Formal Languages and Compilers - Exercises
Lecture 1
Introduction

Michele Caceffo
michele.caceffo@unitn.it

13/03/2012

Outline

Organization

What you need to know:

- All the materials will be on `http://sites.google.com/site/caceffoflc2012/`
- The O'CaML source code (or binaries) is available at `http://caml.inria.fr/`

What you should look at:

- O'CaML manual (which you can find on the O'CaML website)
- "Compilers: Principles, Techniques and Tools" by Aho, Lam, Sethi and Ullman

Organization

What you need to know:

- All the materials will be on `http://sites.google.com/site/caceffoflc2012/`
- The O'CaML source code (or binaries) is available at `http://caml.inria.fr/`

What you should look at:

- O'CaML manual (which you can find on the O'CaML website)
- "Compilers: Principles, Techniques and Tools" by Aho, Lam, Sethi and Ullman

Outline

How to use O'CaML

Interpreter

- Run it with `ocaml`
- Close it with quit ;; (pay attention to the two semicolons)

Compiler(s)

- `ocamlc` produces bytecode output (like Java)
- `ocamlopt` produces compiles directly in machine language

Compilation of a module

- `ocamlc -c <module>.ml` produces `<module>.cmo`
- `ocamlc -c <module_1>.cmo` ... `<module_n>.cmo` links together different modules

How to use O'CaML

Interpreter

- Run it with `ocaml`
- Close it with quit ;; (pay attention to the two semicolons)

Compiler(s)

- `ocamlc` produces bytecode output (like Java)
- `ocamlopt` produces compiles directly in machine language

Compilation of a module

- `ocamlc -c <module>.ml` produces `<module>.cmo`
- `ocamlc -c <module_1>.cmo ... <module_n>.cmo` links together different modules

How to use O'CaML

Interpreter

- Run it with `ocaml`
- Close it with quit ;; (pay attention to the two semicolons)

Compiler(s)

- `ocamlc` produces bytecode output (like Java)
- `ocamlopt` produces compiles directly in machine language

Compilation of a module

- `ocamlc -c <module>.ml` produces `<module>.cmo`
- `ocamlc -c <module_1>.cmo ... <module_n>.cmo` links together different modules

Characteristics of O'CaML

- It's a functional language
- Functions are first-class objects, which means that they can be used as an argument of another function
- Static type checking (types are checked at compile-time). You will hear:

  "If you manage to compile it,
  then it will work for sure!"

  This does not mean it will work the way you expect, that depends on you!
- Static scoping (the values of the variables are static at compile-time)

Characteristics of O'CaML

- It's a functional language
- Functions are first-class objects, which means that they can be used as an argument of another function
- Static type checking (types are checked at compile-time). You will hear:

  "If you manage to compile it,
  then it will work for sure!"

  This does not mean it will work the way you expect, that depends on you!
- Static scoping (the values of the variables are static at compile-time)

Characteristics of O'CaML

- It's a functional language
- Functions are first-class objects, which means that they can be used as an argument of another function
- Static type checking (types are checked at compile-time). You will hear:

  "If you manage to compile it,
  then it will work for sure!"

  This does not mean it will work the way you expect, that depends on you!
- Static scoping (the values of the variables are static at compile-time)

Characteristics of O'CaML

- It's a functional language
- Functions are first-class objects, which means that they can be used as an argument of another function
- Static type checking (types are checked at compile-time). You will hear:

  "If you manage to compile it,
  then it will work for sure!"

  This does not mean it will work the way you expect, that depends on you!
- Static scoping (the values of the variables are static at compile-time)

Characteristics of O'CaML - cont'd

- Type polymorphism
- Constructors of new types
- The module system
- Simple types, like int, float, char, string, bool, ect.
- Built-in simple datatypes as lists, tuples and records

Characteristics of O'CaML - cont'd

- Type polymorphism
- Constructors of new types
- The module system
- Simple types, like int, float, char, string, bool, ect.
- Built-in simple datatypes as lists, tuples and records

Characteristics of O'CaML - cont'd

- Type polymorphism
- Constructors of new types
- The module system
- Simple types, like int, float, char, string, bool, ect.
- Built-in simple datatypes as lists, tuples and records

Characteristics of O'CaML - cont'd

- Type polymorphism
- Constructors of new types
- The module system
- Simple types, like int, float, char, string, bool, ect.
- Built-in simple datatypes as lists, tuples and records

Simple types issues: int and float

int...

- int are integer numbers, with operations:
  - arithmetic: $\{+, -, *, /, \texttt{succ}, \texttt{pred}, \texttt{mod}\}$
  - relational: $\{<, >, <=, >=, =, <>\}$

...and float

- float are numbers in floating-point representation with operators:
  - arithmetic: $\{+., -., *., /., **, \texttt{sqrt}\}$ (notice the '.')
  - relational: $\{<, >, <=, >=, =, <>\}$

We can convert float to int using float_to_int , and int to float using int_to_float

Simple types issues: `int` and `float`

`int`...

- `int` are integer numbers, with operations:
  - arithmetic: $\{+, -, *, /, \texttt{succ}, \texttt{pred}, \texttt{mod}\}$
  - relational: $\{<, >, <=, >=, =, <>\}$

...and `float`

- `float` are numbers in floating-point representation with operators:
  - arithmetic: $\{+., -., *., /., **, \texttt{sqrt}\}$ (notice the '.')
  - relational: $\{<, >, <=, >=, =, <>\}$

We can convert float to int using float_to_int , and int to float using int_to_float

Other simple types: bool, char, unit...

- `bool` = {true, false}
- `char` representes the ASCII characters (`code`, `chr`)
- `unit` is a dummy type, with value (). It's similar to `void` in Java or other languages.
- `string` is a sequence of characters
  - Concatenation: s1 ^ s2
  - Pointing to *i*-th character: s.[i]
  - Module String: length, contains, uppercase...
  - Conversions: string_to_int , float_to_string ...

More structured types

Tuple

- Tuple is a fixed-length list, but the fields may be of differing type: (" hi ", (a, **false**), 3, 4.29)
- Operators are applied element by element:
  - (1, 2, 3) < (4, 5, 6);; results in **true**
  - (9, 3, 4) < (7, 8, 9);; results in **false**

Array

- Array is a fixed-length list, but the fields have to be of the *same* type: [| 1; 2; 3; 4 |];;
- [| 1; 2; 3; 4 |].(2);; will result in... 3

More structured types

`Tuple`

- `Tuple` is a fixed-length list, but the fields may be of differing type: (" hi ", (a, **false**), 3, 4.29)
- Operators are applied element by element:
  - (1, 2, 3) < (4, 5, 6);; results in **true**
  - (9, 3, 4) < (7, 8, 9);; results in **false**

`Array`

- `Array` is a fixed-length list, but the fields have to be of the *same* type: [| 1; 2; 3; 4 |];;
- [| 1; 2; 3; 4 |].(2);; will result in... 3

More structured types

Tuple

- `Tuple` is a fixed-length list, but the fields may be of differing type: (" hi ", (a, **false**), 3, 4.29)
- Operators are applied element by element:
  - (1, 2, 3) < (4, 5, 6);; results in **true**
  - (9, 3, 4) < (7, 8, 9);; results in **false**

Array

- `Array` is a fixed-length list, but the fields have to be of the *same* type: [| 1; 2; 3; 4 |];;
- [| 1; 2; 3; 4 |].(2);; will result in... 3

List, your best friend

Record

- Record is a sequence of elements of particular type:

```
type address = {
    name: string;
    street: string;
    number: int
};;
let jedi = {
    name = ''Yoda'';
    street = ''Dagobah swamp'';
    number = 1
};;
jedi.street;;
```

- will result in "Dagobah swamp"

List, your best friend

List

- List is a sequence of objects of the same type:
  [1.5; 2.0; 3.2]
- Operators are applied like for tuples, so
  [1; 2; 3] < [4; 5; 6];; results in **true**
- Constructors:
  - Empty list []
  - Add an element to a list with ::
    4 :: [1; 2];; results in [4; 1; 2]
  - List concatenation (l1 @ l2)
    [1; 2] @ [3; 4];; results in [1; 2; 3; 4]

List, your best friend

List

- List is a sequence of objects of the same type:
  [1.5; 2.0; 3.2]
- Operators are applied like for tuples, so
  [1; 2; 3] < [4; 5; 6];; results in **true**
- Constructors:
  - Empty list []
  - Add an element to a list with ::
    4 :: [1; 2];; results in [4; 1; 2]
  - List concatenation (l1 @ l2)
    [1; 2] @ [3; 4];; results in [1; 2; 3; 4]

Outline

O'CaML syntax - 1

Variables

- Binding: **let** x=5;;
- Parallel binding: **let** x=5 **and** y=4;;
- Local binding: **let** x=4 **in** x∗2;;
- Remember that the binding is static (compile-time):
  let x=3 **in let** x=2 **in** x−1;; results in 1

Pattern matching

Matches the data composed using constructors:

- **let** couple = (a , 5.3);;
- **let** (first , second)= couple;;
- Substitutes first with a and second with 5.3

O'CaML syntax - 1

Variables

- Binding: **let** x=5;;
- Parallel binding: **let** x=5 **and** y=4;;
- Local binding: **let** x=4 **in** x∗2;;
- Remember that the binding is static (compile-time):
  let x=3 **in** **let** x=2 **in** x−1;; results in 1

Pattern matching

Matches the data composed using constructors:

- **let** couple = (a , 5.3);;
- **let** (first , second)= couple;;
- Substitutes first with a and second with 5.3

O'CaML syntax - 2

Using constructors

It's possible to use [] and :: to match with lists

```
let list = [1; 2; 3];;
let head::tail = list;;
```

Results in head = 1 and tail = [2; 3]

Wildcard

_ is an anonymous pattern that matches everything:

```
let head::_ = list;;
```

Results in head = 1

O'CaML syntax - 2

Using constructors

It's possible to use [] and :: to match with lists

```
let list = [1; 2; 3];;
let head::tail = list;;
```

Results in head = 1 and tail = [2; 3]

Wildcard

_ is an anonymous pattern that matches everything:

```
let head::_ = list;;
```

Results in head = 1

O'CaML syntax - 3

Definition of function

```
let f = fun x -> x*2;;
let f x = x*2;;
val f: int -> int = <fun>
```

Curry

Transforming a function which takes multiple arguments such that it can be called as a chain of functions with a single argument.

```
let f = fun (x, y) -> x + y;;
f: (int * int) -> int
let f = fun x y -> x + y;;
f: int -> int -> int
```

O'CaML syntax - 3

Definition of function

```
let f = fun x -> x*2;;
let f x = x*2;;
val f: int -> int = <fun>
```

Curry

Transforming a function which takes multiple arguments such that it can be called as a chain of functions with a single argument.

```
let f = fun (x, y) -> x + y;;
f: (int * int) -> int
let f = fun x y -> x + y;;
f: int -> int -> int
```

Functions - 1

Pattern matching over functions

```
let rec factorial = function
    0 -> 1
  | n -> n * factorial(n-1);;
```

Substitute a function as a result

```
let mult x y = x*y;;
val mult: int -> int -> int = <fun>
let double = mult 2;;
val double: int -> int = <fun>
```

Functions - 1

Pattern matching over functions

```
let rec factorial = function
    0 -> 1
  | n -> n * factorial(n-1);;
```

Substitute a function as a result

```
let mult x y = x*y;;
val mult: int -> int -> int = <fun>
let double = mult 2;;
val double: int -> int = <fun>
```

Functions - 2

Taking another function as an argument

```
let rec map f list = match list with
    [] -> []
  | head::tail -> f head::map f tail;;
val map: (a -> b) -> a list -> b list = <fun>
```

map double [1; 2; 3];; will result in [2; 4; 6]

What does the map function do?

Given a function f and a list $[i_1, i_2, \dots, i_n]$:

$$\text{map}(f, [i_1, i_2, \dots, i_n]) \longrightarrow [f(i_1), f(i_2), \dots, f(i_n)]$$

Functions - 2

Taking another function as an argument

```
let rec map f list = match list with
    [] -> []
  | head::tail -> f head::map f tail;;
val map: (a -> b) -> a list -> b list = <fun>
```

map double [1; 2; 3];; will result in [2; 4; 6]

What does the map function do?

Given a function f and a list $[i_1, i_2, \ldots, i_n]$:

$$\text{map}(f, [i_1, i_2, \ldots, i_n]) \longrightarrow [f(i_1), f(i_2), \ldots, f(i_n)]$$

Polymorphism

Variables

Variables whose type can't be inferred, have types 'a, 'b

```
let id x = x;;
```

results in

```
val id: 'a -> 'a = <fun>
```

Functions

Functions can be polymorphic in their arguments' and return types

```
let comp f g x = f(g(x));;
val comp:('a ->'b)->('c ->'a)->'c->'b =<fun>
```

Polymorphism

Variables

Variables whose type can't be inferred, have types 'a, 'b

```
let id x = x;;
```

results in

```
val id: 'a -> 'a = <fun>
```

Functions

Functions can be polymorphic in their arguments' and return types

```
let comp f g x = f(g(x));;
val comp:('a ->'b)->('c ->'a)->'c->'b =<fun>
```

Outline

Type declarations

Simple type declaration

```
type color = Red | Blue | Green | Yellow;;
```

Using type constructors

```
type money = Nothing
           | USDollars of float
           | Euro of float
let balance = function
      Nothing -> 0.0
    | USDollars (dollars) -> dollars
    | Euro(euros) -> euros
```

Type declarations

Simple type declaration

```
type color = Red | Blue | Green | Yellow;;
```

Using type constructors

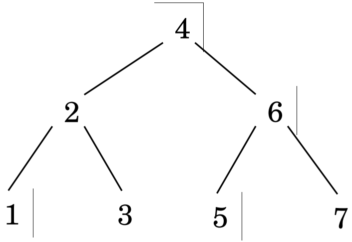
```
type money = Nothing
           | USDollars of float
           | Euro of float
let balance = function
      Nothing -> 0.0
    | USDollars (dollars) -> dollars
    | Euro(euros) -> euros
```

Recursive Data Type: tree

```
type 'a tree = Leaf of 'a
             | Tree of 'a * 'a tree * 'a tree
```

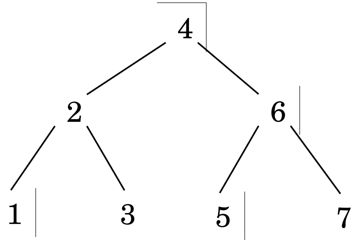
Example

```
let mytree = Tree (4,
Tree(2, Leaf(1), Leaf(3)),
Tree(6, Leaf(5), Leaf(7)));;
```



Recursive Data Type: tree

```
type 'a tree = Leaf of 'a
             | Tree of 'a * 'a tree * 'a tree
```



Example

```
let mytree = Tree (4,
Tree(2, Leaf(1), Leaf(3)),
Tree(6, Leaf(5), Leaf(7)));;
```

Exceptions - 1

Predefined exceptions

Division_by_zero , Out_of_memory,Invalid_argument, ...

User-defined exceptions

```
exception Empty_list of string;;
let head = fun r -> match r with
            [] -> raise (Empty_list("Empty!"))
          | hd::tl -> hd;;

head [];;
Exception: Empty_list "Empty!".
```

Exceptions - 1

Predefined exceptions

Division_by_zero , Out_of_memory,Invalid_argument, . . .

User-defined exceptions

```
exception Empty_list of string;;
let head = fun r -> match r with
              [] -> raise (Empty_list("Empty!"))
            | hd::tl -> hd;;

head [];;
Exception: Empty_list "Empty!".
```

Exceptions - 2

Handling exceptions

O'CaML we can use the **try** . . . **with** construct

```
try
    dangerous expression
with
    exception1 -> action1
  | exception2 -> action2
  ...
  | exceptionN -> actionN
  | _ -> lastChance
```

Outline

Abstract Data Types (ADT)

Structure

- Interface: declarations of data types and functions (like header files in C and interfaces in Java)
- Implementation: `.c` files in C or Java classes

Realization

- Compilation unit (1 file $\longleftrightarrow$ 1 module)
- Module system (1 file $\longleftrightarrow$ 1 or more modules)

Abstract Data Types (ADT)

Structure

- Interface: declarations of data types and functions (like header files in C and interfaces in Java)
- Implementation: `.c` files in C or Java classes

Realization

- Compilation unit (1 file $\longleftrightarrow$ 1 module)
- Module system (1 file $\longleftrightarrow$ 1 or more modules)

Compilation Unit - Example

Interface - myset.mli

```
type 'a set
val emptySet : 'a set
val insert : 'a -> 'a set -> 'a set
val member : 'a -> 'a set -> bool
```

Implementation - myset.ml

```
type 'a set = Null | Ins of 'a * 'a set
let emptySet = Null
let insert x = fun s -> Ins (x, s)
let rec member x = function Null -> false
        | Ins(v, s) -> x=v || member x s
```

Compilation Unit - Example

Interface - myset.mli

```
type 'a set
val emptySet : 'a set
val insert : 'a -> 'a set -> 'a set
val member : 'a -> 'a set -> bool
```

Implementation - myset.ml

```
type 'a set = Null | Ins of 'a * 'a set
let emptySet = Null
let insert x = fun s -> Ins (x, s)
let rec member x = function Null -> false
        | Ins(v, s) -> x=v || member x s
```

Module system

Module system

- Signature = interface
- Structure = implementation

Correspondence between the signature and structure

- one structure for many signatures: changes the visible functionality depending on the needs
- one signature for many structures: changes the implementation without impact on the elements of the module

Module system

Module system

- Signature = interface
- Structure = implementation

Correspondence between the signature and structure

- one structure for many signatures: changes the visible functionality depending on the needs
- one signature for many structures: changes the implementation without impact on the elements of the module

Module system example - 1

Signature

```
module type mysetSig = sig
  type a set
  val emptySet : a set
  val insert : a -> a set -> a set
  val member : a -> a set -> bool
end;;
```

Module system example - 2

Structure

```
module Set: mysetSig = struct
  type 'a set = Null | Ins of 'a * 'a set
  let emptySet = Null
  let insert x = function s -> Ins(x, s)
  let rec member x = function
        Null -> false
      | Ins (v, s) -> x=v || member x s
end;;
```