

# Principles of Programming Languages

## Recursion and Mutation

---

## Limitations of the Environment Model

The environment model we have introduced in the previous lecture has a limitation: it does not support recursive functions. Consider the following program:

```
(let ((fact (lambda (n)
              (if (= n 0)
                  1
                  (* n (fact (- n 1)))))))
  (fact 3)) ;; => fact: unbound identifier
```

## Limitations of the Environment Model

The reason we cannot invoke **fact** in the body of the closure is that the closure is evaluated in the global environment, **before the fact binding is added**. Hence, when we apply the closure by invoking **(fact 3)**, we evaluate the body of the closure in the same global environment - and not in the environment created by the **let** expression.

The solution introduced in Scheme in order to address this issue is a separate special form called **letrec**. **letrec** has exactly the same syntax as **let**, but **different scoping rules**: the right-side of the bindings in **letrec** are evaluated in the environment which includes the bindings.

```
(letrec ((fact (lambda (n)
                  (if (= n 0)
                      1
                      (* n (fact (- n 1)))))))
  (fact 3)) ;; => 6
```

We address in this lecture how the interpreter must be modified to support **letrec**.

A similar problem exists when evaluating global definitions of recursive functions with **define**.

Recall from the binding rules we presented in a previous lecture that in an expression `(define var val)`, `val` should be evaluated within the scope of the `var` declaration. But our implementation of the environment model does not allow this. And indeed the following test **fails** when we use the L4-eval interpreter (see `L4-eval.test.ts`).

```
expect(bind(parseL4(`
  (L4 (define f
      (lambda (x)
        (if (= x 0)
            1
            (* x (f (- x 1))))))
    (f 3))`), evalProgram)).to.satisfy(isFailure)
```

We remember, however, that similar code **did work** when using the L3-eval interpreter (see `L3.test.ts`):

## Global Recursive Definition

```
expect(bind(parseL3(`
  (L3 (define map
      (lambda (f l)
        (if (eq? l '())
            l
            (cons (f (car l)) (map f (cdr l))))))
    (map (lambda (x) (* x x)) '(1 2 3))))`,
  evalL3program))
.to.deep.equal(makeOk(listPrim([1, 4, 9])));
```

That is, we did not meet the problem of global recursive functions in the substitution model, but we do face it in the environment model. Try to understand why this was the case.

It is impossible to define recursive functions using the `let` expression in both models (substitution and environment models).

In this lecture, we clarify how to process recursive definitions - both local (with `letrec`) and global (with `define`).

The semantic evaluation rule of `letrec` we want to achieve is that the right hand side of the bindings are evaluated in the environment that already includes the bindings.

That is, we want to evaluate:

```
(lambda (n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

in an environment where a binding for **fact** already exists. This is important, because the resulting value - a closure - must keep a reference to this environment if we want the recursion to work.

The problem is naturally that we cannot create such an environment because we haven't yet computed the value for **fact** - chicken and egg problem.

If we could create circular data structures, we could think of a solution to this conundrum.

We find a solution that is specialized to procedures and exploits their property.

Let us consider the syntax of `letrec` - and keep it only for cases where we want to bind procedures to names. In other cases, there is rarely a reason to choose `letrec` instead of `let` (there are some cases where this might be useful, but we will not consider them at this point).

```
<letrec-exp> ::= (letrec (<pbinding>*) <cexp>+)  
// letrec-exp(bindings:List(Binding),  
               body:List(Cexp))  
  
<pbinding> ::= (<var-decl> <proc-exp>)  
// binding(var:Var-decl, val:proc-exp)
```

In this syntax, the right-hand side of a binding in **letrec** can only be procedure expression (**lambda**).

Let us consider the evaluation rule of the closely related **let-exp** and pinpoint what exactly must be changed for **letrec**:

# Recursive Environment

```
If exp = let-exp(bindings, body):
  env-eval(exp, env) is computed by:
    let vars = variables in bindings
        vals = value expressions in bindings
        let cvals = map(val => eval(val, env), vals)
                ^ WE MUST CHANGE THIS FOR LETREC
    return eval-sequence(body,
                          extend-env(env,
                                    make-frame(vars,
                                                cvals)))
```

We must change the way the values of the vars are computed - so that they are computed in the right environment.

This is where we see the opportunity: in order to compute a closure, we do in fact very little - we package three values together, the procedure params, the procedure body and the current environment.

That is, the computation of a closure is really a simple affair - no recursion, just packing 3 values together into a closure structure.

Our problem is that when we want to compute these closures, we do not have access to the `env` value to be packed into the closure (recall that a closure is an object with 3 fields `closure(bindings, body, env)`).

The solution to this problem is: **delay**.

We do not have the **env** value when we create the closure - but we do not really need it at this point. We will need it in the future when the closure is applied. When the closure will be applied, we will have access to the new **env**.

Putting these ideas together, we define a new version of the environment data structure which supports a new special form of environments which we call recursive environments:

# Recursive Environment

```
// Add a new option for environments
<env> ::= <empty-env> | <extended-env> | <recursive-env>

// no fields - a singleton data type.
<empty-env> ::= ()

// Linked list of frames as previously
<extended-env> ::= extended-env(frame:Frame,
                                enclosing-env:Env)

<rec-env> ::= rec-env(vars:List(string),
                    paramss:List(List(var-decl)),
                    bodies:List(List(Cexp)),
                    enclosing-env:Env)
```

A **rec-env** stores a frame mapping names to “almost closures” - that is a list of params and a body. We do **not** store the closure-env in the **rec-env**.

But when we compute the **apply-env** of a **rec-env** to a string, we construct the closure at lookup time, with the correct **env**:

# Recursive Environment

```
define apply-env(env:Env, var:string)
  // Same definition as the regular env
  if env is empty-env:
    error "Var not found"
  else if env is extended-env:
    if var is found in env->frame(env)
      return apply-frame(frame, var)
    else return apply-env(env->enclosing-env(env), var)

...
```

# Recursive Environment

```
define apply-env(env:Env, var:string)
  ...

  // New case of the recursive env
  else if env is rec-env:
    if var is the ith item in env->vars:
      let params = env->paramss(env)[i]
      body = env->bodies(env)[i]
      // Construct the closure at lookup time
      // using the env itself
      return make-closure(params, body, env)
    else
      return apply-env(env->enclosing-env(env), var)
```

Observe how the closure is constructed at lookup time - when we retrieve the value of a variable from a recursive environment. At this point, we have access to all the required components: params, body and the env to construct the correct closure.

The solution we have implemented for `letrec` must be extended to support recursive toplevel definitions using `define` expressions.

## Handling Recursive `define` Expressions

Recall that we handled `define` expressions in the `eval-program` procedure: we obtain a list of expressions, which can be either `CExp` or `DefineExp` expressions. We evaluate each one in turn, if it is a `DefineExp`, we evaluate it and obtain a new environment, which is then used for the remaining expressions.

To support recursive `define` expressions, we must consider the cases where the value of the `DefineExp` is a `ProcExp` or a non-procedural expression.

# Handling Recursive define Expressions

We change this code:

```
const evalDefineExps =  
  (def: DefineExp, exps: Exp[], env: Env): Result<Value> =>  
    bind(applicativeEval(def.val, env),  
      (rhs: Value) => evalSequence(exps,  
                                     makeExtEnv([def.var.var],  
                                                 [rhs], env))));
```

# Handling Recursive define Expressions

To this:

```
const evalDefineExps =  
  (def: DefineExp, exps: Exp[], env: Env): Result<Value> =>  
    isProcExp(def.val) ?  
      evalSequence(exps, makeRecEnv([def.var.var], [def.val.args],  
                                     [def.val.body], env)) :  
    bind(applicativeEval(def.val, env),  
        (rhs: Value) => evalSequence(exps,  
                                     makeExtEnv([def.var.var],  
                                                 [rhs], env))));
```

## Handling Recursive `define` Expressions

With this definition, our interpreter supports recursive toplevel procedures such as this program:

```
(define f
  (lambda (n)
    (if (= n 0)
        1
        (* n (f (- n 1))))))
```

This implementation of `define`, however, has 2 limitations:

The following program fails:

```
(define f (lambda (x) (g x)))  
(define g (lambda (x) x))  
(f 5) ;; => g is undefined
```

## Forward Definitions and Toplevel Mutual Recursion

This happens because, when we evaluate the first **define**, we obtain a closure which refers to the environment which contains **f** - but **g** is not yet defined.

When we later define **g** in the second **define**, the environment attached to the **f** closure is not updated - it still does not know about **g**.

This behavior does not fit what happens in Scheme and in JavaScript.

But it is in fact similar to what happens in languages like Java and C++ and Pascal. In such languages, a function cannot use in its body another function which is not already declared (in a header or in an import).

The second problem is that we cannot define toplevel mutually recursive procedures in this implementation:

```
(define even?  
  (lambda (x) (if (= x 0) #t (odd? (- x 1)))))  
(define odd?  
  (lambda (x) (if (= x 0) #f (even? (- x 1)))))  
(even? 10) ;; => odd? is not defined
```

We now develop a variant environment model which implements the behavior of the global environment according to the Scheme semantics and supports the 2 cases above.

Another way to support recursion in the interpreter - for both `letrec` and `define` - is to use **mutation in the environment**.

Let us first introduce support for mutation in our subset of TypeScript by introducing the **Box** data type.

The box datatype is a way to encapsulate places where we want to allow mutation in our program.

The box datatype is defined by the following interface:

```
// Purpose: value constructor for box datatype
// Signature: makeBox(v)
// Type: [T -> Box(T)]

// Purpose: Type predicate for box datatype
// Signature: isBox(x)
// Type: [any -> Boolean]
```

The box datatype is defined by the following interface:

```
// Purpose: Accessor for the box datatype
// Signature: unbox(b)
// Type: [Box(T) -> T]

// Purpose: Mutator for the box datatype
// Signature: setBox(b, v)
// Type: [Box(T) * T -> void]
```

We do not really need the box datatype in TypeScript, since TypeScript supports mutation (it is not a pure functional language). But we introduce this datatype to mark explicitly the places where we use mutation - so that we can analyze and control precisely the impact of these mutations.

We implement the box datatype as follows:

```
// Box datatype  
// Encapsulate mutation in a single type.  
type Box<T> = T[];  
const makeBox = <T>(x: T): Box<T> => ([x]);  
const unbox = <T>(b: Box<T>): T => b[0];  
const setBox = <T>(b: Box<T>, v: T): void => { b[0] = v; }
```

On the basis of the Box datatype, we define a new mutable environment data type. We distinguish two types of environments:

- Contains the primitive bindings when the interpreter starts in a single frame.
- The frame can be extended with new bindings (using the **define** special form).
- Has no enclosing-environment.
- Is the last element in all the extended environments.

## Extended Environments

- These are all the other environments constructed at runtime;
- They form a linked chain of frames ending with the global-environment;
- One cannot add a binding to the frame of an extended environment (in contrast to the global environment).
- But one can change the value of variable in a binding after the binding has been initialized.

# Mutable Environment Data Type

In this new model, the data types of `env` are:

```
<box-env> ::= <global-env> | <extended-box-env>
```

```
<global-env> ::= (global-env frame)  
// global-env(frame:Box(Frame))
```

```
<extended-box-env> ::= (extended-box-env frame enclosing-env)  
// extended-box-env(vars:List(string), frame: Frame)
```

```
<fbinding> ::= (var val)  
// binding(var:string, val:Box(Value))
```

```
<frame> ::= (frame (var val)*)  
// frame(bindings:List(fbinding))
```

# Mutable Environment Data Type

The following operations are defined on this datatype:

```
;; Purpose: lookup the value of a var in a frame.  
;; Signature: applyFrame(frame, var)  
;; Type: [Frame * string -> Value]
```

```
;; Purpose: update the value of a binding in a frame  
;; Signature: setVarFrame(frame, var, val)  
;; Type: [Frame * string * Value -> void]
```

```
;; Purpose: lookup the value of a var in an environment  
;; Signature: applyEnv(env, var)  
;; Type: [Box-env * string -> Value]
```

A key transformation in this model is that variables are now bound to Boxes that contain their values. In all previous models, variables were directly bound to Values

The following operations are defined on this datatype:

```
interface FBinding {  
    tag: "FBinding";  
    var: string;  
    val: Box<Value>;  
};
```

## Mutable Environment Data Type

```
const isFBinding = (x: any): x is FBinding =>
  x.tag === "FBinding";
const makeFBinding = (v: string, val: Value): FBinding =>
  ({tag: "FBinding", var: v, val: makeBox(val)});

const getFBindingVar = (f: FBinding): string => f.var;
const getFBindingVal = (f: FBinding): Value => unbox(f.val);
const setFBinding = (f: FBinding, val: Value): void =>
  { setBox(f.val, val); };
```

## Mutable Environment Data Type

```
interface Frame {  
    tag: "Frame";  
    fbindings: FBinding[];  
};
```

# Mutable Environment Data Type

```
const makeFrame = (vars: string[], vals: Value[]): Frame => ({
  tag: "Frame",
  fbindings: zipWith(makeFBinding, vars, vals)
});
const extendFrame = (frame: Frame, v: string, val: Value): Frame => ({
  tag: "Frame",
  fbindings: cons(makeFBinding(v, val), frame.fbindings)
})
const isFrame = (x: any): x is Frame => x.tag === "Frame";
const frameVars = (frame: Frame): string[] =>
  map(getFBindingVar, frame.fbindings);
const frameVals = (frame: Frame): Value[] =>
  map(getFBindingVal, frame.fbindings);
```

# Mutable Environment Data Type

```
const applyFrame = (frame: Frame, v: string): Result<FBinding> => {  
    const pos = frameVars(frame).indexOf(v);  
    return (pos > -1) ? makeOk(frame.fbindings[pos])  
        : makeFailure(`Var not found: ${v}`);  
};  
  
const setVarFrame =  
    (frame: Frame, v: string, val: Value): Result<void> =>  
        bind(applyFrame(frame, v),  
            (bdg: FBinding) => makeOk(setFBinding(bdg, val)));
```

The global-env is modeled as a single global variable - `theGlobalEnv` with a specific method to add a binding to the single frame of the global env:

# Mutable Environment Data Type

```
interface GlobalEnv {  
  tag: "GlobalEnv";  
  frame: Box<Frame>;  
};  
  
const isGlobalEnv = (x: any): x is GlobalEnv =>  
  x.tag === "GlobalEnv";  
const makeGlobalEnv = (): GlobalEnv =>  
  ({tag: "GlobalEnv", frame: makeBox(makeFrame([], []))});  
const theGlobalEnv = makeGlobalEnv();
```

# Mutable Environment Data Type

```
const globalEnvSetFrame = (ge: GlobalEnv, f: Frame): void =>
    setBox(ge.frame, f);

const globalEnvAddBinding = (v: string, val: Value): void =>
    globalEnvSetFrame(theGlobalEnv,
        extendFrame(unbox(theGlobalEnv.frame), v, val));

const applyGlobalEnvBdg =
    (ge: GlobalEnv, v: string): Result<FBinding> =>
        applyFrame(unbox(ge.frame), v);
```

On the basis of the box-env, we implement the `letrec` evaluation rule as follows:

## Recursion with Box-Env

```
const evalLetrec = (exp: LetrecExp, env: Env): Result<Value> => {
  const vars = map((b: Binding) => b.var.var, exp.bindings);
  const vals = map((b: Binding) => b.val, exp.bindings);
  const extEnv = makeExtEnv(vars,
                             repeat(undefined, vars.length),
                             env);
  // @@ Compute the vals in the extended env
  const cvalsResult =
    mapResult((v: CExp) => applicativeEval(v, extEnv), vals);
  const result = bind(
    cvalsResult,
    (cvals: Value[]) => makeOk(
      zipWith((bdg, cval) => setFBinding(bdg, cval),
        extEnv.frame.fbindings, cvals)
    )
  );
  return bind(result, _ => evalSequence(exp.body, extEnv));
};
```

Observe how we first create the environment with dummy temporary values for all the vars declared in the **letrec** (in the expression `makeExtEnv(vars, repeat(undefined, vars.length), env)`), then we evaluate them in this environment; finally we update the bindings with the computed values.

`define` expressions are handled as part of the `eval-program` which is modified as follows:

## Recursion with Box-Env

```
const evalDefineExps = (def: DefineExp, exps: Exp[]): Result<Value> =>
  bind(applicativeEval(def.val, theGlobalEnv),
    (rhs: Value) => { globalEnvAddBinding(def.var.var, rhs);
                      return evalSequence(exps, theGlobalEnv); });
```

The evaluation rule for **DefineExp** is the only place which invokes the special method **globalEnvAddBinding** which adds a binding to the single frame of the global environment.

When this is performed, in effect, all the extended environments which exist at this point are modified - since their last frame is updated.

The frame is updated by adding the new binding at the beginning of the frame - thus shadowing the previously defined value for the defined string. We could change the implementation to prevent the re-definition of strings in the global environment.

The change we have brought to the mutable environment is quite profound.

- In all the previous models we had, variables were bound to values.
- In this model, variables are bound to boxes that contain values. The bindings are thus mutable.

This modification opens the route to implementing variants of the interpreter such as passing arguments by reference as can be done in C++. We will not pursue this route further, but obviously it would have deep impact on the language design.

In general, when describing the operational semantics of a programming language, we have so far described two sets:

- Expressions
- Values

We now see that in addition to these two sets - it is important to also describe the set called **Denoted Values** which is the set of objects bound to variables. In all the interpreters we have seen before the box-env model, the set structure of the interpreters was:

- Input: Expressions (AST)
- Computed values: Value (Number | Boolean | Sexp | Void | Closure)
- Denoted values = Computed values

In the last model, we obtain:

- Input: Expressions (AST)
- Computed values: Value (Number | Boolean | Sexp | Void | Closure)
- Denoted values = Box(Computed values)

We could design a language which distinguishes mutable and immutable variables. In this case, the structure of the sets would be:

- Input: Expressions (AST)
- Computed values: Value (Number | Boolean | Sexp | Void | Closure)
- Denoted values = Box(Computed values) | Computed Values

This is basically the structure of TypeScript when using the Immutable.js package mentioned in the first chapter.