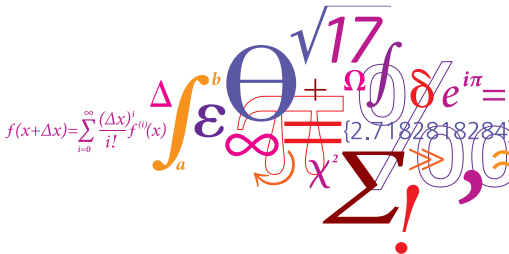


# 02157 Functional Programming

Interpreters for two simple languages

– including exercises

Michael R. Hansen



DTU Informatics

Department of Informatics and Mathematical Modelling

# Purpose

To show the power of a functional programming language, we present a prototype for interpreters for a simple expression language with local declarations and a simple WHILE language.

- Concrete syntax: defined by a contextfree grammar
- Abstract syntax (parse trees): defined by algebraic datatypes
- Semantics, i.e. meaning of programs: inductively defined following the structure of the abstract syntax

succinct programs, fast prototyping

The interpreter for the simple expression language is a higher-order function:

$$eval : Program \rightarrow Environment \rightarrow Value$$

The interpreter for a simple imperative programming language is a higher-order function:

$$I : Program \rightarrow State \rightarrow State$$

# Expressions with local declarations

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

The **abstract syntax** is defined by an algebraic datatype:

```
type ExprTree = | Const of int
                | Ident of string
                | Minus of ExprTree
                | Sum   of ExprTree * ExprTree
                | Diff  of ExprTree * ExprTree
                | Prod  of ExprTree * ExprTree
                | Let  of string * ExprTree * ExprTree;;
```

Example:

```
let et =
  Prod(Ident "a",
       Sum(Minus (Const 3),
           Let("x", Const 5, Sum(Ident "x", Ident "a"))));;
```

# Evaluation in Environments

An *environment* contains *bindings* of identifiers to values.

A `let` tree `Let (str, t1, t2)` is evaluated as in an environment *env*:

- 1 Evaluate  $t_1$  to value  $v_1$
- 2 Evaluate  $t_2$  in the *env* extended with the binding of *str* to  $v_1$ .

An evaluation function

```
eval: ExprTree -> map<string,int> -> int
```

is defined as follows:

```
let rec eval t env =
  match t with
  | Const n      -> n
  | Ident s      -> Map.find s env
  | Minus t      -> - (eval t env)
  | Sum(t1,t2)   -> eval t1 env + eval t2 env
  | Diff(t1,t2)  -> eval t1 env - eval t2 env
  | Prod(t1,t2)  -> eval t1 env * eval t2 env
  | Let(s,t1,t2) -> let v1    = eval t1 env
                    let env1 = Map.add s v1 env
                    eval t2 env1;;
```

Note that the meaning of a let expression is directly represented in the program.

## Example

```
let env = Map.add "a" -7 Map.empty;;  
eval et env;;  
val it : int = 35
```

## Example: Imperative Factorial program

An example of concrete syntax for a factorial program:

```
{Pre: x=K and x>=0}  
  y:=1 ;  
  while !(x=0)  
  do (y:= y*x;x:=x-1)  
{Post: y=K!}
```

Typical ingredients

- Arithmetical expressions
- Boolean expressions
- Statements (assignments, sequential composition, loops, ...)

- Grammar:

$aExp ::= - n \mid v \mid aExp + aExp \mid aExp \cdot aExp \mid aExp - aExp \mid (aExp)$

where  $n$  is an integer and  $v$  is a variable.

- The declaration for the abstract syntax follows the grammar

```
type aExp =          (* Arithmetical expressions *)
  | N of int          (* numbers          *)
  | V of string       (* variables         *)
  | Add of aExp * aExp (* addition          *)
  | Mul of aExp * aExp (* multiplication    *)
  | Sub of aExp * aExp (* subtraction      *)
```

The **abstract syntax** is representation independent (no '+', '-', '(', ')', etc.), no ambiguities — one works directly on syntax trees.

- A **state** maps variables to integers

```
type state = Map<string,int>;;
```

- The meaning of an expression is a function:

```
A: aExp -> state -> int
```

defined inductively on the structure of arithmetic expressions

```
let rec A a s      =  
  match a with  
  | N n           -> n  
  | V x           -> Map.find x s  
  | Add(a1, a2)  -> A a1 s + A a2 s  
  | Mul(a1, a2)  -> A a1 s * A a2 s  
  | Sub(a1, a2)  -> A a1 s - A a2 s;;
```



- Abstract syntax

```
type bExp =      (* Boolean expressions      *)
  | TT          (* true          *)
  | FF          (* false         *)
  | Eq of .... (* equality       *)
  | Lt of .... (* less than     *)
  | Neg of .... (* negation      *)
  | Con of .... ;; (* conjunction  *)
```

- Semantics  $B : bExp \rightarrow State \rightarrow bool$

```
let B b s =
  match b with
  | TT      -> true
  | .....
```

```
type stm =          (* statements          *)
  | Ass of string * aExp      (* assignment *)
  | Skip
  | Seq  of stm * stm        (* sequential composition *)
  | ITE  of bExp * stm * stm (* if-then-else *)
  | While of bExp * stm;;    (* while *)
```

Example of concrete syntax:

```
y:=1 ; while not(x=0) do (y:= y*x ; x:=x-1)
```

Abstract syntax ?

An imperative program performs a sequence of state updates.

- The expression

`update y v s`

is the state that is as `s` except that `y` is mapped to `v`.

Mathematically:

$$(\text{update } y \ v \ s)(x) = \begin{cases} v & \text{if } x = y \\ s(x) & \text{if } x \neq y \end{cases}$$

- Update is a higher-order function with the declaration:

```
let update x v s = Map.add x v s;;
```

- Type?

- The meaning of statements is a function

$$I : stm \rightarrow state \rightarrow state$$

that is defined by induction on the structure of statements:

```
let rec I stm s =  
  match stm with  
  | Ass(x,a)           -> update x ( ... ) s  
  | Skip               -> ...  
  | Seq(stm1, stm2)   -> ...  
  | ITE(b,stm1,stm2)  -> ...  
  | While(b, stm)     -> ... ;;
```

## Example: Factorial function

```

(* {pre: x = K and x>=0}
    y:=1 ; while !(x=0) do (y:= y*x;x:=x-1)
    {post: y = K!} *)

let fac = Seq(Ass("y", N 1),
              While(Neg(Eq(V "x", N 0)),
                    Seq(Ass("y", Mul(V "x", V "y")) ,
                        Ass("x", Sub(V "x", N 1)) )));;

(* Define an initial state *)
let s0 = Map.ofList [("x",4)];;
val s0 : Map<string,int> = map [("x", 4)]

(* Interpret the program *)
let s1 = I fac s0;;
val s1 : Map<string,int> = map [("x", 1); ("y", 24)]

```

- Complete the program skeleton for the interpreter, and try some examples.
- Extend the abstract syntax and the interpreter with *if-then* and *repeat-until* statements.
- Suppose that an expression of the form *inc(x)* is added. It adds one to the value of *x* in the current state, and the value of the expression is this new value of *x*.

How would you refine the interpreter to cope with this construct?