Formal Languages and Compilers

Exercises

Nataliia Bielova

http://disi.unitn.it/~bielova/flc/index.html

Organization

- All the material on the website

  http://disi.unitn.it/~bielova/flc/index.html

- Didattica online: Esse3
- Caml language http://caml.inria.fr/
- Suggested literature:
  - OCaml manual + code of interpreter/compilator
  - The "Dragon" book – 1st or 2nd edition
    - "Compilers: Principles, Techniques, and Tools" by A.V. Aho, M.S. Lam, R. Sethi and J.D. Ullman
- Contact: bielova@disi.unitn.it

Revisiting OCaml

Lection 1

How to run OCaml

- Run the interpreter with
  - ocaml
- Exit the interpreter:
  - # quit;;
- Compilers:
  - ocamlc compiles in bytecode
  - ocamlopt compiles in machine code
- Compilation of a single module
  - ocamlc –c <fileName>.ml
  - Produces <fileName>.cmo
- Linking different modules .cmo:
  - ocamlc –o file1.cmo … fileN.cmo

Characteristics of OCaml

- It's a *functional* language
- Functions are "first-class" objects (as in mathematics, a function can be used as an argument of another function)
- Static type checking (the types are checked at *compile-time* "If you manage to compile it, then it will work for sure!")
- Static scoping (the values of the variables are static at *compile-time*)

Characteristics of OCaml (2)

- Type polymorphism
- Constructors of type
- The module system
- Simple types: int, float, char, string, bool, …
- Built-in simple datatypes: list, tuple, record, …

Characteristics of OCaml (2)

- Type polymorphism
- Constructors of type
- The module system
- Simple types: int, float, char, string, bool, …
- Built-in simple datatypes: list, tuple, record, …

Remember!

Since the language is strongly typed, the int and float types are not compatible:
1.4 + 5 gives an error!

int and float, what's wrong with them?

- int are integer numbers together with operations:
  - arithmetic: + - * / succ pred mod
  - relational: < > <= >= = <>
- float are numbers in floating-point representation with operators:
  - arithmetic: +. -. *. /. ** sqrt (notice the ".")
  - relational: < > <= >= = <>
- Conversions between float and int: int_to_float and float_to_int

bool, char, unit

- bool = {true, false}
- char representes the ASCII characters
- Useful functions in module Char:
  - code returns the ASCII code of the argument
  - chr returns the character with the given ASCII code
- unit is a “not so interesting type”. Its value is () and it’s similar to void in Java & C.

string

- Is a sequence of characters
- Operators:
  - Concatenation s1 ^ s2
  - pointing to one character s.[index]
  - module String: length, contains, uppercase, …
- Conversions: string_to_int, float_to_string, …

Tuple

- Tuple is a *fixed-length* list, but the fields may be of *differing type*

  (2, “hi”, (‘a’ , false))

- Operators are applied element by element:

  (1, 2, 3) < (4, 5, 6);; results in true

List

- List is a sequence of objects of the *same type*:

  [1.5; 2.0; 3.2] (notice the ";")

- Operators are applied like for tuples:

  [1; 2; 3] < [4; 5; 6];; results in true

- Constructors:
  - [] empty list
  - :: add an element to a list:

    4 :: [1; 2];; results in [4; 1; 2]
  - (l1 @ l2) concatenation of the lists:

    [1; 2] @ [3; 4];; results in [1; 2; 3; 4]

Array and Record

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

Array and Record

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

Array and Record

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

- Record is a sequence of elements of *particular type*:
- type address = {name: string; street: string; number: int} ;;

  let friend = {name = "Bart Simpson"; street = "15th avenue";
  number = 1};;

  friend.street;; results in "15th avenue"

Variables

- Binding

  let x=5;;

- Parallel binding

  let x=5 and y=4;;

- Local binding

  let x=4 in x*2;;

- Remember: the binding is *static*

  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

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

- It’ s possible to use [] and :: with the lists

  let list = [1; 2; 3];;

  let head::tail = list;;

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

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

- It’ s possible to use [] and :: with the lists

  let list = [1; 2; 3];;

  let head::tail = list;;

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

- _ is anonymous pattern that matches everything:

  let head::_ = list;;

  results in head = 1

Functions

- Definition

  let f = fun x -> x*2;; let f x = x*2;;

  val f: int -> int = <fun>

Functions

- Definition

  let f = fun x -> x*2;; let f x = x*2;;

  val f: int -> int = <fun>

- Functions always have only one argument
  - let f = fun (x, y) -> x + y;;
    f: (int * int) -> int function *uncurry*
  - let f = fun x y -> x + y;;
    f: int -> int -> int function *curry*

Functions

- Definition

  let f = fun x -> x*2;; let f x = x*2;;

  val f: int -> int = <fun>

- Functions always have only one argument
  - let f = fun (x, y) -> x + y;;
    f: (int * int) -> int function *uncurry*
  - let f = fun x y -> x + y;;
    f: int -> int -> int function *curry*

- Pattern matching over the functions

  let rec factorial = function

  0 -> 1

| n -> n * factorial(n-1);;

Higher-order functions

- Substitute a function as a result

  let mult x y = x*y;;

  let double = mult 2;;

  val double: int -> int = <fun>

Higher-order functions

- Substitute a function as a result

  let mult x y = x*y;;

  let double = mult 2;;

  val double: int -> int = <fun>

- Taking another function as an argument

  let rec map f list = match list with

   [] -> []

  | head::tail -> f head:: map f tail;;

Higher-order functions

- Substitute a function as a result

  let mult x y = x*y;;

  let double = mult 2;;

  val double: int -> int = <fun>

- 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>

Higher-order functions

- Substitute a function as a result

  let mult x y = x*y;;

  let double = mult 2;;

  val double: int -> int = <fun>

- 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];; results in [2; 4; 6]

Polymorphism

- Variables of the type ‘a, ‘b, ‘c, …

  let id x = x;; results in val id: ‘a -> ‘a = <fun>

- Polymorphic function

  let comp f g x = f(g(x));;

  val comp: (‘a -> ‘b) -> (‘c ->’ a) -> c’ -> ‘b = <fun>

Polymorphism

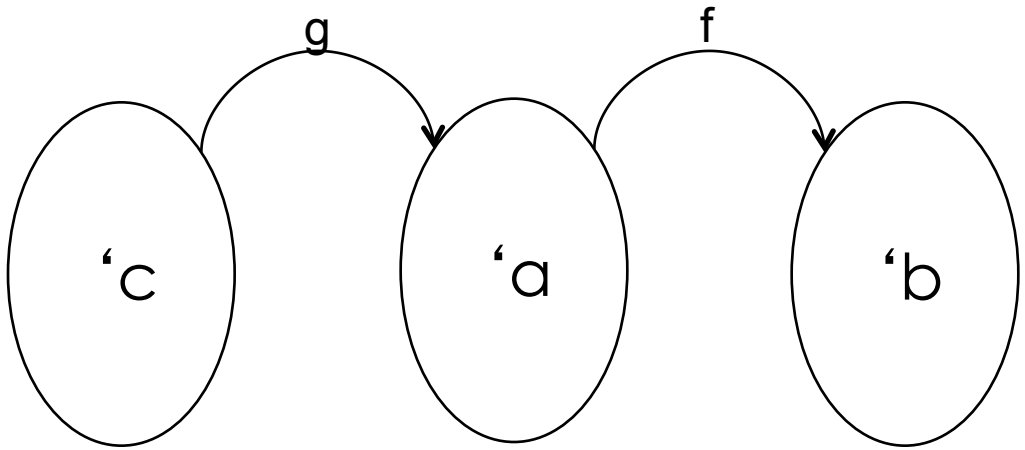
- Variables of the type ‘a, ‘b, ‘c, …

  let id x = x;; results in val id: ‘a -> ‘a = <fun>

- Polymorphic function

  let comp f g x = f(g(x));;

  val comp: (‘a -> ‘b) -> (‘c ->’ a) -> c’ -> ‘b = <fun>



Type declarations

- Simple type declaration

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

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

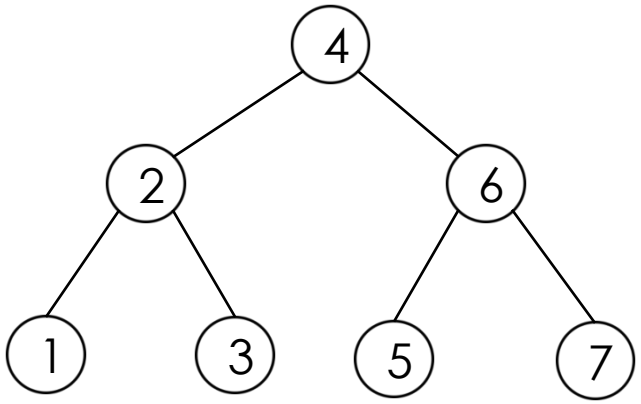
Recursive data type: tree

- type 'a tree =

  Leaf of 'a

  | Tree of 'a * 'a tree * 'a tree

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



Exceptions

- Predefined exceptions: Division_by_zero, Out_of_memory, Invalid_argument, …

Exceptions

- Predefined exceptions: Division_by_zero, Out_of_memory, Invalid_argument, …
- User-defined exceptions:

```
exception Empty_list of string;;
```

Exceptions

- Predefined exceptions: Division_by_zero, Out_of_memory, Invalid_argument, …
- User-defined exceptions:

```
exception Empty_list of string;;

let head = fun l -> match l with

    [] -> raise (Empty_list("Empty!"))

    | hd::tl -> hd;;
```

Exceptions

- Predefined exceptions: Division_by_zero, Out_of_memory, Invalid_argument, …
- User-defined exceptions:

```
exception Empty_list of string;;

let head = fun l -> match l with

    [] -> raise (Empty_list("Empty!"))

    | hd::tl -> hd;;


head [];;
```

Exceptions

- Predefined exceptions: Division_by_zero, Out_of_memory, Invalid_argument, …
- User-defined exceptions:

```
exception Empty_list of string;;

let head = fun l -> match l with

    [] -> raise (Empty_list("Empty!"))

    | hd::tl -> hd;;


head [];;

Exception: Empty_list "Empty!".
```

try … with

- Handle the excetions

  try *dangerous expression*

  with exception1-> action1

  | exception2 -> action2

…

  | exceptionN -> actionN

  | _ -> lastChance

Abstract Data Types (ADT)

- Abstract Data Types:
  - Interface: declarations of data types and functions (in C: file .h, in Java: interface)
  - Implementation: … (In C: file .c, in Java: class)
- It's realized with
  - Compilation unit (1 file ↔ 1 module)
  - Module system (1 file ↔ 1 or more modules)

Compilation unit

- Interface

  File: .mli

  Content: what is visible outside of the module

- Implementation

  File .ml

  Content: implementation of the module ☺

Compilation unit - interface

- File myset.mli:

```
type 'a set

val emptySet : 'a set

val member : 'a -> 'a set -> bool

val insert : 'a -> 'a set -> 'a set
```

Compilation unit - interface

- File myset.mli:

type 'a *set – abstract type, hence is not used directly*

val emptySet : 'a set

val member : 'a -> 'a set -> bool

val insert : 'a -> 'a set -> 'a set

Compilation unit - implementation

- File 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 - use

```
$ ocamlc –c myset.mli

$ ocamlc –c myset.ml

$ ocaml

# #load “myset.cmo”;;

# open Myset;;

# let s1 = emptySet;;

val s1 : ‘a Myset.set -> <abstr>

# let s1 = insert 3 s1;;

val s1: int  Myset.set = <abstr>
```


…

Module system

- Signature = interface
- Structure = implementation

Module system

- Signature = interface
- Structure = implementation
- Correspondence between the signature and structure:
  - 1 structure → many signatures: changes the visible functionality depending on the needs
  - 1 signature → many structures: changes the implementation without impact on the elements of the module

Module system - 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 - 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;;
```

Try the exercises

- http://disi.unitn.it/~bielova/flc/index.html