CSCI 1103 Computer Science 1 Honors

Fall 2022

Robert Muller - Boston College

Lecture Notes

Week 2

Topics:

1. Tuples
2. Names for Values: top-level `let` and the `let-in` Expression
3. Function Definitions and Calls

---

1. Tuples

In addition to the built-in base types `int`, `float`, `char` and `unit` and the built-in type `string`, OCaml provides several so-called *structured types* which can aggregate values together in one way or another. The simplest structured type is the *product type*. The expressions of product type are called *tuples*.

```
# (2 + 3, "Alice" ^ " B. " ^ "Toklas", 1.618);;                   (1)
- : (int * string * float) = (5, "Alice B. Toklas", 1.618)         (2)
```

A tuple is a sequence of expressions separated by commas. Tuples are usually bounded by parentheses though the parentheses aren't strictly necessary. Line (1) shows a 3-tuple, or *triple*, consisting of component expressions of the heterogeneous types `int`, `string` and `float`. Line (2) shows that the 3-tuple in (1) is an expression of *product* type `int * string * float`. The stars appearing in product types are usually pronounced "cross".

> It's worth noting that tuples are so handy that they are featured various programming languages including Python, Swift, Rust and Go.

Simplification

Line (1) in the example shows that the components of a tuple can be arbitrarily complex expressions. A tuple is simplified by simplifying all of its components -- the order doesn't matter. The triple in line (1) is not a value because neither `2 + 3` nor `"Alice" ^ " B. " ^ "Toklas"` are values. The triple requires 3 simplification steps to arrive at the value shown in line (2).

```
(2 + 3, "Alice" ^ " B. " ^ "Toklas", 1.618) ->
  (5, "Alice" ^ " B. " ^ "Toklas", 1.618) ->
  (5, "Alice B. " ^ "Toklas", 1.618) ->
  (5, "Alice B. Toklas", 1.618)
```

`fst` & `snd` for pairs

The pervasive functions `fst` and `snd` work on the special case of 2-tuples or *pairs*.

```
# fst ('A', "Gertrude");;
- : char = 'A'

# snd ('A', "Gertrude");;
- : string = "Gertrude"
```

So we can retrieve the two parts of a pair using these two built-in functions. It turns out that there is a more general way to retrieve the components of a tuple, we'll discuss this in a little bit.

2. Naming Values: top-level `let` & the `let-in` Expression

Earlier we saw that OCaml provides a variety of built-in symbolic names via the `Pervasives` module in the Standard Library. These included (infix) operators `+`, `-`, etc, names for various constants, e.g., `max_int` as well as function names such as `int_of_float`, etc. Names defined in other Standard Library modules can be referenced by prefixing the name with the module name. For example, to refer to the `length` function in the `String` module we would write `String.length`.

In this section we introduce two important forms for associating symbolic names with values. The two forms are:

1. top-level `let`
2. `let-in`

1. Top-level `let`

The top-level `let` form introduces a symbolic name for use anywhere within the file or REPL session. The top-level `let` form:

```
  let variable = expression
```

associates the name `variable` with the value of expression `expression`.

> The symbol `let` is called a *keyword*, i.e., a symbol that has special meaning to OCaml.

```
# let width = 5.0 *. 2.0;;
val width : float = 10.0

# width;;
- : float = 10.0

# let pi = acos(-1.0);;
val pi : float = 3.14159265358979312

# pi *. 2.0;;
- : float = 6.28318530717958623
```

Programming languages generally have fixed rules governing the formation of symbolic names. They usually also have more restrictive non-binding but recommended *conventions* about how names should be formed. In OCaml, the camelCase and snake_case conventions are both common. We'll use camel-case in this course. You should use whichever style suits you, but try not to mix the styles.

> **Advice** : Good coders think carefully about how to name things. A good name conveys to the reader the role of the name in the code and is reasonably short. For example, `interestGroups` is a reasonable name for a set of interest groups while `abc` would be a poor one.

2. `let-in`

The `let-in` form introduces a symbolic name for use within a restricted area of code. In particular, in the form

```
let variable = expression1 in expression2
```

The only meaningful uses of the name `variable` are in `expression2`.

```
# let one = 1 in one + one;;
- : int = 2
```

In this case `expression2` is `one + one` so there are exactly 2 meaningful uses of the name `one`. Note that the following is ill-defined

```
# let number = one in number + number;;
Error: unbound variable one
```

because the symbol `one` is undefined, notwithstanding its definition above.

A Conventional Style for Writing Cascaded `let-in` Expressions

The `let-in` form is ubiquitous in OCaml code, it is especially common for `expression2` to be yet another `let-in` form as in

```
# let one = 1 in let two = one + one in two + two;;
- : int = 4
```

In the above, `expression2` is the `let-in` expression `let two = one + one in two + two`. When `let-in` expressions are cascaded in this way, they are usually written vertically as in

```
let one = 1 in
let two = one + one
in
two + two
```

Note that the keyword `in` for the last (rightmost) `let-in` is on its own line and is lined-up directly beneath the corresponding `let` keyword.

Simplification of `let-in` Expressions

A `let-in` expression `let variable = expression1 in expression2` is simplified in 3 steps:

1. Simplify `expression1` to its value V (if it has one);
2. plug V in for the occurrences of `variable` in `expression2`;
3. simplify the result of step 2.

```
let one = 1 in let two = one + one in two + two ->  (* 1 is a value, plug it in *)
  let two = 1 + 1 in two + two ->
  let two = 2 in two + two ->                          (* 2 is a value, plug it in *)
  2 + 2 ->
  4                                                    (* 4 is a value, all done *)
```

The expression reaches a value in 4 simplification steps.

> Exercise: Simplify `let pair = (1 + 1, 2 + 2) in (fst pair) * (snd pair)`
>
> ```
> let pair = (1 + 1, 2 + 2) in (fst pair) * (snd pair) ->
>  let pair = (2, 2 + 2) in (fst pair) * (snd pair) ->
>  let pair = (2, 4) in (fst pair) * (snd pair) ->
>  (fst (2, 4)) * (snd (2, 4)) ->
>  2 * (snd (2, 4)) ->
>  2 * 4 ->
>  8
> ```

> The expression reaches a value in 6 simplification steps.

Implicit and Explicit Types

In the top-level `let` and `let-in` examples above, we left the issue of typing to OCaml, we wrote our code and let OCaml sort out whether or not our code was well typed and if so, what type it had. The `let` forms optionally allow a programmer to assert the type of a variable. For example, we could as well have written

```
# let one : int = 1 in one + one;;
- : int = 2
```

Here we are using the type annotation `: int` to make two assertions: 1. that `expression1`, i.e., has type `int`, and 2. that the uses of the variable in `expression2` require an `int`. Of course we might be wrong (!), in this case, OCaml (and/or OCaml's Merlin support in the Atom editor) will let us know:

```
# let one : char = 1 in one + one;;
Error: This expression has type int but an expression was expected of type char
```

Or

```
# let one : int = 1 in one ^ "Uh oh!";;
Error: This expression has type int but an expression was expected of type string
```

A programmer might choose to use explicit types as a way to have OCaml check their logic, i,.e., to confirm or refute their logic. Explicit types are also used in some cases simply as machine-checkable *documentation* for future readers of the code.

It is more common to leave the code without the clutter of explicit type information, and to *let the code speak for itself*. That is, the body of the function is taken to express all of the constraints required on the inputs to the function as well as the type of the output.

Patterns & Pattern Matching

The notation `let variable = expression1 in expression2` is actually a simplified form of what OCaml allows. In particular, OCaml allows `let pattern = expression1 in expression2` where `pattern` is a form compatible with the type of `expression1`. For example, if `expression1` yields a pair of integers, we can write

```
# let (m, n) = (1 + 1, 2 + 2) in m + n;;
- : int = 6
```

This simplifies in the expected way

```
let (m, n) = (1 + 1, 2 + 2) in m + n ->
let (m, n) = (2, 2 + 2) in m + n ->
let (m, n) = (2, 4) in m + n ->    (* (2, 4) is a value, plug 2 in for m, 4 in for n *)
2 + 4 ->
6
```

Patterns and pattern match work for top-level `let`s too.

```
# let (i, (j, k)) = (1 + 1, (2 + 2, 3 + 3));;
val i : int = 2
val j : int = 4
val k : int = 6

# j + k;;
- : int = 10
```

The Don't-Care Variable

In some situations we're interested in matching a structure but we aren't interested in all of it, only part of it. In this case we can use the special *don't-care* variable `_`.

```
# let (m, _, n) = (1 + 1, "Alice", 2 + 2);;
val m : int = 2
val n : int = 4
```

3. Function Definitions and Calls

- Functions are the main tool for packaging up computation in almost all programming languages;
- Functions have *definitions* and *uses* (aka *calls* or *applications*);
- Function definitions are usually stored in text files with the **.ml** extension.

Function Definitions

A function can be defined using either the top-level `let` form or the `let-in` expression:

```
let functionName var1 ... varn = expression

let functionName var1 ... varn = expression1 in expression2
```

In both of these we're using `functionName` to denote the name of the newly defined function. The symbols `var1`, ..., `varn` are *variables* and `expression` is the *body* of the function.

Occurrences of variables to the left of the equal sign in a function definition are called *formal parameters* or sometimes just *parameters*. They represent input portals to the function. There are likely other occurrences of these variables in `expression`. Function names follow the same naming conventions as ordinary variables. (In fact, they are ordinary variables.)

```
# let greeting name = "Hello " ^ name ^ "!";;
val greeting : string -> string

# let double n = n * 2;;
val double : int -> int
```

The notation `string -> string` is a type indicating that `greeting` is a function mapping strings to strings.

Implicit and Explicit Type Information for Function Definitions

For the `greeting` and `double` functions above, we relied on OCaml to sort out the type information. We can be explicit if we like

```
let greeting (name : string) = "Hello " ^ name ^ "!";;   (* Explicit input type only *)
let greeting name : string = "Hello " ^ name ^ "!";;     (* Explicit output type only *)
let greeting (name : string) : string = "Hello " ^ name ^ "!";;     (* input & output *)
```

Function Calls

A function *use*, or *call* or *application* has the form:

```
functionName expr1 ... exprn
```

where each of `expr1`, ..., `exprn` is an *expression*. It's common to enclose the entire expression in parentheses.

```
(functionName expr1 ... exprn)
```

> NB: We do not write the more familiar mathematical notation `functionName(expr1, ..., exprn)`.

Simplification of Function Calls

Consider a simple integer division function `div` that returns both the quotient and the remainder:

```
(* div : int -> int -> int * int
*)
let div m n =
  let quotient = m / n in
  let remainder = m mod n
  in
  (quotient, remainder)
```

Simplifying a call `div (3 + 5) (2 * 3)` of the `div` function would proceed as follows.

```
div (3 + 5) (2 * 3) ->
  div 8 (2 * 3) ->
  div 8 6 ->         (* 8 and 6 are values, plug 8 in for m and 6 in for n *)
  let quotient = 8 / 6 in let remainder = 8 mod 6 in (quotient, remainder) ->
  let quotient = 1     in let remainder = 8 mod 6 in (quotient, remainder) ->
  let remainder = 8 mod 6 in (1, remainder) ->
  let remainder = 2 in (1, remainder) ->
  (1, 2)
```

The value is obtained in 7 simplification steps.

How does a defined function do its job? The defined function is *called* in an attempt to simplify the call to a value.

1. When a function call is evaluated, for each i = 1,..,n, `expri` is evaluated. This process may produce a value Vi, it may encounter an error or it may not stop (more on that later).
2. If for each i, the evaluation of `expri` produces a value Vi, the value Vi is plugged-in for `vari` in the function body (i.e., each occurrences of vari is -replaced- by value Vi).
3. Then the body of the function is simplified and the value of the body of the function is the value of the call of the function.

```
let double n = n * 2

double (double (1 + 2)) ->
  double (double 3) ->
  double (3 * 2) ->
  double 6 ->
  6 * 2 ->
  12
```

Functions can also be defined for use within a limited area using `let-in`.

```
let min3 p q r = min p (min q r) in min3 8 4 6 ->
  min 8 (min 4 6) ->
  min 8 4 ->
  4
```

In OCaml, as in almost every other programming language, computations are bundled up in functions.

Monomorphism & Polymorphism

The `div` function above has type `int -> int -> int * int`. This type is an example of a *monomorphic type*. This means that the definition of the `div` function is such that there are constraints requiring the two inputs to be of `int` type and resulting in the output being of product type `int * int`. Since the coder provided no explicit type annotations, OCaml figured this out on its own.

Some function definitions don't impose any constraints on the input. For example, consider the following `makePair` function.

```
# let makePair x = (x, x);;
val makePair : 'a -> 'a * 'a = <fun>

# makePair 343;;
- : int * int = (343, 343)

# makePair "Alice";;
- : string * string = ("Alice", "Alice")
```

The body of the `makePair` function simply puts the input in both positions of a 2-tuple. There are no type-specific operators applied to the input that might restrict the type, `makePair` can be applied to values of *any* type. The `makePair` function is said to be *polymorphic*. The symbol `'a` in the type `'a -> 'a * 'a` is known as a *type variable*, it can range over any type.

> It's worth noting that the coder didn't need to do anything at all to confer such flexibility to this function, the coder simply let the code speak for itself!

All Functions are Functions of One Input

The `div` function above appears to require two inputs, one for `m` and one for `n`. In fact there is a subtlety here involving something called *Currying* which we'll discuss in more detail in a few weeks. In fact, the `div` function and all other functions in OCaml are functions of exactly one argument. Not zero, not two, etc., just one. We can obtain the effect of multiple argument functions by packaging the arguments up in a tuple. We might write

```
(* div : int * int -> int * int
*)
let div (m, n) =                        (* NB: the pair pattern for the input! *)
  let quotient = m / n in
  let remainder = m mod n
  in
  (quotient, remainder)

div (8, 6)
```

Pipes

When coding with functions, it's common to route or "pipe" the output of one function to the input portal of another.

```
let f x = ...
let g y = ...
let h z = ...

let b = f(a) in
let c = g(b) in
let d = h(c)
in
d
```

```
              +-----+            +-----+            +-----+
       a ---->|  f  +----------->|  g  +----------->|  h  +----> d
              +-----+            +-----+            +-----+
```

In many cases, it isn't necessary to name the intermediate results. One could type `h(g(f(a)))` but this is a little hard on the eyes so the *pipe* operator is useful.

```
let d = a |> f |> g |> h

let d =
  a |> f
    |> g
    |> h
```

This is sometimes thought of as a stream of data routing through filters or transformers. Piping is quite common, in the Unix command shell, it's common to pipe data from one command to another.

```
> ls | grep "myData.txt"
```