Lecture 19: Introduction to OCaml
10:00 AM, Oct 19, 2018

Contents

1 Introducing OCaml
2 Elementary types int, float, and bool
3 Tuples
4 Procedure types
5 Applying a procedure
6 Conditional expressions
7 Strings
8 “lambda” expressions
9 Binding variables to values
10 Using pattern-matching with lists
11 Variant Types
12 Parts of patterns
   12.1 Constant Patterns
   12.2 The Wild Card
   12.3 Identifier Patterns
   12.4 Constructor patterns, including tuple patterns and list patterns

1 Introducing OCaml

A family of languages called “ML”, which was originally short for “Meta-Language”. (Another in this family is SML, short for “Standard ML”, which is the language we used in the first year CS17 was taught.) ML was originally implemented in Lisp. It was used in a system for helping with mathematical proofs. Something called the Categorical Abstract Machine (CAM) was being studied.
So this time in France came up with the language Caml, which simultaneously referenced CAM, and ML, and a pun on the French “çà ML”, meaning “that ML”.

Later Caml Light, and then Caml Special Light were implemented, and the logo jokingly made reference to a cigarette brand. When object-oriented programming features were added, it was called Objective Caml, and then just OCaml.

Now OCaml is getting some traction in industry, most famously at Jane St. (a fintech firm) but also at other places, including Facebook.

Though this not the most important difference from Racket, you should know first that OCaml uses parentheses mostly just for grouping (as in algebra) and uses traditional infix notation especially for arithmetic.

```ocaml
# 7+2*5;;
- : int = 17
```

The # is OCaml’s prompt. The double semicolon ;; is how you tell the OCaml REPL that you are done entering the expression; it is not part of the language and should not be used when you are writing OCaml code in a file.

The response tells you three things about the result:

- its name (except in this case there is no name so OCaml prints a dash)
- its type, which is int in this case
- its value, which is 17

The colon could be read “...has type ....” The equals symbol could be read “...has value...”.

Typical precedence rules are used, e.g. multiplication “binds” tighter than addition, so the multiplication is done first.

A type such as int should be considered an (often infinite) set of values. For our purposes, different types are disjoint sets, meaning no data value belongs to more than one type. (This is a simplification; there are subtypes in OCaml, where one type is a subset of another, but this occurs in the object-oriented part of OCaml, which we will not discuss in this course, although I encourage you to explore it after the course ends.)

## 2 Elementary types int, float, and bool

The simplest and most familiar types are int, float, and bool. The type bool has two elements, true and false. There are two kinds of numbers, int and float. For example, 2 is an int, and 2.0 is a float, and they are not the same value.

In fact, the addition operator for ints and the addition operator for floats have different names:

```ocaml
18. +. 17.
- : float = 35.
```
18. /

- : float = 1.05882352941176472

You thus cannot directly add an int to a float. You have to choose whether you are adding ints or adding floats. If you try to use, say, int addition operator on a float, OCaml will complain:

```ocaml
# 2.5 + 2;;
Characters 0-3:
  2.5 + 2;;
    ^
Error: This expression has type float but an expression was expected of type int
```

Equality testing is more complicated; I will come back to that.

## 3 Tuples

From the basic types, you can form *compound types*. The first way to form a compound type is by *Cartesian product*. In set theory, for sets $A$ and $B$, the Cartesian product of $A$ and $B$, written $A \times B$, is the set of pairs $(a, b)$ such that $a$ belongs to $A$ and $b$ belongs to $B$.

For the sets $A = \{1, 2, 3\}$ and $B = \{♥, ♠, ♣, ♦\}$, the Cartesian product is

$$\{(1, ♥), (2, ♥), (3, ♥), (1, ♠), (2, ♠), (3, ♠), (1, ♣), (2, ♣), (3, ♣), (1, ♦), (2, ♦), (3, ♦)\}$$

You can have a Cartesian product of one set with itself. For example:

$$\{1, 2, 3\} \times \{1, 2, 3\} = \{(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)\}$$

Note in both examples that the cardinality of $A \times B$ equals the cardinality of $A$ times the cardinality of $B$. In Mathese,

$$|A \times B| = |A| \cdot |B|$$

Cartesian product is still meaningful when one or both of the sets $A$ and $B$ are infinite. Also, one can have a three-fold Cartesian product $A \times B \times C$, which is a set of triples of the form $(a, b, c)$, or a four-fold Cartesian product, etc. The general word that would describe a pair, a triple, etc. is *tuple*.

In OCaml, we use `*` instead of `×`. We said a type is a kind of set, so the Cartesian product of ints and floats is written `int * float`. To write an expression whose value is of type `int * float`, we use a notation similar to that in math:

```ocaml
# (10 + 7, 3.5 *. 2.5);;
- : int * float = (17, 8.75)
```

Remember, though, parens are mostly for grouping, so we can write a tuple expression without them:

```ocaml
# 10 + 7, 3.5 *. 2.5;;
- : int * float = (17, 8.75)
```
OCaml prefers to print tuples with parens.

# 2 * 4, 3.5 -. 2.5, false;;
- : int * float * bool = (8, 1., false)

I said a compound type is formed from basic types, but of course you can also form a compound type from any types, including compound types. For example, you could have a pair whose first element is a pair of ints and whose second element is a boolean. The type would then be (int * int) * bool.

# ((1,2), true);;
- : (int * int) * bool = ((1, 2), true)

4 Procedure types

Let $A$ and $B$ be sets. The Mathese notation for the set of all mathematical functions that take inputs in $A$ and give outputs in $B$ is

$$A \rightarrow B$$

OCaml, like many languages, refers to procedures as functions. OCaml uses the function notation for types. The type of a procedure that takes inputs that are ints and produces outputs that are floats is

$$\text{int} \rightarrow \text{float}$$

which is written int -> float. Thus procedure types are also compound types.

For example, some built-in procedures are (named) sqrt, abs, abs_float, cos, ceil, floor. Can you guess the types? We can enter these names and check the types:

# abs;;
- : int -> int = <fun>
# sqrt;;
- : float -> float = <fun>
# abs;;
- : int -> int = <fun>
# abs_float;;
- : float -> float = <fun>
# ceil;;
- : float -> float = <fun>

As with ints and so on, OCaml prints a dash (instead of a name), the type, and some representation of the value. You might recall that Racket’s way of printing the value of a procedure was not very informative:

> (lambda (x) 1)
#<procedure>
> +
#<procedure:+>
OCaml’s printed representation of any procedure value is just `<fun>`.

You might be surprised that `ceil` returns a float. I was. How can you get an `int` out of a `float`?

```
# int_of_float;;
- : float -> int = <fun>
```

It should be clear that procedure types are compound types. A procedure type can be made from other compound types. For example, a procedure that takes as input a pair of `ints` and outputs a `bool` is of type `(int * int) -> bool`.

Okay, what is the type of this kind of procedure?

- **input**: a procedure that takes an `int` and produces a `float`
- **output**: a `bool`

Let’s break it down. The type of the input is `int -> float`. The type of the output is `bool`. Therefore the type of the procedure is `(int -> float) -> bool`.

What is the type of this kind of procedure?

- **input**: an `int`
- **output**: a procedure that takes an `int` and produces a `int`

**Answer**: `int -> (int -> int)`

So far we’ve only seen the types of procedures that each take only one argument. That’s because, in OCaml, that is all there is! Why is that okay? Suppose you want a procedure that takes two arguments, an `int` and a `bool`, and produces a `float` as output. You cannot have that, but there are two alternatives:

- A procedure that takes as a single argument a tuple consisting of an `int` and a `bool` (and that outputs a `float`.
- A procedure that
  - takes an `int` as argument, and
  - produces as output a procedure that takes as input a `bool` and produces a `float` as output.

Replacing a procedure of the first kind with one of the second is called currying, after the logician Haskell Curry because it was introduced by Gottlob Frege.

What are the types?

- The type of the first alternative is `(int * bool) -> float`
- The type of the second alternative is `int -> (bool -> float)`

In fact, OCaml doesn’t use parens in writing types, so

- The type of the first alternative is `int * bool -> float`
- The type of the second alternative is `int -> bool -> float`
5 Applying a procedure

How does one construct an expression in which a procedure is applied to an argument? Just take an expression for the procedure followed by an expression for the argument:

```
# sqrt 2.0;;
- : float = 1.41421356237309515
```

Parens can be thrown in as desired. If you are nostalgic for Racket, you can write (sqrt 2.0). If you have been exposed to the disease that is Java, you might want to write sqrt(2.0). OCaml doesn’t care. You can even use extra parens: sqrt((2.0)).

If you enter an expression whose evaluation would result in the application of a procedure to an argument that is of the wrong type, OCaml will not even try to evaluate the expression. It will reject it immediately. Try to predict what happens when you enter this:

```
sqrt (1/0);;
```

OCaml will not complain about division by 0. Instead, it will complain that the input to sqrt is of type int but should be of type float.

OCaml is type-checking the expression before it tries to evaluate it. This is considered a feature, not a bug. Catching errors early is helpful. Many many bugs are caught by type-checking. Many people write something complicated in OCaml, and spend lots of time fixing type errors, and finally OCaml accepts the program—and it works! This is helpful because debugging type errors is much easier than debugging other kinds of errors.

6 Conditional expressions

OCaml has something analogous to Racket’s if and cond: conditional expressions (or call them if expressions)

```
if 17 < 18 then 3.14 else 2.71828
```

The type is float because the value of the whole expression is a float.

What about this?

```
if 17 < 18 then 3.14 else 2
```

This doesn’t type-check! OCaml will not even evaluate it. OCaml requires that an expression be of a single type. The type cannot depend on the results of evaluations.

Warning to those who have programmed in some other languages like Java or c. This is different from the if statements in those languages. Those languages do have conditional expressions but the syntax is different.

OCaml has another feature that often substitutes for conditionals, called pattern-matching.
7 Strings

OCaml does not have symbols but it does have type string. A string does not act as a variable but it is easy to put strings together:

```ocaml
# "hello" ^ "world";;
- : string = "helloworld"
```

This is called concatenation. The caret is the concatenation operator.

8 “lambda” expressions

Of course OCaml has lambda but it is called fun or function. I'll talk about the difference later.

```ocaml
# function x -> x+1;;
- : int -> int = <fun>
```

The name of the (single) formal argument comes right after the function. Then comes an arrow, then the body of the procedure.

The type is int -> int. How did OCaml know? It analyzed the body of the procedure. The + operator requires that its operands are ints, so OCaml inferred that x must be bound to an int. The result of + is an int, so the output of the whole procedure is an int. This process is called type inference.

You can also tell OCaml what you think about the type of the argument. This is called type annotation. It is helpful because type inference is sometimes mysterious.

```ocaml
# function (x: int) -> x+1;;
- : int -> int = <fun>
```

Guess the type of function (x,y,z) -> if x then y+z else y-z.
Answer: bool * int * int -> int.

Guess the type of function x -> (fun y -> x + int_of_float y).
Answer: int -> float -> int.

9 Binding variables to values

The closest thing to Racket's define in OCaml is let.

```ocaml
# let x = 3+4;;
val x : int = 7
```

The let line is a declaration, not an expression. It doesn’t have a value.
There is a difference between Racket’s `define` and OCaml’s `let` but we will discuss it another time.

However, one consequence of this difference is that you cannot define recursive procedures just using `let`. You must use `let rec`.

```
# let fac = function n -> if n = 0 then 1 else n * fac (n-1);;
Error: Unbound value fac
# let rec fac = function n -> if n = 0 then 1 else n * fac (n-1);;
val fac : int -> int = <fun>
```

## Lists

OCaml provides a nice way to form a list out of a collection of expressions. You use square brackets and separate the expressions using a single semicolon:

```
# [1+2; 3+4; 5+6];;
- : int list = [3; 7; 11]
```

Note the type: it is an `int list`. This means that every element of the list is an `int`. OCaml requires every element of a list to have the same type.

```
# [false; not false];;
- : bool list = [false; true]
```

Cons is replaced by an infix operator consisting of two colons, as shown below:

::

For example:

```
3::[2; 1];;
- : int list = [3; 2; 1]
```

Note: :: is not a procedure. It is called a constructor. I’ll talk about the difference later.

Note that list is not a type. You can make types using `list`, such as `int list` and `float list` and `string list` and `int list list`. (The last is the type of a list whose items are `int` lists.)

Okay, but what about an empty list? We can write it as `[]`.

```
# 1::[];;
- : int list = [1]
# 2::1::[];;
- : int list = [2; 1]
```
Let’s write a procedure that, given a nonnegative integer \( n \), produces a list consisting of \( n, n-1, n-2, \ldots, 1 \).

```ocaml
# let rec countdown = fun n -> if n = 0 then [] else n::countdown (n-1);;
val countdown : int -> int list = <fun>
# countdown 10;;
- : int list = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]
```

What is the type of the empty list?

```ocaml
# [];;
- : 'a list = []
```

The quote mark followed by the letter a is a type parameter. This basically says that a list can have any type as its element type.

## 10 Using pattern-matching with lists

OCaml provides a pattern-matching operation that, among other things, can be used on lists. I’ll illustrate this by writing a procedure to test whether a list is empty. Of course, you could do this quite simply, by using = to compare the list to [], but that wouldn’t illustrate pattern-matching!

```ocaml
let emptyP = function mylist ->
  match mylist with
  | [] -> true
  | x::y -> false;;
```

The body of the procedure consists of a match expression. The match expression specifies an expression (in this case just the variable mylist) and a series of clauses separated by vertical bars —. Each clause consists of a pattern and and error and then an output expression. To evaluate the match expression, OCaml evaluates the given expression and then tries to match the expression’s value against each pattern, first to last. When OCaml finds a pattern that matches the value, it evaluates the expression to the right of the corresponding arrow; that expression’s value is then the value of the match expression.

Let’s say the procedure emptyP is applied to the list [1; 2; 3]. The formal variable mylist is bound to the list, and then the body of the procedure is evaluated. Evaluating the body means evaluating the match expression. The list is compared to the pattern in the first clause, [] but does not match, so the list is next compared to the pattern in the second clause, a::b. This pattern matches anything that can be obtained by consing something onto something else. Because [1; 2; 3] can be obtained by consing 1 onto [2; 3], the pattern-match succeeds, so the corresponding expression false is evaluated (resulting in the value false). That, then is the value of the match expression, and hence that is the value returned by the procedure.

Note that the order of the pattern-matching clauses is significant. OCaml tries the patterns in order, and goes with whichever matches first. However, the above procedure emptyP would still work fine if we switched the order of clauses. It’s ideal if this is true of every match expression you write.
The above example did not use the full power of pattern-matching. When $[1; 2; 3]$ is successfully matched against the pattern $a::b$, temporary bindings are created in which $a$ is bound to the car of the list and $b$ is bound to the cdr of the list. These variables can appear in the expression to the right of the corresponding arrow.

Let’s use this power to write `car`.

```ml
let car = function mylist ->
  match mylist with
  | x::y -> x
```

If you enter this in OCaml, it prints the following warning:

this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:

`[]`

We can infer from this that, if `car` were applied to an empty list, there would be an error because in that case none of the clauses would match. (That’s okay because `car` is not supposed to be applied to an empty list.)

Try similarly writing `cdr`.

Finally, let’s use pattern-matching to write `length`.

```ml
let rec length = function mylist ->
  match mylist with
  | [] -> 0
  | x::y -> 1 + length y
```

This is lovely but there is an even more concise way to write this, using the keyword `function`. We’ll see it later.

By the way, I’ve used the variables $x$ and $y$ in these examples but the code would be more readable if these were replaced with more meaningful variable names such as `first` and `rest`.

```ml
let rec length = function mylist ->
  match mylist with
  | [] -> 0
  | first::rest -> 1 + length rest
```

# 11 Variant Types

A list has two forms: it can be empty or it can be formed by consing something onto another list. OCaml allows you to define new types whose values can have multiple forms, even recursive ones. For now, we will stick with nonrecursive types.

The following is a definition of a declaration of a variant type (or just variant for short).
type season =
    | Fall
    | Winter
    | Spring
    | Summer

The word season is the name of this variant type. The four words that follow are constructors. This is an apt name for them, since they are used to construct values of the season type.

The name of a type (like any other OCaml identifier) must begin with a lowercase letter. In contrast, every constructor name must begin with a capital letter. (That’s how OCaml distinguishes constructors from identifiers.)

Once this variant type is defined, the constructors can be used to construct values of type season. For example, the value Fall has type season.

You can also write match expressions using the constructors as patterns, such as:

match this_season with
    | Fall -> 55
    | Winter -> 25
    | Spring -> 65
    | Summer -> 90

For example, if this_season has value Fall then the value of the above match expression is 55.

**Example 2** You can define more interesting variants using constructors that take arguments. To do so, you use the keyword of. For example,

type card =
    | Clubs of int
    | Spades of int
    | Diamonds of int
    | Hearts of int

You can then supply an argument of the specified type to one of the constructors, producing a value of the variant type. For example, Hearts 13 is of type card.

Note that the keyword of appears only in the type definition, not in values.

Now, let’s write a procedure that determines the rank of a card. (The rank of a card is the number on the face of the card.) Here’s how we can do it:

let rank = function c ->
match c with
    | Clubs n -> n
    | Diamonds n -> n
    | Hearts n -> n
    | Spades n -> n;;
Here's how to interpret this code: \(c\) is the card, and we say if \(c\) was constructed with the \texttt{Clubs} constructor and argument \(n\), the answer is \(n\). If \(c\) was constructed with the \texttt{Diamonds} constructor and argument \(n\), the answer is \(n\), and so on.

So, let's apply our procedure to some card, and see what happens!

\[
\text{rank (Hearts 4)};;
\]

- : int = 4

One thing special about the variant type \texttt{card} is that each of the constructors takes an argument of type \texttt{int}. In general, each constructor can take an argument of a different type, and often that is key to the usefulness of the variant type. For example, here is a variant type that is capable of representing two kinds of shapes.

\[
type \texttt{shape} = \texttt{Circle of float} | \texttt{Rectangle of float * float}
\]

In this case, the shape can be a circle, which is specified by a single number (the radius) and the shape can be a rectangle, which is specified by two numbers (the width and the height).

Let's write a procedure to compute the area of a \texttt{shape}.

\[
\text{let area = function myshape ->}
\]

\[
\text{match myshape with}
\]

\[
| \text{Rectangle (width, height) -> width *. height}
| \text{Circle radius -> 3.14159 *. radius *. radius}
\]

The type of this procedure? It takes in a value of type \texttt{shape} and produces a value of type \texttt{float}, so its type is \texttt{shape -> float}.

This time I used meaningful variable names in the patterns.

It's time to mention how the keyword \texttt{function} can be used to make code more concise. Note that in all the examples we've seen so far, the expression matched against the patterns is just the formal variable. That is, the value that is matched against the patterns is just the actual argument. This is not at all required by a match expression but it happens so often that there is a convenient way to write it.

The keyword \texttt{function} means "This is a lambda expression that matches the actual argument against the following patterns." when you use \texttt{function} instead of \texttt{fun}, you can skip specifying the formal argument and introducing the match expression—you just give the clauses. Here's how \texttt{area} can be written.

\[
\text{let area = function}
\]

\[
| \text{Rectangle (width, height) -> width *. height}
| \text{Circle radius -> 3.14159 *. radius *. radius}
\]

I suggest you revisit the examples we have seen so far and rewrite them using the keyword \texttt{function}. Try them out in OCaml.
12 Parts of patterns

Here we discuss pattern-matching in somewhat greater generality.

Patterns are built using four forms: constants, wild card, identifier, and constructor.

12.1 Constant Patterns

A constant pattern is the simplest of all atomic patterns. It matches only itself.

Here is a simple example of a match expression with constant (integer) patterns:

```ocaml
match num with
  | 1 -> "Too small."
  | 2 -> "Right on!"
  | 3 -> "Too big."
```

This expression evaluates exactly as you’d expect. The value of `num` matches the pattern 1 if that value is itself 1, matches the pattern 2 if it is itself 2, and matches the pattern 3 only if it is itself 3.

We can use the above match expression in a procedure.

```ocaml
let guess = function num ->
  match num with
    | 1 -> "Too small."
    | 2 -> "Right on!"
    | 3 -> "Too big."
```

(Try rewriting this procedure using the keyword function.)

The value of the expression `guess 2` is the string "Right on!".

You can use values of basic types (int, string, float, bool) for constant patterns.

12.2 The Wild Card

To achieve the effect of an else clause, OCaml has a special pattern, spelled `_` and pronounced wild card, which matches any expression. For example,

```ocaml
match num with
  | 1 -> "Too small."
  | 2 -> "Right on!"
  | 3 -> "Too big."
  | _ -> "Pick a number between 1 and 3"
```

If the value of `num` is not 1, 2, or 3, the first three patterns will not match it, but the fourth pattern will always match so the value of the match will be the string "Pick a number between 1 and 3".

It is a bit risky to use the wild card in this way because it might mean you don’t pick up on some problem in your code.
12.3 Identifier Patterns

Like a wild card, an identifier pattern matches any value. ("Identifier" is programming-language-ese for "variable.")

Moreover, if an identifier pattern matches some value, then the identifier is temporarily bound to the value it matches for the purpose of evaluating the expression to the right of the corresponding arrow. Because of this, it's a bad idea to use as a pattern-matching identifier any identifier that already has a meaning, such as one of the arguments to your procedure, or the procedure name.

For example:

```ocaml
let guess = function num ->
  match num with
  | 1 -> "Too small."
  | 2 -> "Right on!"
  | 3 -> "Too big."
  | n -> "The number " ^ string_of_int(n) ^ " is not between 1 and 3"

# guess 4;;
- : string = "The number 4 is not between 1 and 3"
```

12.4 Constructor patterns, including tuple patterns and list patterns

Finally, a constructor pattern matches a constructor, such as one declared in a variant type declaration. We've seen these in several examples. The simplest was the match expression involving a value of variant type season.

A slightly more complicated example was the rank procedure (which takes an input of variant type card). This example illustrated that if the constructor takes an argument, the corresponding pattern can have a pattern as argument. In rank, the pattern for the argument used an identifier pattern, but you can use any kind of pattern here.

Another example was the area procedure. In the second clause, the constructor pattern was Circle with an associated argument pattern radius. That is, again the pattern for the argument was an identifier pattern. In the first clause, the argument pattern was (width, height). What kind of pattern is this?

You should think of the parentheses and commas as a constructor for tuples. Thus (width, height) is a pattern that matches a two-element tuple (a pair). In this case width and height are identifier patterns. If you want to match just Rectangles whose heights are 2, you could use the pattern

```
Rectangle (width, 2)
```

You might be tempted to think you could match squares (Rectangles whose width and height are equal) using the same identifier twice

```
Rectangle (x, x)
```

but OCaml does not allow this.

Finally, returning to lists, you should think of brackets and semicolons as forming a constructor.
For example,

\[ (_; 2; _)_ \]

matches all three-element int lists whose second elements are 2.

A special case is the pattern

\[ [] \]

which matches only the empty list, as we saw in our length procedure.

Similarly, the :: operator (cons) is a kind of constructor as well, as we saw in our procedures car and length. To match only nonempty int lists whose first elements are 7, you could use the pattern

\[ 7::\text{rest} \]

or just

\[ 7::_ \]

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback).