OCaml Introduction: Tuples and Lists

Jeff Meister

CSE 130, Winter 2011

So far, we have only dealt with expressions of single values of a single type, like `5 : int` or `9.7 : float` or `"cse130" : string` (note that an OCaml `string` is not made of multiple `char`s, but has its own built-in type). There are plenty of combinations left to consider. What about compound values? The language has a few built-in constructs for these, including tuples and lists. The simple overview: a tuple can hold values of different types, but only a fixed number of them; a list can hold an unlimited number of values, but only of the same type.

You might well ask, what about holding an unlimited number of values of different types? I would reply that such a collection is nearly useless. Because every function has exactly one input type, you can't call any function on every element of the collection, so why collect them together? Counting up how many you have is about all you can do. The fixed-length restriction of the tuple allows us to pair up each contained value with its corresponding type, and then we know which functions are OK to call on which values.

Of course, it would be acceptable to form a list of values of different types *if* you defined a new type that is the logical disjunction of each different one you need (say, `int` *or* `float` *or* `int -> string`), along with a tag so that every value of this new type is labeled to distinguish which case of the *or* it is. OCaml provides a mechanism to do exactly this, which we will see later on.

1 Tuples

For now, back to tuples and lists. Here is a 4-tuple of someone's personal information:

```
# let person = ("larry", 47, 165, 'M');;
val person : string * int * int * char = ("larry", 47, 165, 'M')
```

In general, the type of an n-tuple is written as an ordered sequence of n types, separated by `*`. The `*` is pronounced "cross" (think cross product), but more intuitively it means *and*. So, this personal information contains a string (name), an int (age), another int (weight), and a char (sex). However, the tuple itself is not written using `*`, because in expression-land that is the integer multiplication operator. To write down an n-tuple, we write the n values in order between parentheses and separated by commas.

Any tuple that has a different number of values inside will have a different type than any other tuple, and these types are not compatible. If I add a new element for height, I will have a 5-tuple; functions written for 4-tuples will not work on 5-tuples without modification, and vice versa.

The way to get values back out of a tuple is by ***pattern matching***. This is an important concept that you'll see throughout the course. A pattern is like an expression, but it appears on the left side of an `=` or `->` instead of on the right side. The pattern "looks like" the value of the expression it's matching, but if you put names where the sub-expressions would go, OCaml will bind the names to the corresponding values for you. Like so:

```
# let (name, age, _, sex) = person;;
val name : string = "larry"
val age : int = 47
val sex : char = 'M'
```

Because `person` is a 4-tuple, I need a pattern that looks like 4 things inside parens separated by commas. I'd like to extract the name, age, and sex, so I put those names into the appropriate spots in the pattern. I don't care about the person's weight for whatever reason, but because I'm writing a 4-tuple pattern, I must fill in that spot; for this purpose, OCaml provides the dummy pattern name `_`. Notice that all three values are extracted in one shot; conceptually, the pattern matches them all simultaneously.

1.1 The `unit` type

Besides n-tuples for $n \geq 1$ (a 1-tuple is just a single, i.e., a plain type with no `*`s like we had before), there is also a 0-tuple. To understand it, think of types as sets of values. The type `int` represents the set of all signed 31-bit integers; `char` represents the set of all unsigned 8-bit integers; the type `float * int` represents the set of all `float-int` pairs; and so on. The 0-tuple has type `unit`, and this type represents the set containing exactly one element, namely `()` (also pronounced "unit"). This unit value conveys no information. It is useful only when that's precisely what you want: to indicate that there is no value, and that's OK. For example, the function `print_string` takes a string as input, causes the external effect of writing it to `stdout`, and then has nothing to return. Because all functions in OCaml must take an argument and return a value, `print_string` instead returns the `()` value. Moreover, the function `print_newline` simply terminates the current line on `stdout`; it doesn't even require any input. Again, `unit` is employed.

```
# print_string;;
- : string -> unit = <fun>
# print_newline;;
- : unit -> unit = <fun>
# print_string "the objective is caml"; print_newline ();;
the objective is caml
- : unit = ()
```

Notice (and yet also ignore) the single semicolon sequencing the two print function calls. We have not seen this yet. It works simply by evaluating the expression on the left, throwing the value away, and then evaluating the expression on the right. The only reason you would e**valu**ate an expression with the intent to discard its **valu**e is if the expression has some other side effect (like printing to the screen). Usually, in this case, the value discarded will be `()`. OCaml will warn if you try to discard a value of any other type. In general, unless you're inserting debugging print calls into your code, you *should not* be doing imperative-style statement sequencing with the semicolon. The following code is no good, and trying to write this will produce horrible results, either ugly compiler errors or many points deducted on exams:

```
let x = 1;              (* WRONG *)
let y = 2;              (* BAD IDEA *)
return (x + y, x - y);  (* DON'T DO THIS *)
```

The proper way to sequence code like this is with *let-in*:

```
let x = 1 in
let y = 2 in
(x + y, x - y)
```

2 Lists

We do have a use for the semicolon, though. Whereas tuples were written with their elements inside parentheses and separated by commas, lists are written with their elements inside square brackets and separated by semicolons. Here's one way to write the list of integers from 1 to 5:

```
# [1; 2; 3; 4; 5];;
- : int list = [1; 2; 3; 4; 5]
```

Unlike tuples, we're not listing the type of every element in the type of the collection, because they are all required to have the same type, which in this case is `int`. Any list containing any number of `int` elements has type `int list`, even the empty list, which contains `0` `int`s and is written `[]` (pronounced "nil").

The key operation on lists is the infix operator `::`, pronounced "cons" (think **cons**truct a new list). It takes an element and sticks it at the *front* of an existing list, returning a new list as a result (the existing list is not modified and cannot ever be modified). The expression `1 :: [2; 3; 4; 5]` takes the element `1` and puts it at the front of `[2; 3; 4; 5]` to yield the new list `[1; 2; 3; 4; 5]`. In fact, the bracket-and-semicolon way of writing things is just *syntactic sugar*, a notational convenience. That means the following expressions are equivalent ways of writing the same list value:

```
[1; 2; 3; 4; 5]
1 :: [2; 3; 4; 5]
1 :: 2 :: [3; 4; 5]
1 :: 2 :: 3 :: [4; 5]
1 :: 2 :: 3 :: 4 :: [5]
1 :: 2 :: 3 :: 4 :: 5 :: []
1 :: (2 :: (3 :: (4 :: (5 :: []))))
```

The parentheses are only included in the last case to show that cons is *right-associative*, unlike most binary operators like `+`.

Because the syntactic sugar is based on (and rewritten to) the more fundamental cons notation, I will not consider it for the moment. That leaves us with precisely two ways of creating lists. We can either build the empty list `[]`, or we can build a list by cons-ing an item to an existing list, which must contain items of the same type. In fact, that is exactly the definition of a list, as built into OCaml:

```
type 'a list = [] | :: of 'a * 'a list
```

The `'a`, pronounced "alpha", is a type variable. The definition expresses the following facts: `[]` is an `'a list`; also, `x :: y` is an `'a list` *provided that* `x` is a single `'a` and `y` is an `'a list`. This is just what I said above. To put it another way, the type system enforces that only `int`s can be "consed" onto an `int list` with `::`. Trying to do otherwise will elicit a type error:

```
# 1 :: ["foo"; "bar"; "baz"];;
Error: This expression has type string but an expression was expected of type
         int
```

OCaml complains: you are trying to cons `1`, an `int`, onto a list, so the list had better contain `int`s; but `"foo"` is a `string`.

Type variables such as `'a` are OCaml's way of doing *parametric polymorphism*, like Java generics. Java has classes like `ArrayList<T>` that take a type parameter for the elements of the list; you can instantiate `ArrayList<Integer>` or `ArrayList<String>`, and so on. Similarly, you can have `int list` or `string list` in OCaml, by instantiating the `'a` type parameter. However, you do not have to write these instantiations, because OCaml will figure them out for you! We'll see later on how it does that.

2.1 List pattern matching

As with tuples, lists are not very useful unless we can extract and operate on the items inside them. But unlike tuples, we can't tell from the type of a list how many items there are. It could be the empty list `[]` with no items at all; or it could be a nonempty list, with one item like `1 :: []`, with two items like `1 :: 2 :: []`, or with any unlimited positive number of items. Destructuring a list with *let* will cause a

warning to this effect:

```
# let x = [1; 2; 3; 4; 5];;
val x : int list = [1; 2; 3; 4; 5]
# let head ::  tail = x in head + 10;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
- : int = 11
```

The proper way to get values out of a list is by using *match-with*, a more powerful construct that allows you to test whether your list matches empty or nonempty patterns, in addition to extracting and binding the head and tail values in the nonempty case. Using it makes the warning go away:

```
# match x with
    | [] -> 0
    | head :: tail -> head + 10;;
- : int = 11
```

An important point: the entire *match-with* construct is an expression that returns a value, like *if-then-else* and most other things in OCaml (and unlike the *if* and *switch* statements you know from C or Java). That means I can write something like:

```
let answer = match something with
  | pattern1 -> expr1
  | pattern2 -> expr2
  | ...
  | patternN -> exprN
```

Like all values in OCaml, the one we just named `answer` must have a single type. However, which value we get depends on which `expr` we evaluate, which itself depends on what `pattern` matches the value of `something`. The type system cannot predict this; therefore, **every `expr1` through `exprN` must have the same type**. As a consequence, unlike similar constructions in imperative languages, you cannot have a case of a *match-with* that "does nothing". Using the dummy `unit` value introduced earlier for one case would only be acceptable if every other case also had type `unit`.

Thus, I must provide an expression for the empty list case, even though I know it's not going to be evaluated at run time. All cases (introduced by `|`) must have the same type, so I arbitrarily chose the `int 0`. If you need to fill in a *match* case and you really cannot produce a sensible value, you can write `assert false`, which indicates to OCaml that your program is never intended to reach that point at run time. If it does, you'll get an exception with the location of the failure. I could have written that instead of `0`, but *only* because I am *sure* that the value of `x` is not `[]` (I just bound it to a nonempty list). As we'll see below, and in your assignments, you'll be writing recursive list functions where that assumption no longer holds.

Anyway, in both of these pattern-matchings, we have extracted just one item from the front of the list and added 10 to it, producing a single `int`. That's not very interesting. How can we access all the other items? Well, our pattern bound the name `head` to the first `int 1`, and it bound `tail` to the rest of the `int list [2; 3; 4; 5]`. If we could just repeat the same process on the `tail`, of binding its `head`, adding `10`, then proceeding on *its* `tail`, and so on, we would eventually add `10` to all the numbers. Let's write a recursive function to do exactly that, and keep track of the answers it produces in a list:

```
# let rec add10 nums = match nums with
    | [] -> []
    | head :: tail -> head + 10 :: add10 tail;;
val add10 : int list -> int list = <fun>
# add10 x;;
- : int list = [11; 12; 13; 14; 15]
```

Think about how this function works. Here, the empty list case is important: not only could someone pass an empty list to this function, but the empty list is also the base case of its recursion. Fortunately, we have a sensible value of type `int list` to return: if you want a list that's just like `[]` but with `10` added to all its numbers, then you just want `[]`, because there are no numbers in there to begin with. In the nonempty list case, we add `10` to the `head` as before, but additionally, we stick the result on the front of the list that we get from recursively adding `10` to all the numbers in the `tail`, using the same function `add10`. Make sure you are **convinced** that this works, and that you're not just writing such functions down without knowing why you wrote them and believing they are true, or this class will quickly become very difficult to understand!

It's not so bad though, just think about what expression the function is building up, and trace through its evaluation. Each time, it does `head + 10 ::` then makes a recursive call using `tail`. The first time we call the function on `x`, the parameter `nums` matches the pattern `head :: tail`, with `head = 1` and `tail = [2; 3; 4; 5]`. So we have `11 ::`, and now we have to call the function again on `tail`. This time, parameter `nums` matches the pattern `head :: tail` with `head = 2` and `tail = [3; 4; 5]`, so we have `11 :: (12 ::`, and we call the function again. This process continues until we reach the base case of `[]`, at which point we have built the expression

```
11 :: (12 :: (13 :: (14 :: (15 :: []))))
```

Again, the parens are just for clarity; because of operator precedence I didn't need to write them in the function. Now we have an expression we know how to evaluate! Of course, it results in the list value `[11; 12; 13; 14; 15]`.

You'll see plenty more functions on lists very soon, but I want to reinforce one more thing: using a pattern like `h :: t` to bind names to the components of a list does not modify the original list. **Lists are immutable; there is no way to modify them.**