert (x = 10)
```

State (Continued)

This matters a lot more with loops. In Python:

```
def fact(n):
    assert(n >= 0)
    out = 1
    for i in range(1, n): # i is "stateful"
        out *= i
    return out
```

In OCaml:

```
let rec fact n =
  let _ = assert (n >= 0) in
  if n = 0
  then 1
  else n * fact (n - 1)
```

State (Continued)

This matters a lot more with loops. In Python:

```python
def fact(n):
    assert(n >= 0)
    out = 1
    for i in range(1, n): # i is "stateful"
        out *= i
    return out
```

In OCaml:

```ocaml
let rec fact n =
  let _ = assert (n >= 0) in
  if n = 0
  then 1
  else n * fact (n - 1)
```

Recursive vs Iterative Functions

We can write this recursively in Python too...

```
def fact_rec(n):
    assert(n >= 0)
    if n == 0:
        return 1
    return n * fact_rec(n - 1)
```

Hint. If you can write a function recursively in Python, you can almost certainly write a function in OCaml.

Thinking functionally

We're not writing procedures anymore.

We're writing *definitions* of values (i.e., expressions).

This is enough to do pretty much anything we want.

So *free yourself from pythonic thinking*. . .

Thinking functionally

We're not writing procedures anymore.

We're writing *definitions* of values (i.e., expressions).

This is enough to do pretty much anything we want.

So *free yourself from pythonic thinking*. . .

Thinking functionally

We're not writing procedures anymore.

We're writing *definitions* of values (i.e., expressions).

This is enough to do pretty much anything we want.

So *free yourself from pythonic thinking*. . .

Thinking functionally

We're not writing procedures anymore.

We're writing *definitions* of values (i.e., expressions).

This is enough to do pretty much anything we want.

So *free yourself from pythonic thinking*. . .

Outline

OCaml Toplevel

The OCaml toplevel is a REPL (Read-Eval-Print-Loop) for the OCaml programming language, similar to the Python interpreter.

`ocaml` runs a simple toplevel, but we will most often use `utop`, a more modern version.

Anything you can write in OCaml you can write at the Toplevel, and vice versa.

OCaml Toplevel

The OCaml toplevel is a REPL (Read-Eval-Print-Loop) for the OCaml programming language, similar to the Python interpreter.

`ocaml` runs a simple toplevel, but we will most often use `utop`, a more modern version.

Anything you can write in OCaml you can write at the Toplevel, and vice versa.

OCaml Toplevel

The OCaml toplevel is a REPL (Read-Eval-Print-Loop) for the OCaml programming language, similar to the Python interpreter.

`ocaml` runs a simple toplevel, but we will most often use `utop`, a more modern version.

Anything you can write in OCaml you can write at the Toplevel, and vice versa.

Let's do a demo.

UTop Directives

There are two kinds of things that can be evaluated by UTop

1. OCaml expressions and definitions
2. Commands called *directives* which augment UTop itself

Directives are always prefixed by "#", e.g., `#quit` for exiting UTop

UTop Directives

There are two kinds of things that can be evaluated by UTop

1. OCaml expressions and definitions
2. Commands called *directives* which augment UTop itself

Directives are always prefixed by "#", e.g., `#quit` for exiting UTop

The `#use` Directive

OCaml code is written in files with the extension "`.ml`"

The `#use "some_file.ml"` directive loads the code in that file into UTop

You should think of the `#use` directive as automatically inputing lines of code *you could have written yourself* into UTop (including directives)

The `#use` Directive

OCaml code is written in files with the extension "`.ml`"

The `#use "some_file.ml"` directive loads the code in that file into UTop

You should think of the `#use` directive as automatically inputing lines of code *you could have written yourself* into UTop (including directives)

The `#use` Directive

OCaml code is written in files with the extension `".ml"`

The `#use "some_file.ml"` directive loads the code in that file into UTop

You should think of the `#use` directive as automatically inputing lines of code *you could have written yourself* into UTop (including directives)

Let's do a demo.

A Coding Workflow

In this course, we will interact with OCaml via the following steps:

1. Edit code in a file `some_file.ml`
2. Open UTop and type
   - `#use "some_file.ml"` (note the quotation marks)
3. Interact with the code in UTop
4. Close UTop (This is important)

Note. This is not the cleanest way to interact with OCaml. Modern users use build systems like dune. If we have time, we'll try to cover this.

A Coding Workflow

In this course, we will interact with OCaml via the following steps:

1. Edit code in a file `some_file.ml`
2. Open UTop and type
   - `#use "some_file.ml"` (note the quotation marks)
3. Interact with the code in UTop
4. Close UTop (This is important)

Note. This is not the cleanest way to interact with OCaml. Modern users use build systems like dune. If we have time, we'll try to cover this.

A Coding Workflow

In this course, we will interact with OCaml via the following steps:

1. Edit code in a file `some_file.ml`
2. Open UTop and type
   - `#use "some_file.ml"` (note the quotation marks)
3. Interact with the code in UTop
4. Close UTop (This is important)

Note. This is not the cleanest way to interact with OCaml. Modern users use build systems like dune. If we have time, we'll try to cover this.

A Coding Workflow

In this course, we will interact with OCaml via the following steps:

1. Edit code in a file `some_file.ml`
2. Open UTop and type
   - `#use "some_file.ml"` (note the quotation marks)
3. Interact with the code in UTop
4. Close UTop (This is important)

Note. This is not the cleanest way to interact with OCaml. Modern users use build systems like dune. If we have time, we'll try to cover this.

A Coding Workflow

In this course, we will interact with OCaml via the following steps:

1. Edit code in a file `some_file.ml`
2. Open UTop and type
   - `#use "some_file.ml"` (note the quotation marks)
3. Interact with the code in UTop
4. Close UTop (This is important)

Note. This is not the cleanest way to interact with OCaml. Modern users use build systems like dune. If we have time, we'll try to cover this.

Let's do a demo

A Note on Main

There is no "main" function in OCaml, but it is typical to define a function called main (when appropriate) which kicks off the intended code.

```
let f x = x + 1
let y = 10

let main () =
  print_endline "Computing 10 + 1...";
  print_endline (string_of_int y)

let _ = main ()
```

UTop Cheat Sheet

Task	Command
exit	`#quit;;` (or *Control-d*)
load code	`#use "PATH/TO/FILE";;`
see all directives	`#help;;`

Anything typed into UTop can must end in `;;` in order to be evaluated.

`;;` can also be used as line endings in OCaml files **but you should avoid doing this.**

Outline

Expressions and Values (At a High Level)

Values are the *things* manipulated by a programs: think the integer 7 or the string "seven".

Expressions (attempt to) *describe* values of programming languages.

Example. The expression `(2 + 7)` describes the value 9, whereas the expression `(2 + "seven")` does not describe anything, it's ill-formed.

Expressions and Values (At a High Level)

Values are the *things* manipulated by a programs: think the integer 7 or the string "seven".

Expressions (attempt to) *describe* values of programming languages.

Example. The expression `(2 + 7)` describes the value 9, whereas the expression `(2 + "seven")` does not describe anything, it's ill-formed.

Expressions and Values (At a High Level)

Values are the *things* manipulated by a programs: think the integer 7 or the string "seven".

Expressions (attempt to) *describe* values of programming languages.

Example. The expression `(2 + 7)` describes the value 9, whereas the expression `(2 + "seven")` does not describe anything, it's ill-formed.

Types

Every value *and expression* in OCaml has a type.

Types are used to delineate what *kind of object* an expression is.
Types *restrict* how expressions can be constructed:

```
let f x = x + x
let y = 2
let final_value = f y

(* we can't do this *)
(* let z = f "two" *)
```

Types

Every value *and expression* in OCaml has a type.

Types are used to delineate what *kind of object* an expression is.
Types *restrict* how expressions can be constructed:

```
let f x = x + x
let y = 2
let final_value = f y

(* we can't do this *)
(* let z = f "two" *)
```

Type Annotations

Types in OCaml are often inferred, so it's usually possible to avoid working explicitly, but it is important that we come to understand how they are inferred.

That said, we can be as explicit as we want about types:

```
let f (x : int) : int = (x + x : int)
let y : int = (2 : int)
let final_value : int = (f y : int)
```

We can annotate expressions, function arguments and let-defined variables with types.

Type Annotations

Types in OCaml are often inferred, so it's usually possible to avoid working explicitly, but it is important that we come to understand how they are inferred.

That said, we can be as explicit as we want about types:

```
let f (x : int) : int = (x + x : int)
let y : int = (2 : int)
let final_value : int = (f y : int)
```

We can annotate expressions, function arguments and let-defined variables with types.

Primitive Types and Values

As with any programming language, OCaml manipulates a collection of standard values with standard types.

Type	Values	Operators
`int`	2, 3, `-101`	`+`, `-`, `*`, `/`, `mod`
`float`	`3.`, `-1.01`	`+.`, `-.` , `*.`, `/.`
`bool`	`true`, `false`	&&, ||, `not`
`char`	`'b'`, `'c'`	
`string`	`"word"`, `"@#$#"`	`^`
`unit`	`()`	

Caveats and Tips

Use parentheses to use binary operators as prefix operators, e.g. `(+) 2 3` is the same as `2 + 3`.

There is no overloading, e.g., `(+)` always refers to *integer* addition.

There are natural conversion functions between primitive types, but **not** `char` and `string`. See the textbook for how to get around this.

It is important to note that `==` and `!=` in Python (and so many other languages are `=` and `<>` in OCaml. We just need to train ourselves to remember this.

Caveats and Tips

Use parentheses to use binary operators as prefix operators, e.g. `(+) 2 3` is the same as `2 + 3`.

There is no overloading, e.g., `(+)` always refers to *integer* addition.

There are natural conversion functions between primitive types, but **not** `char` and `string`. See the textbook for how to get around this.

It is important to note that `==` and `!=` in Python (and so many other languages are `=` and `<>` in OCaml. We just need to train ourselves to remember this.

Caveats and Tips

Use parentheses to use binary operators as prefix operators, e.g. `(+) 2 3` is the same as `2 + 3`.

There is no overloading, e.g., `(+)` always refers to *integer* addition.

There are natural conversion functions between primitive types, but **not** `char` and `string`. See the textbook for how to get around this.

It is important to note that == and `!=` in Python (and so many other languages are = and <> in OCaml. We just need to train ourselves to remember this.

Caveats and Tips

Use parentheses to use binary operators as prefix operators, e.g. `(+) 2 3` is the same as `2 + 3`.

There is no overloading, e.g., `(+)` always refers to *integer* addition.

There are natural conversion functions between primitive types, but **not** `char` and `string`. See the textbook for how to get around this.

It is important to note that == and `!=` in Python (and so many other languages are = and <> in OCaml. We just need to train ourselves to remember this.

Evaluation

Expressions are evaluated, and the process of evaluation results of 3 possibilities:

1. the value described by the expression
2. an *exception* is raised, meaning there was something wrong with the expression itself
3. the evaluation process hangs, due to an infinite loop.

UTop evaluates expressions it is given. It's like an OCaml calculator.

Evaluation

Expressions are evaluated, and the process of evaluation results of 3 possibilities:

1. the value described by the expression
2. an *exception* is raised, meaning there was something wrong with the expression itself
3. the evaluation process hangs, due to an infinite loop.

UTop evaluates expressions it is given. It's like an OCaml calculator.

Evaluation

Expressions are evaluated, and the process of evaluation results of 3 possibilities:

1. the value described by the expression
2. an *exception* is raised, meaning there was something wrong with the expression itself
3. the evaluation process hangs, due to an infinite loop.

UTop evaluates expressions it is given. It's like an OCaml calculator.

Evaluation

Expressions are evaluated, and the process of evaluation results of 3 possibilities:

1. the value described by the expression
2. an *exception* is raised, meaning there was something wrong with the expression itself
3. the evaluation process hangs, due to an infinite loop.

UTop evaluates expressions it is given. It's like an OCaml calculator.

Evaluation (Continued)

Let's do a demo

Evaluation (Continued)

Evaluation is one of the most basic element of writing an interpreter for a programming language.

Since we will eventually be writing our own interpreter, we'll need a better understanding of how things work understanding of how this works.

This means understanding syntax and semantics.

Evaluation (Continued)

Evaluation is one of the most basic element of writing an interpreter for a programming language.

Since we will eventually be writing our own interpreter, we'll need a better understanding of how things work understanding of how this works.

This means understanding syntax and semantics.

Evaluation (Continued)

Evaluation is one of the most basic element of writing an interpreter for a programming language.

Since we will eventually be writing our own interpreter, we'll need a better understanding of how things work understanding of how this works.

This means understanding syntax and semantics.

Syntax and Semantics

Syntax refers to the rules which govern which expressions are well-formed.

Dynamic semantics refers to the rules which govern how expressions are evaluated.

Static semantics refers to the rules which govern the type of an expression.

We have to define each one of these for every construction we consider in OCaml.

Syntax refers to the rules which govern which expressions are well-formed.

Dynamic semantics refers to the rules which govern how expressions are evaluated.

Static semantics refers to the rules which govern the type of an expression.

We have to define each one of these for every construction we consider in OCaml.

Syntax and Semantics

Syntax refers to the rules which govern which expressions are well-formed.

Dynamic semantics refers to the rules which govern how expressions are evaluated.

Static semantics refers to the rules which govern the type of an expression.

We have to define each one of these for every construction we consider in OCaml.

Syntax and Semantics

Syntax refers to the rules which govern which expressions are well-formed.

Dynamic semantics refers to the rules which govern how expressions are evaluated.

Static semantics refers to the rules which govern the type of an expression.

We have to define each one of these for every construction we consider in OCaml.

Outline

If-Expressions in Python

We've seen if-*statements* for control-flow in languages like Python:

```
x = -10
if x < 0:
    x = -x
assert(x >= 0)
```

But Python also has if-expressions:

```
x = -10
abs_value = -x if x < 0 else x

# if-expressions have a value
y = (-x if x < 0 else x) + 2
```

If-Expressions in Python

We've seen if-*statements* for control-flow in languages like Python:

```python
x = -10
if x < 0:
    x = -x
assert(x >= 0)
```

But Python also has if-expressions:

```python
x = -10
abs_value = -x if x < 0 else x

# if-expressions have a value
y = (-x if x < 0 else x) + 2
```

If-Expressions vs. If-Statements

If-statements *change the underlying state of the program* whereas if-expressions *have a value*.

We can think of the if-expression pattern as a function call:

```python
def ifthen(if_case, cond, else_case):
    if cond:
        return if_case
    else:
        return else_case

x = 10
assert((-x if x < 0 else x) == ifthen(-x, x < 0, x))
```

If-Expressions in OCaml

The syntax in OCaml is similar:

```
let x = -10
let abs_value = if x < 0 then -x else x
let y = (if x < 0 then -x else x) + 2
```

Remember. OCaml has no state so we can't have if-statements.

Let's do a demo.

If-Expressions in OCaml (Formally)

Syntax. Given expressions `e1`, `e2`, and `e3`, we have the expression

```
if e1 then e2 else e3
```

Dynamic Semantics.

- If `e1` evaluates to `true` and `e2` evaluates to `v`, then `if e1 then e2 else e3` evaluates to `v`.
- If `e2` evaluates to `false` and `e3` evaluates to `w`, then `if e1 then e2 else e3` evaluates to `w`.

Static Semantics If `e1` has type `bool`, and `e2` and `e3` have type `t`, then `if e1 then e2 else e3` has type `t`.

If-Expressions in OCaml (Formally)

Syntax. Given expressions `e1`, `e2`, and `e3`, we have the expression

```
if e1 then e2 else e3
```

Dynamic Semantics.

- If `e1` evaluates to `true` and `e2` evaluates to `v`, then `if e1 then e2 else e3` evaluates to `v`.
- If `e2` evaluates to `false` and `e3` evaluates to `w`, then `if e1 then e2 else e3` evaluates to `w`.

Static Semantics If `e1` has type `bool`, and `e2` and `e3` have type `t`, then `if e1 then e2 else e3` has type `t`.

If-Expressions in OCaml (Formally)

Syntax. Given expressions `e1`, `e2`, and `e3`, we have the expression

```
if e1 then e2 else e3
```

Dynamic Semantics.

- If `e1` evaluates to `true` and `e2` evaluates to `v`, then `if e1 then e2 else e3` evaluates to `v`.
- If `e2` evaluates to `false` and `e3` evaluates to `w`, then `if e1 then e2 else e3` evaluates to `w`.

Static Semantics If `e1` has type `bool`, and `e2` and `e3` have type `t`, then `if e1 then e2 else e3` has type `t`.

If-Expressions in OCaml (Formally)

Syntax. Given expressions `e1`, `e2`, and `e3`, we have the expression

```
if e1 then e2 else e3
```

Dynamic Semantics.

- If `e1` evaluates to `true` and `e2` evaluates to `v`, then `if e1 then e2 else e3` evaluates to `v`.
- If `e2` evaluates to `false` and `e3` evaluates to `w`, then `if e1 then e2 else e3` evaluates to `w`.

Static Semantics If `e1` has type `bool`, and `e2` and `e3` have type `t`, then `if e1 then e2 else e3` has type `t`.

Syntax. Given expression e1, e2, and e3, we have the expression

```
if e1 then e2 else e3
```

Dynamic Semantics.

$$\frac{e_1 \Longrightarrow \text{true} \qquad e_2 \Longrightarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Longrightarrow v} \qquad \frac{e_1 \Longrightarrow \text{false} \qquad e_3 \Longrightarrow w}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Longrightarrow w}$$

Static Semantics.

$$\frac{e_1 : \text{bool} \qquad e_2 : \text{t} \qquad e_3 : \text{t}}{(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \text{t}}$$

Let-Expressions

Let-expressions allow you do locally define variables *within* an expression.

There is no analog in Python, there is no need, since local variables are defined in local state:

```
def f(x):
    y = x * x
    return y + y
```

In OCaml:

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

But this is doing something very different.

Let-Expressions

Let-expressions allow you do locally define variables *within* an expression.

There is no analog in Python, there is no need, since local variables are defined in local state:

```
def f(x):
    y = x * x
    return y + y
```

In OCaml:

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

But this is doing something very different.

Let-Expressions

Let-expressions allow you do locally define variables *within* an expression.

There is no analog in Python, there is no need, since local variables are defined in local state:

```python
def f(x):
    y = x * x
    return y + y
```

In OCaml:

```ocaml
let f x =
  let y = x * x in
  y + y
```

But this is doing something very different.

Let-Expressions vs. Let-Definitions

The `let` keyword plays two roles: in let-expressions and let-definitions.

Let-definitions define variables at the toplevel. The key difference is that *let-definitions don't have values*:

```
(* we can't do this *)
(* let x = (let y = 3) *)
```

Again, let-expressions define local variables *within* an expression (and are expressions themselves).

Let-Expressions vs. Let-Definitions

The `let` keyword plays two roles: in let-expressions and let-definitions.

Let-definitions define variables at the toplevel. The key difference is that *let-definitions don't have values*:

```
(* we can't do this *)
(* let x = (let y = 3) *)
```

Again, let-expressions define local variables *within* an expression (and are expressions themselves).

Let-Expressions vs. Let-Definitions

The `let` keyword plays two roles: in let-expressions and let-definitions.

Let-definitions define variables at the toplevel. The key difference is that *let-definitions don't have values*:

```
(* we can't do this *)
(* let x = (let y = 3) *)
```

Again, let-expressions define local variables *within* an expression (and are expressions themselves).

Let's do a demo.

Multiple Let-Expressions

It is common to use a sequence of let-expressions to mimic locally-defined variables.

In Python:

```
def squared_dist(x1, y1, x2, y2):
    x_diff = x1 - x2
    y_diff = y1 - y2
    return x_diff * x_diff + y_diff * y_diff
```

In OCaml:

```
let squared_dist x1 y1 x2 y2 =
  let x_diff = x1 - x2 in
  let y_diff = y1 - y2 in
  x_diff * x_diff + y_diff * y_diff
```

How do let-expressions work?

Since let-expressions are expressions, they must be evaluated, so the question really is

How do we *evaluate* let-expressions?

Intuitively, what should this expression evaluate to?

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

Substitution

We will talk *a fair amount* about substitution in this course. One of the hardest problems we'll face in building our own interpreter is getting substitution right.

For now, we will use our intuitions:

```
let x = 5 + 5 in x + 3

(* should evaluate to *)
(* 10 + 3 *)
(* which evaluates to *)
(* 13 *)
```

That is, we substitute the "`x`" in `(x + 3)` with the *value* of `5 + 5`.

Shadowing

We have to be a bit careful though. Consider:

```
let b =
  let x = 4 in
  (let x = 5 in x) + x
```

When we substitute "4" for "x" in `(let x = 5 in x) + x`, we only substitute the *last* one.

Formally speaking, we only substitute free occurrences of `x`.

Hint. This is almost never a problem in your actual code. Just **don't write code like this.**

Shadowing

We have to be a bit careful though. Consider:

```
let b =
  let x = 4 in
  (let x = 5 in x) + x
```

When we substitute "4" for "x" in `(let x = 5 in x) + x`, we only substitute the *last* one.

Formally speaking, we only substitute free occurrences of x.

Hint. This is almost never a problem in your actual code. Just **don't write code like this.**

Shadowing

We have to be a bit careful though. Consider:

```
let b =
  let x = 4 in
  (let x = 5 in x) + x
```

When we substitute "4" for "x" in `(let x = 5 in x) + x`, we only substitute the *last* one.

Formally speaking, we only substitute free occurrences of `x`.

Hint. This is almost never a problem in your actual code. Just **don't write code like this.**

Shadowing

We have to be a bit careful though. Consider:

```
let b =
  let x = 4 in
  (let x = 5 in x) + x
```

When we substitute "4" for "x" in `(let x = 5 in x) + x`, we only substitute the *last* one.

Formally speaking, we only substitute free occurrences of `x`.

Hint. This is almost never a problem in your actual code. Just **don't write code like this.**

Let-Expressions in OCaml (A Bit More Formally)

Syntax. For expressions `e1` and `e2` and *valid* variable name `x`, we have the expression

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

Dynamic Semantics.

- `e1` evaluates to `v1` and
- `e2'` is the expression in which `v1` is substituted for `x` in `e2` and
- `e2'` evaluate to `v2` then
- `(let x = e1 in e2)` evaluates to `v2`

Static Sematics.

- if `e1` is of type `t1` and
- if `e2` is of type `t2` assuming `x` is type `t1` then
- `(let x = e1 in e2)` is of type `t2`

Let-Expressions in OCaml (A Bit More Formally)

Syntax. For expressions `e1` and `e2` and *valid* variable name `x`, we have the expression

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

Dynamic Semantics.

- `e1` evaluates to `v1` and
- `e2'` is the expression in which `v1` is substituted for `x` in `e2` and
- `e2'` evaluate to `v2` then
- `(let x = e1 in e2)` evaluates to `v2`

Static Sematics.

- if `e1` is of type `t1` and
- if `e2` is of type `t2` assuming `x` is type `t1` then
- `(let x = e1 in e2)` is of type `t2`

Let-Expressions in OCaml (A Bit More Formally)

Syntax. For expressions `e1` and `e2` and *valid* variable name `x`, we have the expression

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

Dynamic Semantics.

- `e1` evaluates to `v1` and
- `e2'` is the expression in which `v1` is substituted for `x` in `e2` and
- `e2'` evaluate to `v2` then
- `(let x = e1 in e2)` evaluates to `v2`

Static Sematics.

- if `e1` is of type `t1` and
- if `e2` is of type `t2` assuming `x` is type `t1` then
- `(let x = e1 in e2)` is of type `t2`

Let-Expressions in OCaml (A Bit More Formally)

Syntax. For expressions e1 and e2 and *valid* variable name x, we have the expression

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

Dynamic Semantics.

- e1 evaluates to v1 and
- e2' is the expression in which v1 is substituted for x in e2 and
- e2' evaluate to v2 then
- (let x = e1 in e2) evaluates to v2

Static Sematics.

- if e1 is of type t1 and
- if e2 is of type t2 assuming x is type t1 then
- (let x = e1 in e2) is of type t2

Let-Expressions in OCaml (A Bit More Formally)

Syntax. For expressions `e1` and `e2` and *valid* variable name `x`, we have the expression

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

Dynamic Semantics.

- `e1` evaluates to `v1` and
- `e2'` is the expression in which `v1` is substituted for `x` in `e2` and
- `e2'` evaluate to `v2` then
- `(let x = e1 in e2)` evaluates to `v2`

Static Sematics.

- if `e1` is of type `t1` and
- if `e2` is of type `t2` assuming `x` is type `t1` then
- `(let x = e1 in e2)` is of type `t2`

Let-Expressions in OCaml (A Bit More Formally)

Syntax. For expressions `e1` and `e2` and *valid* variable name `x`, we have the expression

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

Dynamic Semantics.

- `e1` evaluates to `v1` and
- `e2'` is the expression in which `v1` is substituted for `x` in `e2` and
- `e2'` evaluate to `v2` then
- `(let x = e1 in e2)` evaluates to `v2`

Static Sematics.

- if `e1` is of type `t1` and
- if `e2` is of type `t2` assuming `x` is type `t1` then
- `(let x = e1 in e2)` is of type `t2`

Let-Expressions in OCaml (A Bit More Formally)

Syntax. For expressions `e1` and `e2` and *valid* variable name `x`, we have the expression

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

Dynamic Semantics.

- `e1` evaluates to `v1` and
- `e2’` is the expression in which `v1` is substituted for `x` in `e2` and
- `e2’` evaluate to `v2` then
- `(let x = e1 in e2)` evaluates to `v2`

Static Sematics.

- if `e1` is of type `t1` and
- if `e2` is of type `t2` assuming `x` is type `t1` then
- `(let x = e1 in e2)` is of type `t2`

Let-Expressions in OCaml (A Bit More Formally)

Syntax. For expressions `e1` and `e2` and *valid* variable name `x`, we have the expression

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

Dynamic Semantics.

- `e1` evaluates to `v1` and
- `e2'` is the expression in which `v1` is substituted for `x` in `e2` and
- `e2'` evaluate to `v2` then
- `(let x = e1 in e2)` evaluates to `v2`

Static Sematics.

- if `e1` is of type `t1` and
- if `e2` is of type `t2` assuming `x` is type `t1` then
- `(let x = e1 in e2)` is of type `t2`

Let-Expressions in OCaml (More Formally)

Syntax. For expressions `e1` and `e2` and *valid* variable name `x`, we have the expression

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

Dynamic Semantics.

$$\frac{e_1 \Longrightarrow v_1 \qquad e_2[v_1/x] \Longrightarrow v_2}{(\text{let } x = e_1 \text{ in } e_2) \Longrightarrow v_2}$$

Static Sematics.

$$\frac{e_1 : t_1 \qquad e_2 : t_2 \;\text{ given }\; x : t_1}{(\text{let } x = e_1 \text{ in } e_2) : t_2}$$

Scope

(We'll also talk more formally about scope in this course.)

Informal Definition. The scope of a variable where it is meaningful.

- the scope of a let-defined variable is global
- the scope of the variable of a let-expression is exactly the body of the let-expression.

(This is far simpler than in Python, where we have to keep track of what the scope is based on different statements.)

Scope

(We'll also talk more formally about scope in this course.)

Informal Definition. The scope of a variable where it is meaningful.

- the scope of a let-defined variable is global
- the scope of the variable of a let-expression is exactly the body of the let-expression.

(This is far simpler than in Python, where we have to keep track of what the scope is based on different statements.)

Scope

(We'll also talk more formally about scope in this course.)

Informal Definition. The scope of a variable where it is meaningful.

- the scope of a let-defined variable is global
- the scope of the variable of a let-expression is exactly the body of the let-expression.

(This is far simpler than in Python, where we have to keep track of what the scope is based on different statements.)

(We'll also talk more formally about scope in this course.)

Informal Definition. The scope of a variable where it is meaningful.

- the scope of a let-defined variable is global
- the scope of the variable of a let-expression is exactly the body of the let-expression.

(This is far simpler than in Python, where we have to keep track of what the scope is based on different statements.)

Outline

Summary

Values are the *things* manipulated by programs and expression describe values.

There is no state in the functional paradigm.

Remember, we're not just learning OCaml, we're using this as an opportunity to analyze a programming language.

Take some time to practice this stuff. Get your programming environment set up soon.