CSci 2041: Advanced Programming Principles
Introduction to OCaml

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Spring 2015

An Overview of the OCaml Language

There are actually two parts to the language:

- a *core language* in which you construct the actual pieces of your code
- a *modules language* for combining these pieces to realize larger functionality

Our plan of study

- we will first focus on the core language and use it to understand basic programming principles
- we will later look at the modules language and use it to understand principles for building big systems

An Arial View of the Core Language

- takes seriously the *expression evaluation* view of programming

  leave the idea of "changing variable values" behind with lower-level languages
- focuses almost exclusively on *data values*

  let the compiler handle representation issues
- is *strongly* and *statically* typed

  every expression must have a *unique* type *before* execution
- has a *rich* type system that supports *polymorphism*
  - user can identify new kinds of data with associated types
  - user can parameterize data and operations by types
- has the capability of often *inferring* the "right" types
- allows functions to be treated almost like any other data

Running OCaml Programs

You can run OCaml programs in two modes:

- as standalone pieces of code with no reference to OCaml
- in an interactive mode with OCaml "watching over" what you are doing

We will initially use the latter mode which is better suited to code development

We will see later how to run our developed programs as standalone entities

An Important Note: OCaml programs are *compiled* in either way of running them

In other words, do not let anyone confuse you into believing that "interactive" is the same as "interpreted"

Interacting with OCaml

When you start OCaml up, it will put you in a loop in which

- you give it an expression
- it will analyze what you have given it
- it will evaluate the expression if it passes the analysis and show you the result

For example, here is an interaction

```
lalit:~/teaching/umn/CSCI2041/code> ocaml
        OCaml version 4.02.1

# 2 + 3 ;;
- : int = 5
# "hello" ^ " world" ;;
- : string = "hello world"
# ...
```

Some Simple Types and Expressions in OCaml

What expressions we can write are tied to types in OCaml

A few simple types supported directly by OCaml to get started

- `int` the type for integers
  sequences of digits, `+`, `-`, `*`, `/`, `mod`, etc
- `float` the type for floating point numbers
  digit sequences separated by a period, `+.`, `-.`, `*.`, `/.`
- `char` the type for characters
  character within single quotes, case and code conversions
- `bool` the type of booleans
  `true`, `false`, `||`, `&&`, `<`, `>`, `=`, `<>`, `if-then-else`
- `string` the type of strings
  character string within double quotes, `^`, `"hello".[1]`

Hickey, Chapter 2 and OCaml library links have more details

Interacting with OCaml (Contd)

Of course, OCaml cannot give you a “good result” for everything you present it

In particular, there can be three kinds of problems

- the input may have a *syntax error*
- the input may have a *type error*
- the input may cause a execution or *runtime* error

Learning to read these error messages can save frustration and can also be very rewarding

We will look quickly at what they are here, use the second lab to really start understanding what they mean

Syntax Errors

These errors arise when the input is ill-formed to an extent that OCaml cannot even parse it

For example

```
# 2 + ;;
Characters 4-6:
  2 + ;;
      ^^
Error: Syntax error
#
```

Type Errors

These errors arise when the input can be parsed but operations are being applied to data they cannot possibly work on

For example,

```
# 2 + "string";;
Characters 4-12:
  2 + "string";;
      ^^^^^^^^
Error: This expression has type string but an
       expression was expected of type
         int
#
```

Really understanding type errors requires us to learn more about the type system of OCaml

However, make this a central goal: fixing type errors often leads right away to a correct program

Runtime Errors

These can be of two kinds

- the application of operations to inappropriate data that cannot be determined by type checking

  Some examples

```
# 3 / 0;;
Exception: Division_by_zero.
# "hello".[7];;
Exception: Invalid_argument "index out of bounds".
#
```

- expressions whose evaluations never terminate

We will learn how design programs so that these kinds of errors can be avoided from the start

Function Expressions in OCaml

The expressions we present to OCaml can also be *function valued*, not just simple values such as integers or booleans

The simplest form for such an expression is the following

```
fun <arg> -> <exp>
```

where `<arg>` is allowed to be used in `<exp>`

A concrete example in OCaml

```
# fun x -> 2 + x ;;
- : int -> int = <fun>
#
```

Note the type shown for the expression and the way the expression itself is shown

Terminology: `<arg>` is called the *(formal) argument* of the function and `<exp>` is called its *body*

Applying Function Expressions

The usual purpose for constructing functions is that we can then *apply* them to arguments

We do this in OCaml by writing the function and its *actual* argument next to each other, i.e. by *juxtaposing* them

A concrete example of an application

```
# (fun x -> 2 + x) 3 ;;
- : int = 5
#
```

Notice the use of parentheses; we can *always* use them like this to indicate grouping

Function Application and Type Checking

Recall that OCaml is a strongly typed language

This means that not all function applications will yield acceptable expressions

In particular, an expression of the form `(e1 e2)` is acceptable only if

- `e1` has a type of the form `T1 -> T2`, and
- `e2` has a type that matches `T1`

For example, `(fun x -> 2 + x) 3` satisfies these requirements

However, `(fun x -> 2 + x) true` does not

Functions of Multiple Arguments

The body of a function can *itself* be a function expression

If it is, then what we get is a function of *multiple arguments*

A concrete example of this kind is the “plus” function below

```
# fun x -> (fun y -> x + y ) ;;
- : int -> int -> int = <fun>
#
```

OCaml drops parentheses in the type it shows us by assuming `->` associates to the right

If we reinsert these parentheses, we see that what we have here is a function that returns a function as a result

Note also that we could have dropped the parentheses in the expression we wrote

Applying Multiple Argument Functions

As we have just noted, when we apply multiple argument functions, what we get back are functions themselves

For example, consider the following

```
# (fun x -> fun y -> x + y) 3;;
- : int -> int = <fun>
```

Observe that the resulting expression is *still* function valued

Question: Can you describe what the function does?

We can apply the function again to a suitable argument

```
# ((fun x -> fun y -> x + y) 3) 2 ;;
- : int = 5
#
```

Incidentally, application associates to the left so we can drop the second set of parentheses

A Shorthand for Multiple Argument Functions

Multiple argument functions are used so often that OCaml gives us a shorter way of writing them

In particular, instead of writing

```
fun <arg1> -> ... -> fun <argn> -> <exp>
```

we can write the following

```
fun <arg1> ... <argn> -> exp
```

Here is a concrete example showing the plus function and its application

```
# (fun x y -> x + y) 3 2 ;;
- : int = 5
#
```

Naming Expressions

OCaml allows us to name expressions for later (re)use

The structure of a "naming expression"

```
let <name> = <exp>
```

A simple example of such a binding expression

```
# let x = 5;;
val x : int = 5
#
```

OCaml tells us `x` has become bound to a value of type `int`

The binding *scopes* over everything that follows

```
# let x = 5;;
val x : int = 5
# let y = 2 + x;;
val y : int = 7
#
```

Naming Expressions (Contd)

More interesting examples of binding occur when we bind names to function expressions

For example, we can name our “plus” function as follows

```
# let plus = fun x y -> x + y ;;
val plus : int -> int -> int = <fun>
#
```

Observe and understand what OCaml is telling us in response

We can now use the name in place of the expression itself

```
# (plus 2) 3 ;;
- : int = 5
#
```

A Simplified Notation for Naming Functions

Since naming function expressions occurs so often, OCaml gives us a simpler notation for it

Instead of writing

```
let <name> = fun <arg1> ... <argn> -> <exp>
```

we can write

```
let <name> <arg1> ... <argn> = <exp>
```

For example, we could have bound the name `plus` as follows

```
let plus x y = x + y
```

Comment: Feeling now like function definitions in Java or C, but simpler notation and more general mechanisms!

Names versus Variables

Names in OCaml are very different from variables in a language like Java or C

For example consider the following sequence of input to OCaml

```
# let x = 5;;
...
# let plusx y = x + y;;
...
# let x = 21;;
...
# plusx y;;
```

Question: What value will OCaml display at the end?

Another important point to note: A name is interpreted to be whatever it is bound to *at the point where it is used*

This is what is known as *static* or *lexical* scoping, an important principle for modularity

Localizing Name Bindings

The bindings created by `let` that we have seen govern everything that follows

The *let expression* allows us to limit such bindings to a particular expression

The form for such an expression

```
let <name> = <exp1> in <exp2>
```

An example of the use of such an expression

```
# let x = 5;;
val x : int = 5
# let x = 2 in x + 7;;
- : int = 9
# x;;
- : int = 5
#
```

Nesting Let Expressions

Recall the form for a let expression

```
let <name> = <exp1> in <exp2>
```

In this form, `<exp2>` can be any permitted expression, *including another let expression*

For example, consider the following interaction

```
# let x = 5 in let y = 7 in x + y;;
- : int = 12
#
```

This kind of expression is called a *nested let*

Since let bindings can be for functions, we can use such lets to give us nested function definitions as we will see later

Expressions Must be Closed in Context

An important property of expressions in OCaml

> *Names and constants used in an expression must be known in the context in which the expression appears*

For example, here is what happens if `x` has not been previously bound by a let

```
# 2 + x ;;
Characters 4-5:
  2 + x ;;
      ^
Error: Unbound value x
#
```

The requirement also makes sense: how can we evaluate the expression if what `x` stands for is not known?

Like the “definition-before-use” convention in other languages

Defining Recursive Functions

One place where the definition-before-use restriction causes problems is in *recursive definitions*

E.g., consider the following attempt to define the factorial function:

```
let fact =
    fun n ->
     if (n = 0) then 1 else n * (fact (n - 1)) ;;
```

Ocaml will signal an error because we are using `fact` before it has been defined

However, in principle this definition seems okay

In particular, we are trying to define the behaviour of a *function* on an argument by using its behaviour on (other) arguments

At worst, we may have a function that does not terminate on some arguments

Defining Recursive Functions (Contd)

OCaml permits a violation of the definition-before-use principle but only for functions and only if we "warn" it beforehand

To warn OCaml of this, we use the word `rec`

```
let rec fact =
    fun n ->
     if (n = 0) then 1 else n * (fact (n - 1)) ;;
```

Using the shorthand for function definitions we could also write

```
let rec fact n =
  if (n = 0) then 1 else n * (fact (n - 1)) ;;
```

Mutually Recursive Functions

Sometimes, we may want to define a collection of functions that are mutually recursive

E.g., consider the even and odd functions for positive integers

- a (positive) number n is even if
  - it is zero, or
  - if it is not zero and $(n-1)$ is odd
- a (positive) number n is odd if it is not zero and $(n-1)$ is even

In such cases, you combine all the definitions into one big `let` using the word `and`

```
# let rec even n =
    if (n = 0) then true else odd (n-1)
  and odd n =
    if (n = 0) then false else even (n-1);;
```

Digression: Programs in OCaml

We have been acting up to this point as if everything in the interaction with OCaml happens at the prompt

However, this is typically *not* how you want to program

- if you enter everything at the interaction level, it is all lost when you quit OCaml
- you also want to *compose* your code carefully, without pressure from OCaml!

Instead, you compose all the expressions you want OCaml to process in a file and then read this file to start an interaction

Note that later this will become the file that you compile to produce a standalone version of your code

When you follow this approach, you do not need to terminate each expression with two semicolons

Digression: Comments in OCaml

Like with any other language, it is important to include comments in your programs

Comments in OCaml are enclosed within `(*` and `*)`

Typical things to include in the comments for function definitions

- the type of the function, to be compared with what OCaml infers
- the assumptions about its input (precondition)
- how you expect the output to be related to the input (invariant)

E.g. for the factorial function I might write the following

```
(* fact : int -> int
   precondition : input must be positive
   invariant: output is factorial of input *)
```

Polymorphism in OCaml

The functions we have seen up to this point have all applied to one kind of data; e.g., `fact` and `even` apply only to integers

However, we can think of functions that carry out their operation *without heed* to the specific type of the input

An example of this kind is the *identity* function

- it can apply to integer input
- it can apply to boolean input
- it can apply to a function on integers
- in fact, it can apply even to itself

A beauty of OCaml: it allows us to define functions that are *polymorphic* in this kind of way

The main requirement: the function should work the *same* way at *all* the types

Polymorphism in OCaml (Contd)

OCaml allows for polymorphism through the following mechanisms

- it allows for variables, denoted by tokens with a leading `'`, in types

  such variables denote all possible types

- it allows for types with such variables to be associated with functions

  the function works at every type obtained by instantiating the variables in its type

- it allows such variables to be instantiated when checking if a function can be properly applied to an argument

  by instantiating type variables, we are essentially picking to use the function at a particular type

A Polymorphism Example

Consider the following definition in OCaml

```
# let id x = x;;
val id : 'a -> 'a = <fun>
#
```

Intuitively, the type inferred for `id` says it can take an object of any type and will return something of the same type

We can then think of the following applications of `id`

- `(id 5)`: type-checks by instantiating `'a` with `int`
- `(id true)`: type-checks by instantiating `'a` with `bool`
- `(id fact)`: type-checks by instantiating `'a` with `int -> int`
- `(id id)`: type-check by instantiating `'a` with `'b -> 'b`

In the last case, we have renamed the type variable in the type of the second `id` which is acceptable

Tuples as an Example of Structured Data

All the types we have treated up to now are *unstructured* in the sense that they do not have sub-components

The simplest kind of structured data in OCaml is the *tuple*

A tuple is constructed from two data objects simply by writing them down next to each other and separated by a comma

For example, `1, true` is a tuple

The type of a tuple is `ty1 * ty2` if `ty1` and `ty2` are the types of the components

For example `1, true` has the type `int * bool`

Mathematically, a tuple corresponds to an ordered pair and the type corresponds to a cartesian product

Lists as an Example of an Inductive Type

Lists are perhaps the simplest form of an *inductively defined* type that we have seen in the past

The collection of lists is described as follows

- a structure containing nothing is an (empty) list
- a structure with a “head” element and a “tail” which is a list is a list

The first clause gets us started, the second clause allows us to construct bigger and bigger lists as we go along

Also, something can be list *only* in one of these two ways

Another thing to note: the structure of a list is independent of the type of the element, i.e. it is polymorphic in this sense

However, we want to treat only lists in which all the elements are of the *same* type

Lists in OCaml

The first issue to deal with is to provide some way of talking about a type that

- distinguishes lists from other types like `int`, `bool`, etc
- allows us to talk about lists whose elements can be of arbitrary type, and
- in any given instance, forces all the elements to be of the same type

To support these requirements simultaneously, OCaml uses the idea of a *type constructor*

A type constructor is a function on types that takes other types as arguments to yield a type of a particular structure

For lists, OCaml provides the type constructor `list` that can be used to yield the types `(int list)`, `(bool list)`, etc

Note that application for types is written backwards

Lists in OCaml (Contd)

Once we have figured out what type to use, the only remaining things to figure out are

- how to indicate the empty list
- how to indicate a list with a particular head and tail

To realize this, OCaml provides the following *value constructors*

```
[]      of type ('a list)
::      of type ('a * ('a list) -> ('a list))
```

Also, the constructor `::`, pronounced "cons", is treated as an infix and right associative operator

For example, the list with `1` and `2` as its elements is written as `1 :: 2 :: []`

Observe that these constructors are also polymorphic, but in the controlled way we desire

A Special Notation for Lists

Lists are used so frequently in OCaml that there is a special notation for them

We can show them by writing their elements separated by semicolons between square brackets

For example, the list `(1 :: 2 :: 3 :: [])` can also be written as `[1; 2; 3]`

Decomposing Structured Data

When we have structured data, we typically want a means for decomposing them into their parts

For example, given a tuple, we might want to extract its left and right components

Similarly, given a (non-empty) list, we may want to extract its head and its tail

This kind of operation is known as *destructuring* or, more simply as *destructing* structured data

To support *destructing*, OCaml provides *patterns* and *pattern matching*

Patterns and Pattern Matching

Patterns are expressions that are constructed using value constructors over identifiers

For example the following are all patterns

```
(f,s)     (h::t)     []
```

Pattern matching is a process that

- tries to check if a pattern matches an actual data item, and
- binds components of the data to the identifiers if successful

For example

- `(f,s)` matches `(1,2)` and binds `f` to `1` and `s` to `2`
- `(h::t)` matches `[1;2]` and binds `h` to `1` and `t` to `[2]`
- `(h::t)` *does not* match `[]`

Important: Pattern and data must be of the *same* type

The Match or Case Expression

The *match* or *case* expression in OCaml uses pattern matching

The format of the match expression

```
match <exp> with
  | <pat_1> -> <exp_1>
      ...
  | <pat_n> -> <exp_n>
```

The meaning of such an expression

- evaluate `<exp>` and then try to match it with each `<pat_i>` in turn
- bind the identifiers in the *first* pattern that matches and evaluate the corresponding expression
- if no pattern matches, raise a “match exception” error

Note that identifiers in the pattern can be used in the expression

Examples of Match Expressions

Decomposing a tuple of integers and adding them

```
# match (3,4) with
    | (x,y) -> x + y;;
- : int = 7
#
```

Checking if a given list is empty

```
# match [2] with
    | [] -> true
    | (h::t) -> false;;
- : bool = false
#
```

Using Match in Function Definitions

The expression to be matched can be the argument of a function too

A function for adding a tuple of numbers

```
# let plus' ip =
    match ip with
     | (x,y) -> x + y;;
val plus' : int * int -> int = <fun>
# plus' (3,4);;
- : int = 7
```

Observe the type of `plus'`

Question: How does this `plus'` compare with the function `plus` we saw earlier?

Another Example of Match in a Function

We can similarly define a function for checking list emptiness

```
# let empty lst =
  match lst with
    | [] -> true
    | (h::t) -> false;;
val empty : 'a list -> bool = <fun>
# empty [];;
- : bool = true
# empty [1];;
- : bool = false
# empty [["string"]];;
- : bool = false
#
```

Observe that OCaml is able to infer a type for `empty`; we will understand how it does this soon

Observe also that this type is polymorphic

Detecting Match Exceptions at Compile Time

When there is no pattern that matches a given data item, this causes a *runtime error* called a match exception

In many cases, OCaml can determine the possibility of such an error at *compile time* and will warn you about it

A concrete example

```
#  let nonempty lst =
    match lst with
     | (h::t) -> true;;
    Characters 22-56:
  ....match l with
        | (h::t) -> true..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val nonempty : 'a list -> bool = <fun>
#
```

Such warnings *can* alert you to deeper semantical errors

Recursive Functions on an Inductive Type

We often want to carry out computations on data of an inductive type that has the following character

- we distinguish between the different cases for the data item
- in each case, we do a calculation on the components possibly using recursion and then compose the result

For example, consider adding up the numbers in an integer list

- we distinguish the cases of an empty and a non-empty list
- we treat the two cases as follows
  - if the list is `[]`, the sum is `0`
  - if the list has the form `(h::t)`, we calculate the sum for `t` in the same way and then add `h`

Match and recursion are “made to fit” for such computations!

Example: Summing Up an Integer List

As a simple example, here is how the definition of `sumlist` would play out

```
let rec sumlist lst =
    match lst with
      | [] -> 0
      | (h::t) -> h + (sumlist t)
```

A Convenient Notation for Functions with Matching

The definition of `sumlist` is actually a shortened form of

```
let rec sumlist =
    fun lst ->
     match lst with
        | [] -> 0
        | (h::t) -> h + (sumlist t)
```

The combination

```
fun <id> -> match <id> with ...
```

appears so often that OCaml lets it be simplified to `function`

For example, we could have defined `sumlist` as follows

```
let rec sumlist =
     function
        | [] -> 0
        | (h::t) -> h + (sumlist t);;
```

Example: Appending Two Lists

We want to define a function `append` that combines the elements in two lists

For example, given the list `[1; 2]` and `[3; 4]` it should return the list `[1; 2; 3; 4]`

Note also that "appending" does not pay attention to the particular type of the elements of the list

Thus, appending `[true; false]` and `[false; true]` should also work and should give us the list `[true; false; false; true]`

In other words, `append` should have the following type

```
append : ('a list) -> ('a list) -> ('a list)
```

Appending Two Lists (Example Contd)

The definition can be structured by cases on the first list

- if the list is `[]`, then the result is simply the second list
- if the list is `(h::t)`, then we first append `t` to the second list and then add on `h` at the head

This translates easily into the following

```
let rec append lst1 lst2 =
    match lst1 with
      | [] -> lst2
      | (h::t) -> h :: (append t lst2)
```

Type Inference in OCaml

OCaml actually infers a type for `append` based on our definition

```
# let rec append lst1 lst2 =
      match lst1 with
        | [] -> lst2
        | (h::t) -> h :: (append t lst2);;
val append : 'a list -> 'a list -> 'a list = <fun>
#
```

Note that it has figured out a type, in fact the one we want, *without* our giving it *any* type information

Note also that this type information is useful; e.g, it will tell us that `(append 1 [2])` is bad even at compile time

The Question: How does OCaml do this?

Type Inference in OCaml (Contd)

Each newly defined identifier is initially given a most general type which is then refined in the course of type checking

The refinement uses some fairly basic principles, e.g.

- known constants must be used at instances of their types
- a function must have a type of the form `T1 -> T2`
- argument's type must match that needed by the function
- same type for the two branches of an `if-then-else`
- identical types for the patterns in each case of a `match`
- identical types for the result of each case in a `match`
- same type for every occurrence of a bound identifier
- same type for every occurrence of an identifer that is being defined recursively

If the refinement is successful, we get a '"most general" type for the identifier; of course, it could also produce a type error

Type Inference Applied to Append

```
let rec append lst1 lst2 =
    match lst1 with
      | [] -> lst2
      | (h::t) -> h :: (append t lst2)
```

- The process starts by assigning the type `'a` to `append`
- Based on the header, this gets refined to `'b -> 'c -> 'd`
- Looking at the first match case, this becomes `('b list) -> 'c -> 'c`
- Looking at the second match case, this is further refined to `('b list) -> ('b list) -> ('b list)`
- This fits the use of `append`, thus type checking succeeds, confirming the inferred type

An Alternative Definition for Append

We could have defined another version of append as follows

```
let rec append' =
    function
      | ([],lst2) -> lst2
      | ((h::t),lst2) -> h :: append' (t,lst2)
```

What is the difference?

`append'` takes a *single* argument that must be a *pair* of lists

More specifically, OCaml infers the following types for the two

```
val append : 'a list -> 'a list -> 'a list = <fun>
val append' : 'a list * 'a list -> 'a list = <fun>
```

When a function is defined so that it can take its arguments one after the other, it is said to be in *curried* form

Curried or Uncurried?

Which version should we prefer?

The curried version permits "partial application," e.g. consider

```
# let app12 = (append [1; 2]);;
val app12 : int list -> int list = <fun>
# (app12 [3; 4]);;
- : int list = [1; 2; 3; 4]
#
```

Partial application, which has some modularity and efficiency benefits, does not work with the uncurried version

The uncurried version has the benefit of more familiar syntax

```
# append' ([1;2],[3;4]);;
- : int list = [1; 2; 3; 4]
#
```

However, note that the parentheses here are only for grouping

More Examples of Recursive Functions on Lists

Lets consider defining a few more recursive functions on lists

- `member : 'a -> 'a list -> bool`

  `(member x lst)` evaluates to `true` if `x` appears in `lst` and to `false` otherwise

- `reverse : 'a list -> 'a list`

  `(reverse lst)` evaluates to the reverse of the list `lst`

- `zip : 'a list * 'b list -> ('a * 'b) list`

  `(zip (lst1,lst2))` returns the list of pairs of elements of the two lists, truncated to the length of the shorter list

- `drop : int -> 'a list -> 'a list`

  `(drop n lst)` returns the result of dropping the first `n` elements from `lst`; `n` must be non-negative

General Recursion versus Tail Recursion

Consider the two functions defined below

```
let rec fact n =
    if (n = 0) then 1
    else n * (fact (n - 1))

let rec fact' n acc =
    if (n = 0) then acc
    else fact' (n-1) (n * acc)
```

Both can be used to compute the factorial of a number *n*

Both functions are recursive, but there is a difference

> *In the first case control has to return to the caller to complete the computation, not so in the second*

Functions that have the characteristic of the second are said to be *tail recursive*

General Recursion versus Tail Recursion (Contd)

Lets consider the version of the factorial function that is not tail-recursive in more detail

```
let rec fact n =
    if (n = 0) then 1
    else n * (fact (n - 1))
```

Two things have to be remembered each time a recursive call is made

- the current value of `n`
- the remaining computation to be carried out

There will, in fact, be a *stack* of such items to remember

General Recursion versus Tail Recursion (Contd)

Now let us consider the tail-recursive version

```
let rec fact' n acc =
    if (n = 0) then acc
    else fact' (n-1) (n * acc)
```

Since we do not need `n` and `acc` after the recursive call, we can reuse their space for the changed arguments

Further, the structure of control flow is very simple

- we return to the top-level with the value of `acc`, or
- we loop back to execute the body of the function again

More specifically, the computation has the structure of a while loop with fixed storage

For this reason, tail recursion is also referred to as iteration

Tail Recursion and Iteration

The definition of `fact'` can, in fact, be seen as a different way to describe the following iterative program

```
n = n0;
acc = 1;
while (n != 0)) {
    acc = n * acc;
    n = n - 1;
  }
return acc
```

Note here that

- the program variables parameterize the recursive function
- the initialization determines the first recursive call
- changes to the program variables determine the parameters to the recursive call

Accumulators and Tail Recursion

The tail-recursive version of factorial can also be seen as *accumulating* the result in the course of recursion

In particular, the invariant associated with it is the following

`(fact' n acc)` ***evaluates to*** `(n! * acc)`

Here, `acc` functions as the "accumulator"

Once we have decided on this invariant, the rest follows easily

- `n!` can be obtained essentially as `fact' n 1`
- the definition of `fact'` also writes itself
  - `(fact' 1 acc)` is obviously equal to `acc`
  - assuming the invariant, obviously we have
    `(fact' n acc) = (fact' (n-1) (n * acc))`

Of course, figuring out how to accumulate the result in this way requires a careful understanding of the computation

Another Example: Reversing a List

A naive version of a reverse function can be obtained as follows

- the reverse of `[]` is `[]`
- the reverse of `(h::t)` is identical to the reverse of `t` appended to `[h]`

This translates into the following definition in OCaml

```
let rec reverse =
    function
      | [] -> []
      | (h::t) -> append (reverse t) [h]
```

However, this is not tail-recursive and also requires a (costly) use of `append`

Question: Can we do better?

An Iterative Version of List Reverse

We could think of using a while loop for the computation

```
lst1 = lst;
lst2 = [];
while (lst1 != []) {
  let (h::t) = lst1 in
    { lst2 = h :: lst2;
      lst1 = t;
    }
}
return lst2;
```

Translating this into a (tail) recursive program

- the function will have two parameters
- the body of the loop determines how the parameters are modified for the recursive call
- the initialization determines the first recursive call

Iterative Reverse as a Tail-Recursive Function

The iterative code translates naturally into the following definition of `rev`

```
let rec rev lst1 lst2 =
   match lst1 with
   | [] -> lst2
   | (h::t) -> rev t (h::lst2)
```

To reverse a list `lst` we would invoke `(rev lst [])`

Tail-Recursive Reverse Visualized as Accumulation

The computation actually accumulates the reverse as we traverse the list

In particular, think of `rev` as satisfying the following invariant

> `(rev lst1 lst2)` *evaluates to the reverse of* `lst1` *appended to* `lst2`

Here `lst2` is the "accumulated" reverse

Given this invariant, the rest seems obvious

- the reverse of `lst` is `(rev lst [])`
- the definition of `rev` also has the right structure
  - `(rev [] lst)` is obviously `lst`
  - assuming the invariant, we obviously have `(rev (h::t) lst) = rev t (h::lst)`

Nested Function Definitions

The functions `rev` and `fact'` are needed only to define the main functions

We can hide them from the top-level by using nested lets

```
let reverse' lst =
   let rec rev lst1 lst2 =
      match lst1 with
      | [] -> lst2
      | (h::t) -> rev t (h::lst2)
   in rev lst []

let fact n =
  let rec fact' n acc =
       if (n = 0) then acc
       else fact' (n-1) (n * acc)
  in fact' n 1
```

The Unit Type

Ocaml has a basic type called `unit` with `()` as its sole element

This type gets used in two ways in OCaml

- When the value to be returned is irrelevant

  E.g. when an expression is evaluated only for its side effects

- To control when an expression is evaluated

  Ocaml evaluates `<exp>` eagerly in the let definition

  `let <name> = <exp>`

  To control the evaluation, we can use instead

  `let <name> () = <exp>`

  and then invoke `(<name> ())` to force evaluation

We will consider both uses in detail later in the course