INTERPRETERS
OCAML PROGRAMMING

Ssummary: Why Algebraic Types?

"Algebra" refers to the fact that types contain both sum and product types.

The sum types come from the fact that a value of a variant is formed by *one of* the constructors.

The product types come from that fact that a constructor can carry other component types.

Example

```
type string_or_int =
  | String of string
  | Int of int


let rec sum : string_or_int list -> int = function
   | [] -> 0
   | String s :: t -> int_of_string s + sum t
   | Int i :: t -> i + sum t

let lst_sum = sum [String "1"; Int 2]
```

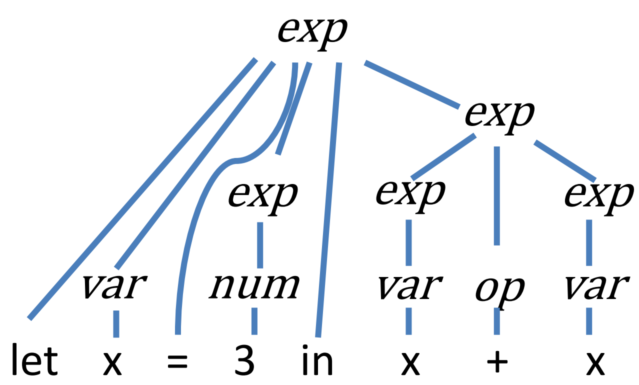
More discussion on …

`https://cs3110.github.io/textbook/chapters/interp/intro.html`

NOW SYNTAX AND INTERPRETERS

SYNTAX: PARSE TREE

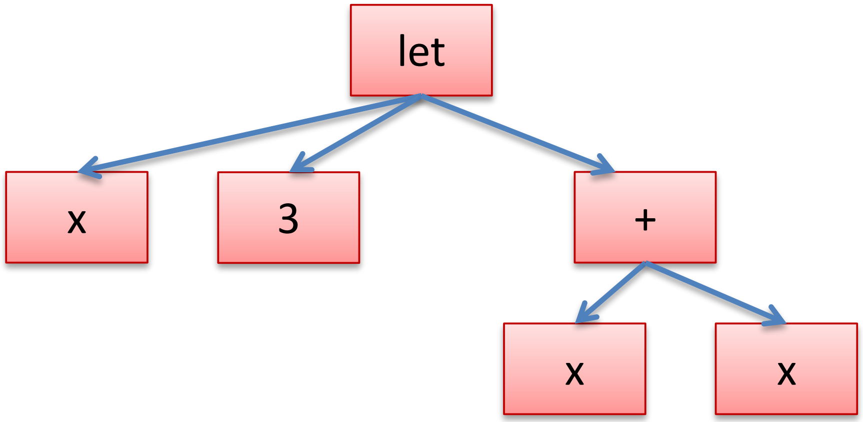
let x = 3 in x + x



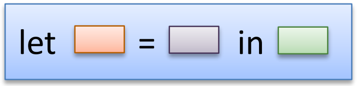
This is the "parse tree." Useful for some purposes, but for the semantics it's Too Much Information.

ABSTRACT SYNTAX TREES (AST)

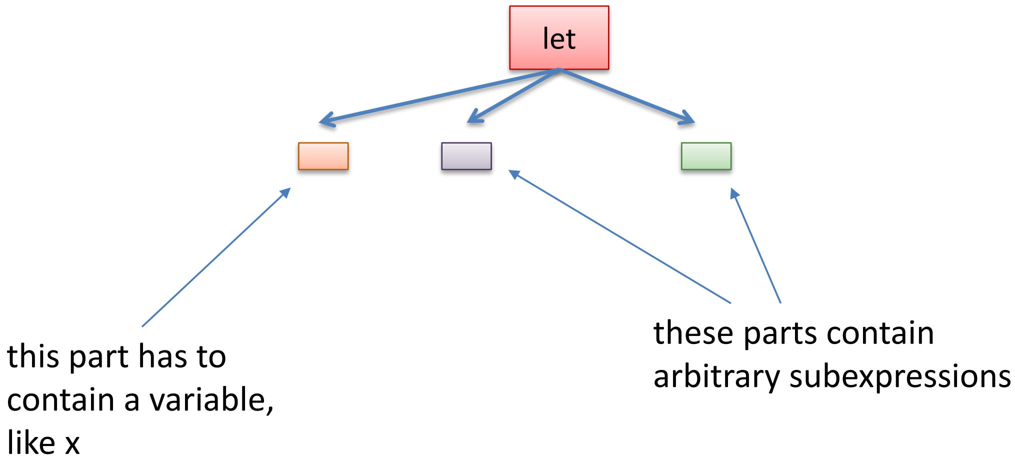
```
let x = 3 in
x + x
```



More generally each let expression has 3 parts:



And you can represent a let expression using a tree like this:



WHY OCAML?

Functional programming languages have sometimes been called "domain-specific languages for compiler writers"

Datatypes are amazing for representing complicated tree-like structures and that is exactly what a program is.

Use a different constructor for every different sort of expression

- one constructor for variables
- one constructor for let expressions
- one constructor for numbers
- one constructor for binary operators, like add
- ...

TODAY

Design and implementation of a functional programming language

→

Approach: Exploit OCML to represent

- The intermediate representation (Abstract Syntax Tree)
- The interpreter of the language

The program

```
Let x = 5 in
 Times(
          Sum(2, x),
          Minus(18, x)
 )
```

CONCRETE SYNTAX

Will be represented by the expression

ABSTRACT SYNTAX

```
Let("x",
   Eint 5,
   Times(Sum(Eint 2, Den "x"), Minus(Eint 18, Den "x"))
   )
```

```
# type ide = string
type exp = Eint of int
  | Den of ide
  | Sum of exp*exp
  | Times of exp * exp
  | Minus of exp * exp
  | Let of ide * exp * exp;;

type ide = string
type exp =
    Eint of int
  | Den of ide
  | Sum of exp * exp
  | Times of exp * exp
  | Minus of exp * exp
  | Let of ide * exp * exp

# Let("x",
    Eint 5,
    Times(Sum(Eint 2, Den "x"), Minus(Eint 18, Den "x"))
    )      ;;
- : exp =
Let ("x", Eint 5, Times (Sum (Eint 2, Den "x"), Minus (Eint 18, Den "x")))
#
```

Evaluation aka Interpreter

AST

```
Let("x",
    Eint 5,
    Times(Sum(Eint 2, Den "x"),
          Minus(Eint 18, Den "x"))
    )
```

x	5

RUN TIME
DATA STRUCTURE
STACK

```
Times(Sum(Eint 2, Den "x"),
      Minus(Eint 18, Den "x"))
```

Binding & scope (you well know…)

With the term **binding** we mean an association between a name and a language entity (function, data structure, object, etc.).

The **scope** of a binding defines that part of the program in which the binding is active.

Environment

The **environment** is defined as the set of name-entity binding existing at run time at a specific point in the program and at a specific time of execution

In the abstract language machine, for each name and for each section of the program, the environment determines the correct association

Environment & Declaration

- **Which are the linguistic construct that allows associations to be introduced into the environment?**

```
int x;
int  f( ) {
  return 0;
}
```

Variable declaration

Function declation

```
type BoolExp =
  | True
  | False
  | Not of BoolExp
  | And of BoolExp*BoolExp
```

Type Declaration

Blocks

```
A: { int aVar;
     aVar = 2;
     B: { char aVar;
              aVar = 'a';
        }
   }
```

In-line block

open A;
open B;
close B;
close A;

Implementing blocks: LIFO

Changes in the environment occur upon entering and leaving blocks (including nested blocks)

Environments

- ***Local environment:*** ***the set of locally declared associations, including the associations related to parameters***
- ***Non-local environment:*** ***associations for names that are visible within the block but not declared in the block itself***
- ***Global environment:*** ***associations for names that can be used by all components of the program***

QUIZ

```
let x = 5;;

let f z = let w = x + z
              in w +1 ;;

let x = 10;;

f 25;
```

- Q1:
- Which is the local environment of the call of f?
- Q2:
- Which is the non local environment of the call of f?

OCAML SIMULATION OF THE ENVIRONMENT

Environment (env)

Type structure (polymorphic) used both at **language design** and in **implementations** to maintain binding between names and values of an appropriate type

Environment

- The environmnet ***env*** is a collection of bindings
  - **env = {x -> 25, y -> 6}**
- **env** contains two "bindings"
  - Binding between **x** and the value **25**
  - Binding between **y** and the value **6**
  - **z** is not bound in **env**
- The env type

  Ide → Value + Unbound
- The constant **Unbound** makes the env function a total function

Environment

- **env: Ide → Value + Unbound**
- **env(x)** denotes either the value **v** bound to **x** in env or the special value **Unbound**
- **env[x=v]** denotes the environment
  - **env[x=v](y) = v** if **y = x**
  - **env[x=v](y)** = **env(y)** if **y != x**
- Assume env = {x -> 25, y -> 7} then

  env[x=5] = {x -> 5, y -> 7}

Implementation (simple)

```
let emptyenv = []
(* the empty environment *)

let rec lookup env x =
    match env with
    | []          -> failwith ("not found")
    | (y, v)::r -> if x = y then v else
lookup r x

let bind env x val = (x val)::env
```

```
let emptyenv = [];;
val emptyenv : 'a list = []

(* the empty environment *)
let rec lookup env x =
  match env with
    | []       -> failwith ("not found")
    | (y, v)::r -> if x = y then v else lookup r x;;
val lookup : ('a * 'b) list -> 'a -> 'b = <fun>

let bind env (x:string) (v:int) = (x,v)::env;;
val bind : (string * int) list -> string -> int -> (string * int) list = <fun>
```

Let’s program …

- We consider the **core of a functional language** subset of OCAML without types or pattern matching
- Goal:
  - To examine all aspects related to the implementation of interpreter
  - of the run-time support for the language

MiniCaml (AST)

```
type ide = string
type exp =
    | CstInt of int
    | CstTrue
    | CstFalse
    | Times of exp * exp
    | Sum of exp * exp
    | Sub of exp * exp
    | Eq of exp * exp
    | Iszero of exp
    | Or of exp * exp
    | And of exp * exp
    | Not of exp
    | Den of ide
    | Ifthenelse of exp * exp * exp
    | Let of ide * exp * exp
    | Fun of ide list * exp
    | Apply of exp * exp list
```

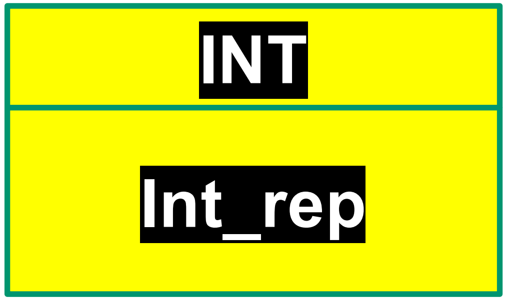
Simple expressions

```
type exp =
        CstInt of int
      | CstTrue
      | CstFalse
      | Times of exp * exp
      | Sum of exp * exp
      | Sub of exp * exp
      | Eq of exp * exp
      | Iszero of exp
      | Or of exp * exp
      | And of exp * exp
      | Not of exp
      | Ifthenelse of exp * exp * exp
```

Expressible values

- **Expressible values (result of the evaluation of expressions)**

```
type evT = Int of int
         | Bool of bool
         | Unbound
```



Type descriptor

- **environment: ide → evT env**

The interpreter

```
let rec eval (e: exp) = match e with
  | CstInt(n) -> Int(n)
  | CstTrue -> Bool(true)
  | CstFalse -> Bool(false)
  | Iszero(e1) -> ?????
  | Den(i) -> ???
```

PRIMITIVE TYPES
INT & BOOL

Typechecking (dynamic)

```
let typecheck (type, typeDescriptor) =
 match type with
  | "int" ->
      (match typeDescriptor with
       | Int(u) -> true
       | _ -> false)
   | "bool" ->
       (match typeDescriptor with
        | Bool(u) -> true
        | _ -> false)
   | _ -> failwith ("not a valid type");;

val typecheck : string * evT -> bool = <fun>
```

Basics

```
let is_zero x = match (typecheck("int",x), x) with
   | (true, Int(y)) -> Bool(y=0)
   | (_, _) -> failwith("run-time error");;

let int_eq(x,y) =
  match (typecheck("int",x), typecheck("int",y), x, y) with
     | (true, true, Int(v), Int(w)) -> Bool(v = w)
     | (_,_,_,_) -> failwith("run-time error ");;

let int_plus(x, y) =
match(typecheck("int",x), typecheck("int",y), x, y) with
   | (true, true, Int(v), Int(w)) -> Int(v + w)
   | (_,_,_,_) -> failwith("run-time error ");;
```

Basics

```
let is_zero x = match (typecheck("int",x), x) with
  | (true, Int(y)) -> Bool(y=0)
  | (_, _) -> failwith("run-time error");;

let int_eq(x,y) =
 match (typecheck("int",x), typecheck("int",y), x, y) with
   | (true, true, Int(v), Int(w)) -> Bool(v = w)
   | (_,_,_,_) -> failwith("run-time error ");;

let int_plus(x, y) =
match(typecheck("int",x), typecheck("int",y), x, y) with
  | (true, true, Int(v), Int(w)) -> Int(v + w)
  | (_,_,_,_) -> failwith("run-time error ");;
```

implementation basic ops

Operazioni di base

```
let is_zero x = match (typecheck("int",x), x) with
  | (true, In
  | (_, _) ->

let int_eq(x
  match (typ
    | (true, t
    | (_,_,_,_

let int_plus
match(type
  | (true, true, Int(v), Int(w)) -> Int(v + w)
  | (_,_,_,_) -> failwith("run-time error ");;
```

The basic operations are implemented through an eager evaluation rule:
before applying the operator, all subtrees (subexpressions) are evaluated

The interpreter

```
let rec eval (e:exp) (env: evT env) =
 match e with
  | CstInt(n) -> Int(n)
  | CstTrue -> Bool(true)
  | CstFalse -> Bool(false)
  | Iszero(e1) -> is_zero(eval e1 env)
  | Eq(e1, e2) -> int_eq((eval e1 env), (eval e2 env))
  | Times(e1,e2) -> int_times((eval e1 env), (eval e2 env))
  | Sum(e1, e2) -> int_plus ((eval e1 env), (eval e2 env))
  | Sub(e1, e2) -> int_sub ((eval e1 env), (eval e2 env))
  | And(e1, e2) -> bool_and((eval e1 env), (eval e2 env))
  | Or(e1, e2) -> bool_or ((eval e1 env), (eval e2 env))
  | Not(e1, env) -> bool_not((eval e1 env))
```

Binding

```
Den(i)-> lookup i env
```

Conditional

```
Ifthenelse(cond,e1,e2) ->
  let g = eval cond env in
    match (typecheck("bool", g), g) with
        | (true, Bool(true)) -> eval e1 env
        | (true, Bool(false)) -> eval e2 env
        | (_, _) -> failwith ("nonboolean guard")
```

```
the conditional  does not follow
an eager strategy: the evaluation
of the subtree is based on the evaluation
of the guard
```

Let semantics

$$\frac{env\ e_1 \rhd v_1 \quad env[x = v_1] \rhd e_2 \Rightarrow v_2}{env \rhd Let(x, e_1, e_2) \Rightarrow v_2}$$

To **evaluate let x = e1 in e2**:

1. Evaluate e1 in the current environment to a value v1.
2. Evaluate e2 in the environment containing the binding between x and v1 to a value v2.
3. The result of evaluating the let expression in the current environment is v2.

Let semantics (intuition)

env =
run-time
stack

push RA su env

$$\frac{env \rhd v_1 \quad env[x = v_1] \rhd e_2 \Rightarrow v\_2}{env \rhd Let(x, e_1, e_2) \Rightarrow e_2}$$

pop env

The interpreter

```
let rec eval((e: exp), (env: evT env)) =
    match e with
    :
    | Let(i, e, ebody) ->
          eval ebody (bind env i (eval e env))
```

REPL

```
# let myp =
  Let("x", CstInt(30), Let("y", CstInt(12), Sum(Den("x"),Den("y"))));;
val myp : exp = Let ("x", CstInt 30, Let ("y", CstInt 12, Sum (Den "x", Den "y")))

# eval myp emptyEnv;;
- : evT = Int 42

# let myp' = CstInt(3);;
val myp' : exp = CstInt 3

# let e = Eq(CstInt(5),CstInt(5));;
val e : exp = Eq (CstInt 5, CstInt 5)

# let myite = Ifthenelse(e,myp,myp');;
val myite : exp =
Ifthenelse (Eq (CstInt 5, CstInt 5), Let ("x", CstInt 30, Let ("y", CstInt 12, Sum (Den
"x", Den "y"))), CstInt 3)

# eval myite emptyEnv;;
- : evT = Int 42
```

Functions

Functional abstraction

- Fun of ide * exp

Application

- **Apply of exp * exp**

Functional abstraction

Anonymous Functions

- **Fun("x", body)**
- "**x**" **formal parameter**,
- **fbody body of the function**

Ocaml expressions

let f x = x+7 in f 2

becomes

```
Let("f",
    Fun("x", Sum(Den("x"), CstInt(7))),
        Apply(Den("f"),CsInt(2))
)
```

A first step

- For simplicity we assume that the functional application is of the first order
  - The first argument of the functional application must be the name of the function to be invoked
  - **Apply(e,arg)** must be of the form **Apply(Den("f"), arg)**
- No recursion.

A bit of semantics

What is the value of a **function**? We assume static scoping

type evT = | Int of int | Bool of bool | Unbound
| Closure of ide * exp * evT env

the expressible value of a functional abstraction is a closure, which includes

- name of the formal parameter (ide)
- code of the declared function (exp)
- environment at the time of declaration (evT env)

Static scoping: nonlocal references of the abstraction are solved in the function declaration environment

Semantics: the value of a function

$$env \rhd Fun(x, e) \Longrightarrow Closure(\text{“}x\text{”}, e, env)$$

An anonymous function is already a value. There is no computation to be performed.

All together now

$$\frac{env \rhd Var(\text{"}f\text{"}) \Rightarrow Closure(\text{"}x\text{"}, body, fDecEnv) \quad env \rhd \mathrm{arg} \Rightarrow va \quad fDecEnv[x = va] \rhd body \Rightarrow v}{env \rhd Apply(Den(\text{"}f\text{"}), arg) \Rightarrow v}$$

To evaluate f arg:

1.Lookup the environment to find the function value (closure) bounded to f.

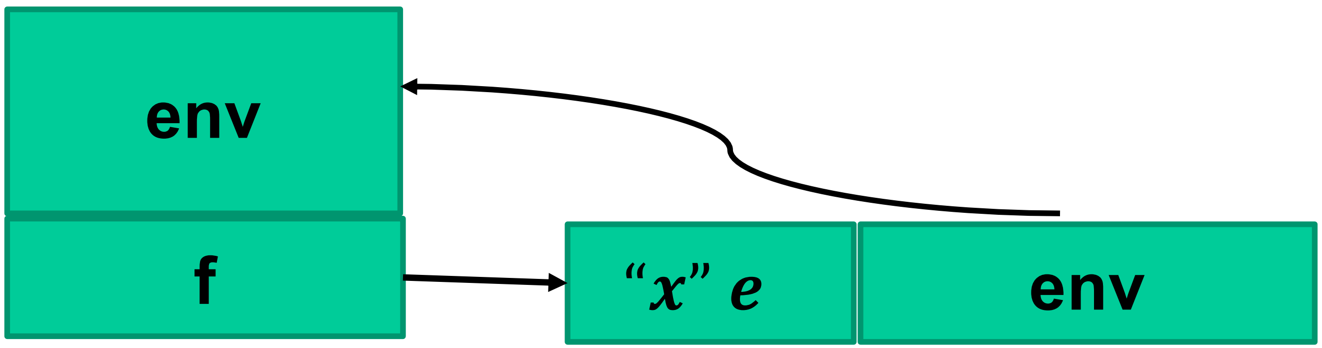
2. Evaluate the actual parameter arg in the current environment to get the actual value va (call-by-value)

3. Perform parameter passing to construct the actual execution environment by taking the declaration environment.

4.Evaluate the body of the function in the actual environment

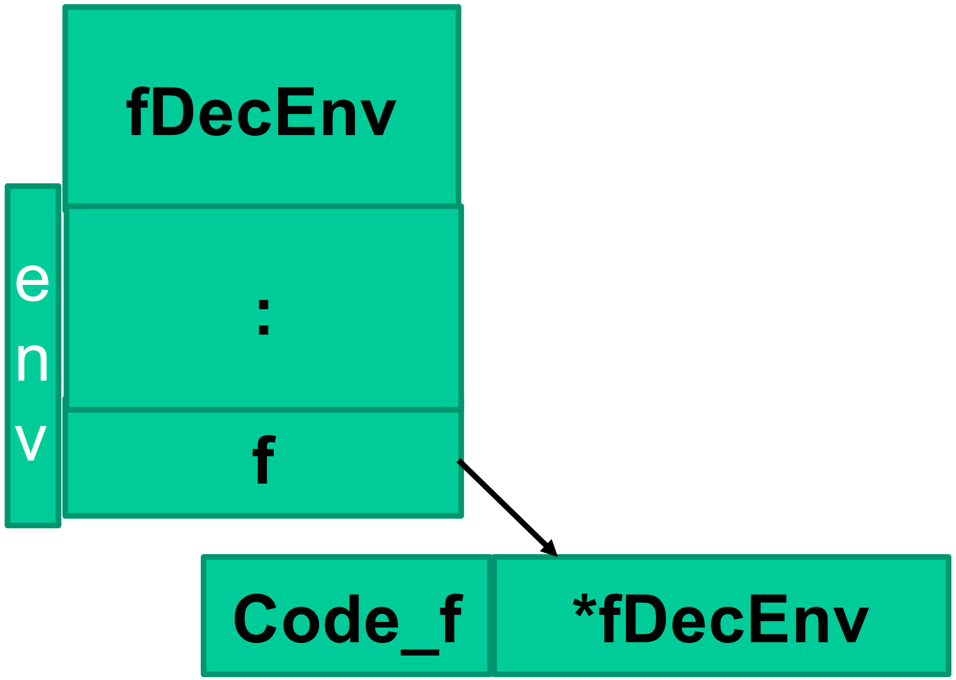
Pictorially

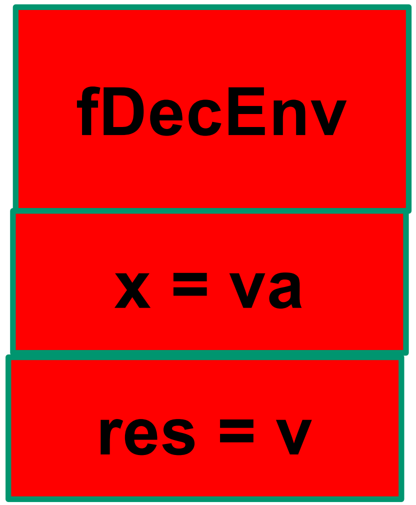
$$env \triangleright Fun(x, e) \Longrightarrow Closure(\text{"}x\text{"}, e, env)$$



Function Application

$$\frac{env \triangleright Den(\text{"}f\text{"}) \implies Closure(\text{"}x\text{"}, body, fDecEnv) \qquad env \triangleright \mathbf{arg} \implies va \qquad fDecEnv[x = va] \triangleright body \implies v}{env \triangleright Apply(Den(\text{"}f\text{"}), arg) \implies v}$$





$\triangleright\ body \implies v$

The interpreter

```
let rec eval((e: exp), (env: evT env)) =
  match e with
   | ...
   | Fun(i, a) -> Closure(i, a, env)
   | Apply(Den(f), eArg) ->
           let fclosure = lookup env f in
            (match fclosure with
               |  Closure(arg, fbody, fDecEnv) ->
                       let aVal = eval eArg env in
                       let aenv = bind fDecEnv arg aVal in
                           eval fbody aenv
               |  _ -> failwith("non functional value"))
    | Apply(_,_) -> failwith("Application: not first order function") ;;
```

REPL

```
# let e = Let ("x", CstInt 5,
   Let ("f", Fun ("z", Sum (Den "z", Den "x")), Apply (Den "f", CstInt 1)));;
val e1 : exp =
  Let ("x", CstInt 5,
   Let ("f", Fun ("z", Sum (Den "z", Den "x")), Apply (Den "f", CstInt 1)))
# eval e1 emptyEnv ;;
- : evT = Int 6
```

Dynamic scope

```
type evT =    | Int of int | Bool of bool | Unbound
              | Funval of efun
 and efun = ide * exp
```

The definition of efun shows that the functional abstraction contains only the code of the declared function

The body of the function will be evaluated in the obtained environment by binding the formal parameters to the actual parameter values in the environment in which the application takes place

Semantics

$$env \rhd Fun(\text{“}x\text{”}, e) \Longrightarrow Funval(\text{“}x\text{”}, e)$$

$$\frac{env \;\; \rhd Den(\text{“}f\text{”}) \Longrightarrow Funval(\text{“}x\text{”}, e) \qquad env \;\; \rhd \mathrm{arg} \Longrightarrow va \quad env[x = va] \rhd e \Rightarrow v}{env \;\; \rhd Apply(Den\text{”}f\text{”}), \mathrm{arg}) \;\; \Longrightarrow v}$$

Interpreter

```
| Fun(arg, ebody) -> Funval(arg,ebody)
| Apply(Den(f), eArg) ->
    let fval = lookup env f in
      (match favl with
        |  Funval(arg, fbody) ->
             let aVal = eval eArg env in
              let aenv = bind env arg aVal in
                eval fbody aenv
        |  _ -> failwith("non functional value"))
| Apply(_,_) -> failwith("Application: not first order function") ;;
```

Recursion

Recursion

```
Let("fact",
    Fun("x", Ifthenelse(Eq(Den "x", Eint 0), Eint 1,
                         Prod(Den "x",
                              Appl(Den "fact",
                                   [Diff(Den "x", Eint 1)])))),
    Appl(Den "fact", [Eint 4]))
```

In OCaml

```
let rec fact x =
        if (x == 0) then 1 else (x * fact(x-1)) in fact(4)
```

OUR INTERPRETER DOES NOT HANDLE RECURSION

The interpreter

```
| Let(i, e1, e2) -> eval(e2, bind (env, i, eval(e1, env)))
| Fun(i, a) -> Closure(i, a, env)
| Apply(Den(f), eArg) ->
            let fclosure = lookup env f in
             (match fclosure with
                |  Closure(arg, fbody, fDecEnv) ->
                       let aVal = eval eArg env in
                       let aenv = bind fDecEnv arg aVal in
                          eval fbody aenv
                |  _ -> failwith("non functional value"))
| Apply(_,_) -> failwith("Application: not first order function")
```

The body **a** (which includes **Den "fact"**) is evaluated in an environment (**aenv**) that extends **fDecEnv** with an association for the formal parameter **x**. But env contains no bindings for the name **"fact"** therefore **Den "fact"** returns **Unbound**!!!

Hence

- To allow recursion we need the body of the function to be evaluated in an environment in which the association between the name and the function has already been entered
- **Design choices**
  - **a different construct for "declaring" recursive functions (such as ML's let rec)**
  - **or a different abstraction construct for recursive functions**

Letrec

```
type exp =
        :
    | Letrec of ide * ide * exp * exp
```

Letrec("f", "x", fbody, letbody)

"f" function name,
"x" formal parameter,
fbody body of the function,
letbody let body.

The factorial

```
Letrec("fact", "n",
Ifthenelse(Eq(Den("n"),CstInt(0),
                 CstInt(1)),
                 Times(Den("n"),Apply(Den("fact"),Sun(Den("n"),CstInt(1))))),
Apply(Den("fact"),CstInt(3)))
```

evT

```
type evT = | Int of int | Bool of bool
  | Unbound | Closure of ide * exp * evT env
  | RecClosure of ide * ide * exp * evT env
```

RecFunVal

```
RecClosure of ide * ide * exp * evT env

RecClosure(funName,
           param,
           funBody,
           staticEnvironment)
```

The code

```
:
| Letrec(f, i, fBody, letBody) ->
   let benv =
     bind(env, f, (Recfunval(f, i, fBody, env)))
       in eval(letBody, benv)
:
```

The recursive closure contains the name of the function itself

The code (2)

```
   :
| Apply(Den f, eArg) ->
    let fclosure = eval(f, r) in
      match fclosure with
        | closure(arg, fbody, fDecEnv) ->
            ::
        | RecClosure(f, arg, fbody, fDecEnv) ->
          let aVal = eval(eArg, env) in
            let rEnv = bind(fDecEnv, f, fclosure) in
              let aEnv = bind(rEnv, arg, aVal) in
                eval(fbody, aEnv)
        | _ -> failwith("non functional value”)
| Apply(_,_) -> failwith(“not function”)
```

REPL

```
# let myRP =
   Letrec("fact", "n",
          Ifthenelse(Eq(Den("n"),EInt(0)),
                     EInt(1),
                     Prod(Den("n"),
                          Apply(Den("fact"),
                                Sub(Den("n"),CstInt(1))))),
          Apply(Den("fact"),EInt(3)));;

val myRP : exp = …

# eval myRP emptyEnv;;
- : eval = Int 6
```

Higher Order Functions

We extend the syntax of the the MiniCaml language to have the possibility of treating functions as first-class values.

This means admitting the possibility that the result of evaluating an expression is a fiunction.

Higher Order Functions

exp ::= …. | Apply of exp * exp

The functional application **Apply(eF, eArg)** is obtained:

1. by evaluating the expression **eF** we get a functional value closure),
2. by evaluating the body of the function (extracted from the closure) in the static environment extended with the binding between the formal parameter and the current parameter value (**eArg**)

Interpreter

```
| Apply(eF, eArg) ->
   let fclosure = eval eF env in
      (match fclosure with
        |  Closure(arg, fbody, fDecEnv) ->
            let aVal = eval eArg env in
               let aenv = bind fDecEnv arg aVal in
                     eval fbody aenv
        |  RecClosure(f, arg, fbody, fDecEnv) ->
             let aVal = eval eArg env in
                let rEnv = bind fDecEnv f fclosure in
                    let aenv = bind rEnv arg aVal in
                          eval fbody aenv
        |  _ -> failwith("non functional value")) ;;
```