Lectures 3-4

Functional languages

Iztok Savnik, FAMNIT

March, 2023.

Literature

John Mitchell, Concepts in Programming Languages, Cambridge Univ Press, 2003 (Chapter 7)

Michael L. Scott, Programming Language Pragmatics (3rd ed.), Elsevier, 2009 (Chapters 10)

Outline

- History
- Values
- `let` statement, functions
- Blocks, name spaces
- Recursion
- Polymorphism
- Pattern-matching
- Types in functional languages
- Higher-order functions
- Trees

Models of computation

- In 1930s many mathematicans were dealing with the formal notion of algorithm
  - Turing, Church, Kleene, Post, ...
  - Formalizations of »effective procedure«
- Mathematical models
  - Lambda calculus, General recursive functions. Combinatory logic. Abstract rewriting systems
- State machine models
  - Finite state machines. Pushdown automata. Random access machines. Turing machines
- Concurrent models
  - Cellular automaton. Kahn process networks. Petri nets. Synchronous Data Flow. Interaction nets. Actor model

Models of computation

- Church's thesis says that all such formalizations are equally powerful
- Foundations of imperative and functional languages
  - Turing machine
  - Lambda calculus

Functional languages

- In recent years functional languages have become increasingly more popular
  - Scientific as well as bussiness applications
- Functional languages have a great deal in common with imperative and object-oriented relatives
  - Names, scoping, expressons, types, recursion, ...
- We will learn common concepts of PL in different paradigms
  - Functions, parameter passing, blocks, ...
- Specific concepts of particular computation models will also be presented
  - Iteration, recursion, OO paradigm, ...

Outline

- History
- Values
- `let` statement, functions
- Blocks, name spaces
- Recursion
- Polymorphism
- Pattern-matching
- Types in functional languages
- Higher-order functions
- Trees

Values

- Atomic types
  - integer, real, boolean, string
  - Simple value is an instance of atomic type
- Functional languages use values
- Imperative languages use variables
- Ocaml includes atomic types:
  - Integer, float, char, string
  - Operations are bound to particular types
  - Derivation of expression type is easier

```
# 7;;
- : int = 7
# 5.3;;
- : float = 5.3
# 'c';;
- : char = 'c'
# "spring";;
- : string = "spring"
```

Lists

- Values of some type can be stored into lists
  - List is either empty [], or, it includes valus of fixed type
- List has a head and a tail:
  - 1 is head and [2; 3] a tail
  - [1; 2; 3] ≡ 1::[2; 3] ≡ 1::2::3::[]
  - Operator :: corresponds to Lisp cons operator
- Two lists can be concatenated using operator append @
  - [1]@[2; 3] ≡ [1; 2; 3]
- Implementation details & other aspects
  - Lectures on (1) Imperative lang. and (2) Types

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

Products

- Products are defined as in mathematics
  - Interpretation of a product type is the Cartesian product of the interpretations of types that comprise product
  - Instances of products are pairs, triples, n-tuples

```
# ( 65 , ’B’ , "ascii") ;;
- : int * char * string = 65, ’B’, "ascii"
```

```
# ( 12 , "October") ;;
- : int * string = 12, "October"
```

```
# fst;;
- : ’a * ’b -> ’a = <fun>
# fst ( "October", 12 ) ;;
- : string = "October"
```

Outline

- History
- Values
- `let` statement, functions
- Blocks, name spaces
- Recursion
- Polymorphism
- Pattern-matching
- Types in functional languages
- Higher-order functions
- Trees

Bindings

- A binding is an association between two things, such as a name and the thing it names
- let statement binds value with its name
  - let x = M in N ≡ (λx.N)M
- Value can be defined globally or locally
  - Global definition is accessible in global context
  - Local definition is seen solely in local context

```
let name₁= expr₁
...
and nameₙ= exprₙ
in expr
```

```
let name₁= expr₁
...
and nameₙ= exprₙ
```

Statement `let`: Examples

```
# let a = 3.0 and b = 4.0;;
val a : float = 3.
val b : float = 4.
# a;;
- : float = 3.
```

```
# let x = 3 in x * x ;;
- : int = 9
# (let x = 3 in x * x) + 1 ;;
- : int = 10
```

```
# let a = 3.0 and b = 4.0 in sqrt (a*.a +. b*.b);;
- : float = 5.
# a;;
Error: Unbound value a
```

```
# let c = (let a = 3.0 and b = 4.0 in sqrt (a*.a +. b*.b));;
val c : float = 5.
```

Functions

- Function is lambda abstraction λx.M
  - x is a parameter and M is the body of a function
- Function expression is composed of a parameter x and a function body M
  - Function parameter is formal
- Example:
  - λx.x*x
  - (λx.x*x) 5

```
function x -> M
```

```
# function x -> x*x ;;
- : int -> int = <fun>
# (function x -> x * x) 5 ;;
- : int = 25
```

Functions

- Function body can be another function

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

- The result of a function can again be a function

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

```
# (function x -> function y -> 3*x + y) 4 5 ;;
- : int = 17
```

Function

- The arity of function is number of its parameters
  - Functions have single parameters in OCaml
  - This is called Curry form of function

$f\ (g, x) = g(x)$

$f_{curry} = \lambda g.\lambda x.g\ x$

- Single parameter can also be a tuple or a record
  - Curry form of the same function

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

```
# function x -> function y -> 3*x + y;;
- : int -> int -> int = <fun>
# (function x -> function y -> 3*x + y) 4 5;;
- : int = 17
```

Function values

- Function is treated as a value
  - Using `let` statement to define binding between function and name
- Examples:

```
# let succ = function x -> x + 1 ;;
val succ : int -> int = <fun>
# succ 420 ;;
- : int = 421
# let g = function x -> function y -> 2*x + 3*y ;;
val g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
```

Function definition

- Alternative syntax for function definition in Ocaml

let name p_1 p_2 ... = <function-body>

let name = function p_1 -> ... -> function p_n -> <function-body>

- Example:
  - h1(y) = 2 + 3*y
  - h2(x) = 2*x + 6

```
# let s x = x + 1;;
val s : int -> int = <fun>
# let g x y = 2*x + 3*y ;;
val g : int -> int -> int = <fun>
# let h1 = g 1 ;;
val h1 : int -> int = <fun>
# let h2 = function x -> g x 2 ;;
val h2 : int -> int = <fun>
# h2 2 ;;
- : int = 10
```

Outline

- History
- Values
- `let` statement, functions
- Blocks, name spaces
- Recursion
- Polymorphism
- Pattern-matching
- Types in functional languages
- Higher-order functions
- Trees

Blocks

```
{ int x = . . . ;
  { int y = . . . ;
    x = y+2;
    { int x = . . . ;
      ...
     };
  };
};
```

- Most modern programming languages provide some kind of blocks
  - Block is program region that includes begin and end
  - Blocks have local variables
  - Global variable of some block is defined in some encompassing block
- Block are used in all modern PLs
  - C, C++, Java, Scala, Rust, C#, ML, ...
- Local def. of value can hide def. of global value

```
# let a = 1.0;;
- : float = 1
# let a = 3.0 and b = 4.0 in sqrt (a*.a +. b*.b) ;;
- : float = 5
```

Scope and lifetime

- Scope of value (or variable) is program area where it is defined
  - Value defined in some global block is defined in all subsuming blocks
- Lifetime of value (or variable) is duration of definition of value
- In many cases scope implies lifetime
  - Variables defined in a scope are disposed together with the environment of the scope
  - Global declarations in C, C++ can appear locally

Static and dynamic scope

- Let some value be defined outside current block
  - Then variable is global relatively to local block
- Static scope
  - Variables are first searched in local block and then in structurally enclosing blocks
- Dynamic scope
  - Variables are searched by following function calls i.e. blocks where function was invoked
- Static: C, Schema, Scala, Rust, ML, Pascal
  Dynamic: older Lisp, macros in C

```
# let x=1;;
val x : int = 1
# let g z = x+z;;
val g : int -> int = <fun>
# let f y = let x = y+1 in g (y*x);;
val f : int -> int = <fun>
# f(3);;
- : int = 13 (or 16)?
```

Outline

- History
- Values
- `let` statement, functions
- Blocks, name spaces
- Recursion
- Polymorphism
- Pattern-matching
- Types in functional languages
- Higher-order functions
- Trees

Recursion

- Definition of a symbol includes reference to itself
- Recursion was first introduced in Lisp
  - McCarthy advocated to use recursion in Algol
- Lambda abstractions do not have name
  - McCarthy suggestion for naming functions in Lisp

```
(label f (lambda (x) (cond ((eq x 0) 0) (true (+ x (f (- x 1)))))))
```

  - Later they simply declared function using `define`

```
(define f (lambda (x) (cond ((eq x 0) 0) (true (+ x (f (- x 1)))))))
```

Linear recursion

- Recursion that evolves in one direction
  - Recursive call is at the end of function body
- Example: factorial

```
# let rec factorial n -> if n=1 then 1 else n * factorial (n-1);;
val factorial : int -> int = <fun>
```

- Tail recursion
  - Can be converted into iteration

```
fact 6 = 6 * fact 5
       = 6 * ( 5 * fact 4)
       = 6 * ( 5 * (4 * fact 3))
       = 6 * ( 5 * (4 * ( 3 * fact 2)))
       = 6 * ( 5 * (4 * ( 3 * ( 2 * fact 1))))
       = 6 * ( 5 * (4 * ( 3 * ( 2 * 1))))
       = 720
```

Examples

- Function `sigma` computes sum from `1` to n.
  - Linear recursion

```
# let rec sigma x = if x = 0 then 0 else x + sigma (x-1) ;;
val sigma : int -> int = <fun>
# sigma 10 ;;
- : int = 55
```

- Function `even` and `odd` return boolean value
  - Recursion in two cycles

```
# let rec even n = (n<>1) && ((n=0) or (odd (n-1)))
  and odd n = (n<>0) && ((n=1) or (even (n-1)));;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
# even 4 ;;
- : bool = true
# odd 5 ;;
- : bool = true
```

Ackermann function

$$A(0, y) = y+1$$
$$A(x+1, 0) = A(x, 1)$$
$$A(x+1, y+1) = A(x, A(x+1, y))$$

- Ackermann f. is not primitive recursive but decidable (total fun.)
  - Linear recursion with termination condition
  - Growing faster than any primitive recursive f.
  - Decidable functions exist "above" primitive recursive functions.
- Example
  - A(1,1) and A(2,1)

$$\begin{aligned}
A(1, 1) &= A(0, A(1, 0)) \\
&= A(0, A(0, 1)) \\
&= A(0, 2) \\
&= 3 \\
A(2, 1) &= A(1, A(2, 0)) \\
&= A(1, A(1, 1)) \\
&= A(1, 3) \\
&= A(0, A(1, 2)) \\
&= A(0, A(0, A(1, 1))) \\
&= A(0, A(0, 3)) \\
&= A(0, 4) \\
&= 5
\end{aligned}$$

Ackermann function

- Implementation in OCaml
  - Recursive functions can be transl. almost directly to ML
- Function is growing very fast
  - Num. of atoms in the universe
  - Try: ack 4 2;;
    - Notebook with 16GB RAM
    - Combinatorial explosion
    - ack 2 n = 2*n + 3
    - ack 3 n = $2^{(n + 3)} - 3$
    - ack 4 2 = $2^{65536} - 3$
    - ack 4 3 = $2^{2\hat{}65536} - 3$

```
# let rec ack x y =
  if (x == 0) then y+1
  else if (y == 0) then ack (x-1) 1
       else ack (x-1) (ack x (y-1));;
val ack : int -> int -> int = <fun>
# ack 3 5;;
- : int = 253
```

```
# ack 1 2;;
- : int = 4
# ack 2 3;;
- : int = 9
# ack 3 14;;
- : int = 131069
# ack 3 15;;
Stack overflow during evaluation (looping recursion?).
# ack 4 1 ;;
- : int = 65533
# ack 4 2;;
Stack overflow during evaluation (looping recursion?).
#...
```

Recursive functions on lists

- A list is a linear data structure
  - Grows in one direction h::t (head and tail)
  - A tail again has a head and a tail, etc.
- Linear recursion
  - Recursion guided by the structure
  - Pattern:
    - Take the head and use it to compute the task.
    - Recur the action on the tail.
    - On return from the recursive call, take the result and build something that is again returned as the result.
    - Stopping condition is the end of a list.
    - Parameters can be used to assist in the construction of the result
- Can be converted into iteration, in some cases.

Recursive functions on lists

```
# let null l = (l = []) ;;
val null : ’a list -> bool = <fun>

# let rec size l =
if null l then 0
else 1 + (size (List.tl l)) ;;
val size : ’a list -> int = <fun>

# let rec reverse l =
if null l then []
else (reverse (List.tl l)) @ [(List.hd l)];;
val reverse : ’a list -> ’a list = <fun>
```

Recursive functions on lists

```
# let rec member x l =
if null l then false
else if x = List.hd(l) then true
else member x (List.tl l);;
val member : 'a -> 'a list -> bool = <fun>
# member 3 [2;3;1];;
- : bool = true

# let rec inter(xs, ys) =
if null xs then []
else let x = List.hd xs
        in if (member x ys) then x :: inter(List.tl xs, ys)
            else inter(List.tl xs, ys);;
val inter : 'a list * 'a list -> 'a list = <fun>
# inter ([1;2;3],[4;2]);;
- : int list = [2]
```

Outline

- History
- Values
- `let` statement, functions
- Blocks, name spaces
- Recursion
- Polymorphism
- Pattern-matching
- Types in functional languages
- Higher-order functions
- Trees

Polymorphism

- Polymorphism (Greek, »many shapes«)
- Parametric polymorphism
  - Function can have "many shapes"
  - Types of parameters are variables
- Type variables
  - Variable that stands for any type
  - 'a, 'b, 'c, ...
- A form of genericity

```
# let make_pair a b = (a,b) ;;
val make_pair : 'a -> 'b -> 'a * 'b = <fun>
# let p = make_pair "paper" 451 ;;
val p : string * int = "paper", 451
# let a = make_pair 'B' 65 ;;
val a : char * int = 'B', 65
# fst p ;;
- : string = "paper"
# fst a ;;
- : char = 'B'
```

Examples: Polymorphic functions

```
# let app = function f -> function x -> f x ;;
val app : ('a -> 'b) -> 'a -> 'b = <fun>
```

- Function application
  - Any function f of type 'a->'b can be passed as parameter

```
# app odd 2;;
- : bool = false
# let id x = x ;;
val id : 'a -> 'a = <fun>
# app id 1 ;;
- : int = 1
```

- Function composition
  - Any functions f and g of type 'a->'b and 'c->'a can be passed as parameters

```
# let compose f g x = f (g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let add1 x = x+1 and mul5 x = x*5 in compose mul5 add1 9 ;;
- : int = 50
```

Examples:
Polymorphic functions on lists

```
# let rec member x l =
if null l then false
else if x = List.hd(l) then true
else member x (List.tl l);;
val member : ’a -> ’a list -> bool = <fun>
# member 3 [2;3;1];;
- : bool = true
# member 'a' ['c';'d';'e';'a'];;
- : bool = true
# member 1 ['c';'d';'e';'a'];;
Error: This expression has type char but an
expression was expected of type int
```

```
# let rec append l1 l2 =
if null l1 then l2
else (List.hd l1)::append (List.tl l1) l2    ;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
# append ["2";"ab"] ["3";"c";"d"];;
- : string list = ["2"; "ab"; "3"; "c"; "d"]
```

Outline

- History
- Values
- `let` statement, functions
- Blocks, name spaces
- Recursion
- Polymorphism
- Pattern-matching
- Types in functional languages
- Higher-order functions
- Trees

Pattern matching

- A special feature of languages of ML family
  - Haskell, Erlang, SML, ...
  - Close to Prolog unification
  - Very complex `case` statement
- Declarative language construct !
  - Logic based languages?
  - Not functional and not imperative?
- Allows simple access to the components of complex data structures
- Functions can be defined by cases
  - Pattern matching over an argument

Pattern matching

- The patterns for the data structures characteristic for the functional languages are presented
  - Lists, products and unions
- The patterns used for the data structures of the imperative languages are presented later
  - ... in the frame of the imperative prog. Languages
  - Records and arrays.

Pattern matching

- Pattern definition
  - Structure comprised of tuples, records, unions and lists including constants (of predefined types) and variables
  - Variables are "hooks" that catch the values
  - The symbol _ is called the wildcard pattern: matches to any data (as in Prolog)
- The evaluation is parametrised by data
  - When pattern in data is recognised, the corresponding expression is evaluated.

Statement match

```
match <expression> with
| <pattern_1> -> <expression_1>
| <pattern_2> -> <expression_2>
....
| <pattern_k> -> <expression_k>
```

- Patterns in match must be of the same type
- Pattern must be linear - a variable can appear just once in a pattern
- Patterns in match are tested sequentially!
- The expression with the first match is evaluated
- List of patterns in match must be exhaustive - every value must be matched with a pattern in the list
- First pipe (character | on the first line) is optional

Examples

- Simple match:

```
# let imply v = match v with
    | (false,false) -> true
    | (false, true) -> true
    | (true, false) -> false
    | (true, true) -> true;;
val imply : bool * bool -> bool = <fun>
```

- With variable:

```
let imply2 a b = match (a,b) with
   | (true, x) -> x
   | (false,x) -> true;;
Val imply2 : bool -> bool -> bool = <fun>
```

- With wildchard pattern:

```
let imply3 a b = match (a,b) with
   | (true,false) -> false
   | _ -> true;;
let imply3 : bool -> bool -> bool = <fun>
```

Linearity and completeness

- Every pattern must be linear:

```
let equal a b = match (a,b) with
   | (x,x) -> true
   | _ -> false;;
Error: Variable x is bound several times in this matching
# equal 1 2;;
Error: Unbound value equal
```

- Every pattern must be exhaustive:

```
# let iszero x = match x with 0 -> true;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val iszero : int -> bool = <fun>
# iszero 0;;
- : bool = true
# iszero 10;;
Exception: "Match_failure //toplevel//:3:-34".
```

Combining patterns

Pattern matching is sequential:

1. 'A' is "Consonant"
2. 'A' matched by "Vowel"

```
# let char_discriminate c = match c with
   |’a’ | ’e’ | ’i’ | ’o’ | ’u’ | ’y’
   | ’A’| ’E’ | ’I’ | ’O’ | ’U’ | ’Y’ -> "Vowel"
   | ’a’..’z’ | ’A’..’Z’ -> "Consonant"
   | ’0’..’9’ -> "Digit"
   | _ -> "Other";;
val char_discriminate : char -> string = <fun>
# val char_discriminate ’A’;;
- : string = "Vowel"
# val char_discriminate ’z’;;
- : string = "Consonant"
# val char_discriminate ’$’;;
- : string = "Other"
```

Named patterns

- During pattern matching, it is sometimes useful to name part or all of the pattern. This is useful when one needs to take apart a value while still maintaining its integrity.

```
# let less_rat pr = match pr with
   | ((_,0),p2) -> p2
   | (p1,(_,0)) -> p1
   | (((n1,d1) as r1), ((n2,d2) as r2)) ->
      if (n1*d2) < (n2*d1) then r1 else r2;;
val min_rat : (int * int) * (int * int) -> int * int = <fun>
```

- As a result, the value matched by the named pattern can be returned.

Pattern guards

- Guard is a conditional expression applied immediately after the pattern is matched.
- Syntax:

```
match <expression> with
....
| <pattern_i> when <condition> -> <expression_i>
....
```

- Example:

```
# let eq_rat cr = match cr with
     ((_,0),(_,0)) -> true
   | ((_,0),_) -> false
   | (_,(_,0)) -> false
   | ((n1,1), (n2,1)) when n1 = n2 -> true
   | ((n1,d1), (n2,d2)) when ((n1 * d2) = (n2 * d1)) -> true
   | _ -> false;;
val eq_rat : (int * int) * (int * int) -> bool = <fun>
```

Matching on arguments

```
function
| <pattern_1> -> <expression_1>
....
| <pattern_k> -> <expression_k>
```

- Pattern matching is used in an essential way for defining (unary) functions by cases.
- Ordinary function with one argument:
  - function <x> -> <expression>
  - Using a single pattern

```
# let f (x,y) = 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

```
# let f = function (x,y) -> 2*x + 3*y;;
val f : int * int -> int = <fun>
```

```
# let rec sigma = function
0 -> 0
| x -> x + sigma (x-1) ;;
val sigma : int -> int = <fun>
# sigma 10 ;;
- : int = 55
```

- We will see more examples of functions defined by patterns shortly! Let's consider first pattern matching on lists.

Examples: Matching on arguments

- Size of a list

```
# let rec length = function
   | [] -> 0
   | _::tl -> 1 + length tl;;
val length : 'a list -> int = <fun>
# length [1;2;3;4;5];;
- : int = 5
```

- Joinng two lists

```
# let rec append = function
    | [], l -> l
    | hd::tl, l -> hd :: append (tl,l);;
val append : 'a list * 'a list -> 'a list = <fun>
# append ([1;2;3;4], [5;6;7]);;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

Outline

- History
- Values
- `let` statement, functions
- Blocks, name spaces
- Recursion
- Polymorphism
- Pattern-matching
- Types in functional languages
- Higher-order functions
- Trees

Type declaration

- Type is defined from simpler types using type constructors: *, |, list, array, ...
- Type definition in Ocaml
- Example:

```
type name = typedef ;;
type name1 = typedef1
and name2 = typedef2
...
and namen = typedefn ;;
```

```
# type int_pair = int*int;;
type int_pair = int * int
# let v:int_pair = (1,1);;
val v : int_pair = (1, 1)
```

C:

```
typedef char byte;
typedef struct {int m;} A;
typedef struct {int m;} B;
A x; B y;
x=y; /* incompatible types in assignment */
```

Parametrized types

- Type declarations can include type variables
- Type variable is a variable that can stand for arbitrary type
- Types that include variables are called parametrized types or also polymorphic types
- Parametrized type in Ocaml:

```
# type ('a,'b) pair = 'a*'b;;
type ('a, 'b) pair = 'a * 'b
# let v:(char,int) pair = ('a',1);;
val v : (char, int) pair = ('a', 1)
```

```
type 'a name = typedef ;;
type ('a1 . . . 'an ) name = typedef ;;
```

Products

- Products of types $T_1 * T_2 * \ldots * T_n$
  - Denotation: Cartesian product
    of sets that correspond to types $T_1 T_2 \ldots T_n$
  - $I(T_1 * T_2 * \ldots * T_n) = I(T_1) \times I(T_2) \times \ldots \times I(T_n)$
- Examples in Ocaml:
- Operations
  - Pattern matching
  - fst, snd

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# snd;;
- : 'a * 'b -> 'b = <fun>
```

```
# let (a,b,c) = (1,"2",'3');;
val a : int = 1
val b : string = "2"
val c : char = '3'
# let first t = match t with
        x,_,_ -> x ;;
val first : 'a * 'b * 'c -> 'a = <fun>
# first a ;;
- : int = 1
```

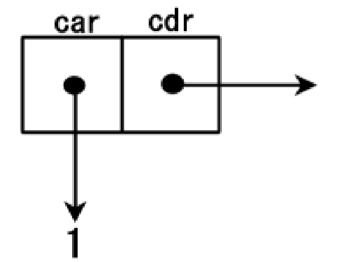
Product: Ocaml examples

```
# type 'param paired_with_integer = int * 'param ;;
type 'a paired_with_integer = int * 'a
# type specific_pair = float paired_with_integer ;;
type specific_pair = float paired_with_integer
# let x:specific_pair = (3, 3.14) ;;
val x : specific_pair = 3, 3.14
```

```
# type ('a,'b,'c) triple = 'a*'b*'c;;
type ('a, 'b, 'c) triple = 'a * 'b * 'c
# let t = 1,'1',"1";;
val t : int * char * string = (1, '1', "1")
# let t:(int,char,string) triple = 1,'1',"1";;
val t : (int, char, string) triple = (1, '1', "1")
```

Lists

- Functional and logic languages
  - Work via recursion and higher-order functions
  - In Lisp a program is a list; can extend itself at run time
  - Built-in polymorphic functions to manipulate arbitrary lists
- Lists in Lisp
  - Two pointers: First (car) and Rest (cdr)
    - Names are historical accidents (contents of address|decrement register)
  - Lists are implemented in this way in Lisp
    - Also in Python, Prolog
  - Lists in Lisp are heterogeneous (of different types)



Lists

- Lists in ML-family
  - Lists in ML are homogeneous (of the same type)
  - Chains of blocks including element and a pointer to the next block
    - Also Clu (Barbara Liskov, 1974), Haskell
- Lists can also be used in imperative programs
  - Implementation as an example
- Lists work best in a language with automatic garbage collection
  - many of the standard list operations tend to generate garbage

Pattern matching on lists

- Matching lists defined with:
  - Matching the head with a variable; can use '_'
  - Matching the tail with a variable
  - Matching list elements as vars
  - Wild card '_' instead of a variable
- Examples with `let` statement
  - All examples get warning:
    - Warning 8 [partial-match]:
      this pattern-matching is not
      exhaustive.
      Here is an example of a case
      that is not matched: []

```
# let h::t = [1; 2; 3; 4];;
val h : int = 1
val t : int list = [2; 3; 4]
# let [a;b;c;d] = [1; 2; 3; 4];;
val a : int = 1
val b : int = 2
val c : int = 3
val d : int = 4
# let h::h1::t = [1; 2; 3; 4];;
val h : int = 1
val h1 : int = 2
val t : int list = [3; 4]
```

Examples: Pattern matching on lists

- Membership test

```
# let rec contains e lst = match lst with
    | [] -> false
    | hd::_ when e = hd -> true
    | _::tl -> contains e tl;;
val contains : 'a -> 'a list -> bool = <fun>
# contains 1 [1;2;3;4;5;6];;
- : bool = true
# contains 10 [1;2;3;4;5;6];;
- : bool = false
```

- Intersection of two lists (as sets)

```
# let rec meet l1 l2 = match l1 with
    | [] -> []
    | hd::tl when (contains hd l2) -> hd :: meet tl l2
    | _::tl -> meet tl l2;;
val meet : 'a list -> 'a list -> 'a list = <fun>
# meet [1;2;3;4;5] [3;4;5;6];;
- : int list = [3; 4; 5]
```

Examples: Pattern matching on lists

- Union of two lists (as sets)

```
# let rec union l1 l2 = match l1 with
    | [] -> l2
    | hd::tl when (contains hd l2) -> union tl l2
    | hd::tl -> hd :: union tl l2;;
val union : 'a list -> 'a list -> 'a list = <fun>
# union [1;2;3;4;5] [4;5;6;10];;
- : int list = [1; 2; 3; 4; 5; 6; 10]
```

Example: Key-value store

- Key-value store implemented in a sorted list
  - Alternative names: dictionary, map, associative array

```
# type ('a,'b) kvstore = ('a*'b) list;;
type ('a, 'b) kvstore = ('a * 'b) list
```

```
# let empty_kvs : ('a,'b) kvstore = [];;
val empty_kvs : ('a, 'b) kvstore = []
# let rec put (k,v) (kvs : ('a,'b) kvstore) : ('a,'b) kvstore =
    match kvs with
    | [] -> [(k,v)]
    | (a,b)::t when k>a -> (a,b)::put (k,v) t
    | l -> (k,v)::l;;
val put : 'a * 'b -> ('a, 'b) kvstore -> ('a, 'b) kvstore = <fun>
```

Example: Key-value store

```
# let s = put (5,"five") (put (1,"one") (put (7,"seven") empty_kvs));;
val s : (int, string) kvstore = [(1, "one"); (5, "five"); (7, "seven")]
```

```
# exception Not_found;;
# let rec get k (kvs : ('a,'b) kvstore) : 'b = match kvs with
    | (a,_)::t when k>a -> get k t
    | (a,b)::_ when a=k -> b
    | _ -> raise Not_found;;
val get : 'a -> ('a, 'b) kvstore -> 'b = <fun>
# get 5 s;;
- : string = "five"
# get 8 s;;
Exception: Not_found.
```

Unions

- Type constructed by union
  - Make a new set by taking the union of existing sets
  - $I(T_1|T_2|...|T_n) = I(T_1) \cup I(T_2) \cup \ldots \cup I(T_n)$
- Unions in Ocaml
- Operations:
  - Construction of instance
  - Pattern matching

```
type name = ...
    | Name_i ...
    | Name_j of t_j ...
    | Name_k of t_k * ...* t_l ... ;;
```

```
# type coin = Heads | Tails;;
type coin = | Heads | Tails
# Tails;;
- : coin = Tails
```

Example: Tarot

```
# type suit = Spades | Hearts |
              Diamonds | Clubs ;;
# type card =
      King of suit
    | Queen of suit
    | Knight of suit
    | Knave of suit
    | Minor_card of suit * int
    | Trump of int
    | Joker ;;
```

```
# King Spades ;;
- : card = King Spades
# Minor_card(Hearts, 10) ;;
- : card = Minor_card (Hearts, 10)
# Trump 21 ;;
- : card = Trump 21
```

Example: Tarot

```
# let rec interval a b = if a = b then [b] else a :: (interval (a+1) b) ;;
val interval : int -> int -> int list = <fun>
# let all_cards s =
     let face_cards = [ Knave s; Knight s; Queen s; King s ]
     and other_cards = List.map (function n -> Minor_card(s,n)) (interval 1 10)
     in face_cards @ other_cards ;;
val all_cards : suit -> card list = <fun>
# all_cards Hearts ;;
- : card list =
[Knave Hearts; Knight Hearts; Queen Hearts; King Hearts;
 Minor_card (Hearts, 1); Minor_card (Hearts, 2); Minor_card (Hearts, 3);
 Minor_card (Hearts, ...); ...]
```

Pattern matching on unions

```
# let string_of_suit = function
        Spades   -> "spades"
      | Diamonds -> "diamonds"
      | Hearts   -> "hearts"
      | Clubs    -> "clubs";;
val string_of_suit : suit -> string = <fun>
# let string_of_card = function
        King c            -> "king of "^ (string_of_suit c)
      | Queen c           -> "queen of "^ (string_of_suit c)
      | Knave c           -> "knave of "^ (string_of_suit c)
      | Knight c          -> "knight of "^ (string_of_suit c)
      | Minor_card (c, n) -> (string_of_int n) ^ "of "^(string_of_suit c)
      | Trump n           -> (string_of_int n) ^ "of trumps"
      | Joker             -> "joker";;
val string_of_card : card -> string = <fun>
```

Outline

- History
- Values
- `let` statement, functions
- Blocks, name spaces
- Recursion
- Polymorphism
- Pattern-matching
- Types in functional languages
- Higher-order functions
- Trees

Higher-order functions

- Higher-order function either takes function as the parameter, or, returns function.
  - We have already seen many examples
- Function compose is an example of higher-order function

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

  - The result is compositum of two functions stated as parameters compose
  - Result is again a function

Example: Higher-order functions

```
# let rec sum_ints (a,b) =
        if (a > b) then 0 else a + sum_ints (a+1,b);;
val sum_ints : int * int -> int = <fun>
# sum_ints (5,7);;
- : int = 18
```

```
# let cube x = x*x*x;;
val cube : int -> int = <fun>
# let rec sum_cubes (a,b) =
        if (a > b) then 0 else cube a  + sum_cubes (a+1,b);;
val sum_cubes : int * int -> int = <fun>
```

```
# let rec fact x = if x=0 then 1 else x*fact (x-1);;
val fact : int -> int = <fun>
# let rec sum_facts (a,b) =
        if (a > b) then 0 else fact a  + sum_facts (a+1,b);;
val sum_facts : int * int -> int = <fun>
```

Example: Higher-order functions

- Functions sum_ints, sum_cubes in sum_facts can be generalized by implementing sum f (a,b).
- Function sum f (a,b) is example of a function in Curry form.
  - All functions sum_* can be now constructed.

```
# let rec sum f (a,b) =
      if (a > b) then 0 else f a + sum f (a+1,b);;
val sum : (int -> int) -> int * int -> int = <fun>
```

```
# let sum_ints = sum (function x -> x);;
val sum_ints : int * int -> int = <fun>
# sum_ints (1,5);;
- : int = 15
# let sum_cubes = sum cube;;
val sum_cubes : int * int -> int = <fun>
# let sum_facts = sum fact;;
val sum_facts : int * int -> int = <fun>
# sum_facts (2,4);;
- : int = 32
```

Currying

- Function with more than one argument can be represented by sequence of functions with one argument (Currying)
- Curry functions can be evaluated partially in order to get some specific function
  - Intermediate functions can be used to define a new fun.
  - They can be linked with other function
  - We will see an example...
- Let’s have a look at the example of functions that transform a function in Curry form into ordinary function, and back.

Example: Currying

```
# let curry f = function x -> function y -> f (x,y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# let uncurry f = function (x,y) -> f x y;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
# uncurry add1;;
- : int * int -> int = <fun>
# curry(uncurry add1);;
- : int -> int -> int = <fun>
# (uncurry add1) (2,3);;
- : int = 5
# curry(uncurry add1) 2 3;;
- : int = 5
```

```
# let add1 x y = x + y;;
val add1 : int -> int -> int = <fun>
# add1 3 4;;
- : int = 7
```

```
# let add2 (x,y) = x + y;;
val add2 : int*int -> int = <fun>
# add2 (3,4);;
- : int = 7
```

Higher-order functions on lists

- Function map is an example of higher-order function

```
# let rec map f l =
if null l then []
else f(List.hd l)::(map f (List.tl l));;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let square x = string_of_int (x*x) ;;
val square : int -> string = <fun>
# map square [1; 2; 3; 4] ;;
- : string list = ["1"; "4"; "9"; "16"]
```

- Higher-order functions can be useful as programming tool
  - Used to manipulate big data! Map-Reduce and Spark.

Higher-order functions on lists

- Function for_all is an example of higher-order function
  - Universal quantification for a property expressed as boolean function f on elements of list l

```
# let rec for_all f l =
if null l then true
else (f (List.hd l)) && for_all f (List.tl l);;
val for_all : ('a -> bool) -> 'a list -> bool = <fun>
# for_all (function n -> n<>0) [-3; -2; -1; 1; 2; 3] ;;
- : bool = true
# for_all (function n -> n<>0) [-3; -2; 0; 1; 2; 3] ;;
- : bool = false
```

Higher-order functions on lists

```
# type 'a option = Some of 'a | None;;
type 'a option = Some of 'a | None
```

```
# let div5 n = if (n mod 5 = 0) then Some n else None;;
val div5 : int -> int option = <fun>
# let rec find_map f = function     (* from module List *)
  | [] -> None
  | h::t when (f h)=None -> find_map f t
  | h::_ -> (f h);;
val find_map : ('a -> 'b option) -> 'a list -> 'b option = <fun>
# find_map div5 [1;2;3;4;5];;
- : int option = Some 5
```

Aggregate functions

- Folding a list
  - fold_left f a [e_1; e_2; ... ; e_n] = f (... (f (f a e_1) e_2) ... e_n).
- Parameters
  - f : 'a -> 'b -> 'a
  - a : 'a
    - Initial value
  - [e_1; ... ; e_n] : 'b list
- Algorithm
  - Liner recursion
  - Result is computed in the second parameter
  - Folding starts on the left side of l;
    it is computed when l = [].

```
# let rec fold_left f a l =
    if null l then a
    else fold_left f (f a (List.hd l)) (List.tl l) ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

```
# let sum_list = fold_left (+) 0 ;;
val sum_list : int list -> int = <fun>
# sum_list [2;4;7] ;;
- : int = 13
# let concat_list = fold_left (^) "";;
val concat_list : string list -> string = <fun>
# concat_list ["Hello "; "world"; "!"] ;;
- : string = "Hello world!"
```

Aggregate functions

- Folding a list
  - fold_right f a [e_1; e_2; ... ; e_n] = f e_1 (f e_2 (... (f e_na) ...)).

```
# let rec fold_right f a l =
if null l then a
else f (List.hd l) (fold_right f a (List.tl l)) ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

  - Parameters are the same as in fold_left
  - Algorithm
    - Linear recursion
    - Folding starts at the right-hand side of l
    - Initial a is used when recursion is completed
    - The result is built backwards: f e_1 ... (f e_{n-1} (f e_n a)) ...)

Functions that construct functions

- Function constructs function

```
# let rec iterate n f =
if n = 0 then (function x -> x)
else compose f (iterate (n-1) f) ;;
val iterate : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

- Construction of function power

```
# let rec power i n =
let i_times = ( * ) i in
iterate n i_times 1 ;;
val power : int -> int -> int = <fun>
# power 2 8 ;;
- : int = 256
```

Outline

- History
- Values
- `let` statement, functions
- Blocks, name spaces
- Recursion
- Polymorphism
- Pattern-matching
- Types in functional languages
- Higher-order functions
- Trees

Example: Binary search tree

```
# type 'a bin_tree =
      Empty
    | Node of 'a bin_tree * 'a * 'a bin_tree ;;
# let t = Node (Node (Empty,2,Empty), 5, Node (Empty,7,Empty));;
val t : int bin_tree =  Node (Node (Empty, 2, Empty), 5, Node (Empty, 7, Empty))
```

```
# let rec insert x = function
      Empty -> Node(Empty, x, Empty)
    | Node(lb, r, rb) -> if x < r then Node(insert x lb, r, rb)
                                          else Node(lb, r, insert x rb) ;;
val insert : 'a -> 'a bin_tree -> 'a bin_tree = <fun>
```

Example: Binary search tree

```
# let rec member x = function
    Empty -> false
  | Node(_,e,_) when e=x -> true
  | Node(l,_,r) -> member x l || member x r;;
val member : 'a -> 'a bin_tree -> bool = <fun>
```

```
# let rec member x = function
    Empty -> false
  | Node(_,e,_) when e=x -> true
  | Node(l,v,r) when x<v -> member x l
  | _ -> member x r;;
val member : 'a -> 'a bin_tree -> bool = <fun>
```

```
# let rec height = function
    Empty -> 0
  | Node( l, _, r ) -> 1 + max (height l) (height r);;
val height : 'a bin_tree -> int = <fun>
# height a;;
- : int = 9
```

Example: Binary search tree

```
# let rec list_of_tree = function
      Empty -> []
    | Node(lb, r, rb) -> (list_of_tree lb) @ (r :: (list_of_tree rb)) ;;
val list_of_tree : 'a bin_tree -> 'a list = <fun>
```

```
# let rec tree_of_list = function
      []     -> Empty
    | h :: t -> insert h (tree_of_list t) ;;
val tree_of_list : 'a list -> 'a bin_tree = <fun>
```

Example: Binary search tree

```
# let t = tree_of_list [1;3;5;7;2;6;4];;
val t : int bin_tree =
  Node (Node (Node (Empty, 1, Empty), 2, Node (Empty, 3, Empty)), 4,
   Node (Node (Empty, 5, Empty), 6, Node (Empty, 7, Empty)))
```

```
# let sort x = list_of_tree (tree_of_list x) ;;
val sort : 'a list -> 'a list = <fun>
# sort [5; 8; 2; 7; 1; 0; 3; 6; 9; 4] ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```