Programming Languages (Langages Evolués)

Roel Wuyts

Scheme in Scheme

Goal

- Show a simple interpreter
- Study semantics of a limited Scheme
- Focusing on special forms
  - if, let, lambda, begin, set!, define, letrec
  - function application
  - function evaluation

Roadmap

- Environments
- Self-evaluating expressions
- Variable
- Scheme function evaluation
- Special forms
- Read-Eval Loop and initialization
- Analyzing tests

Syntax? What!

- Manipulate parenthesized expressions
- Basically trees
- No parser, scanner needed
- Using the Scheme (read) function

%Sch

```
<expr> ::=
<constant>
    | IDENT
    | (<special-form>  ...)
    | (<expr> <expr> ...)
<special-form> ::=
(lambda (IDENT*) <expr> <expr> ...)
   | (if <expr> <expr> <expr>)
   | (let ((IDENT <expr>)*) <expr>+)
   | (letrec ((IDENT <expr>)*) <expr>+)
   | (set! IDENT <expr>)
   | (begin <expr>*)
   | (define IDENT <expr>)
   | (quote <expr>)
<constant> ::= NUMBER | #t | #f
```

Evaluation

- Finding the value of an expression only makes sense within an environment
- Example:

```
(let ((x 2))
   (+ x a))
```

  - Which context?
  - What is the value of the variable +?
  - What is the value of the variable a?
- Let's define %*eval*

self-evaluating?

- self-evaluating elements are elements that have themselves as value
- The booleans and numbers of %sch are the ones of scheme
- Code:

```
(define (%self-evaluating? expr)
  (or (number? expr) (boolean? expr)))
```

- So

```
(define (%eval expr env)
  (cond
        ((%self-evaluating? expr) expr)
         ...))
```

Environment

- An environment is an ordered list of *binding tables*
- A binding table
  - represents a list of bindings
  - is list of pairs (symbol . value), like this:

```
((a . 2) (b . 3) (c . 4))
```

Environments Example

```
(let (  (a 1)
        (b 2) )
  (let (  (x 5)
          (y 10) )
    (* y (+ a b))))
```

env0: (%top-level-env)

env1: (((a . 1) (b . 2))
%top-level-env)

env2: (((x . 5) (y .10))
((a . 1) (b . 2))
%top-level-env)

%make-binding

```
(define (%make-binding id val)
  ;; creates a binding for a table binding element
  (cons id val))
```

%binding

(%binding id env)

- Look for a binding in a binding table
  - left to right
- Examples

```
(%binding ‘a ‘(((a . 20) (a . 1) (b . 2) (z . 26))))
=> ‘(a . 20)

(%binding ‘a ‘(((a . 20) (b1 . 3)) ((a . 1) (b . 2) (z . 26))))
=> ‘(a . 20)

(%binding ‘e ‘(((a . 20) (a . 1) (b . 2) (z . 26))))
=> #f
```

%binding definition

```
(define (%binding id env)
  ;; symbol * env -> binding
  ;; return the first binding whose car is id.
  ;; return #f when no binding is found
  (if (null? env)
      #f
      (or (assq id (car env)) (%binding id (cdr env)))))
```

To determine whether some item is
the first item of a pair in a list of pairs

%lookup

(%lookup id env)

- Lookup the value of an identifier in an environment
- Examples

```
(%lookup ‘kk ‘(((a . 20) (b1 . 3)) ((a . 1) (b . 2) (z . 26))))
=> error
(%lookup ‘a ‘(((a . 20) (b1 . 3)) ((a . 1) (b . 2) (z . 26))))
=> 20
```

%lookup definition

```
(define (%lookup id env)
  ;; return the val of id in env
  ;; error if id is not defined in env
  (let ((binding (%binding id env)))
    (if (boolean? binding)
        (error "Error: unknown identifier " id)
        (cdr binding))))
```

%extend-env

(%extend-env lids lvals env)

- return a new binding table composed of a table binding from a list of identifiers *lids* and a list of values *lvals* and *env*
- Example

```
(equal? (%extend-env '(a z) '(100 200) %test-env)
           (cons '((a . 100) (z . 200)) %test-env))
```

%extend-env definition

```
(define (%extend-env lids lvals env)
  ;; add a new a binding table in front of the env
  (cons (map %make-binding lids lvals) env))
```

Evaluating a Variable

The value of a variable is the value of the most recent binding for the variable found in the environment

```
(define (%eval expr env)
  (cond
       ((%self-evaluating? expr) expr)
       ((symbol? expr) (%lookup expr env))
        ...))
```

Special Form

- A special form does not follow applicative order
    - Control the order of argument evaluation
- Code

```
(define (%special? expr)
  (and (pair? expr)
       (member (car expr)
               '(if let letrec
                    lambda begin set! define quote))))
```

- So

```
(define (%eval expr env)
  (cond ((%self-evaluating? expr) expr)
        ((symbol? expr) (%lookup expr env))
        ((%special? expr) (%eval-special expr env))
        ...))
```

Evaluating Functions

- (f a b c)
  - *f* is evaluated as well as *a*, *b*, and *c*
  - The *value* of f is applied to the *values* of a, b, and c
  - The order of evaluation is unspecified
- So

```
(define (%eval expr env)
  (cond ((%constant? expr) expr)
        ((symbol? expr) (%lookup expr env))
        ((%special? expr) (%eval-special expr env))
        (else (%apply (%eval (car expr) env)
                      (%eval-list (cdr expr) env)))))
```

Lexical (static) scoping revisited...

- Remember: Variables occuring free in the procedure should obey the lexical scope rule
  - e.g. they are bound statically
- Example

```
(let ( (x 5) )
   (define (foo y)
       (+ y x))
   (foo 4))
(define x 600)
=> 9
```

Closures

- We need to keep bindings around when procedures are defined
    - if not, we can not properly evaluate a procedure
- Closure: implementation technique for representing procedures with free variables

Closure Representation

- Representing a closure as:
  - a parameter list
  - a body = an expression
  - a reference towards its environment of compilation
- So:
  `(magic-closure-identifier (x) (+ x a) env)`

Closure code

```
(define magic-closure-tag '*closure*)

(define (%make-closure args body env)
  (list magic-closure-tag args body env))

(define (%closure? clos)
  (and (pair? clos)
       (eq? (car clos) magic-closure-tag)))


;;accesing parts of the closure
(define (%closure-args clos)
  (cadr clos))

(define (%closure-body clos)
  (caddr clos))

(define (%closure-env clos)
  (cadddr clos))
```

why *pair?*

(%apply f l)

- So we had this:
```
(%apply (%eval (car expr) env)
        (%eval-list (cdr expr) env))
```
- Why %apply does not require the current environment? (%apply f l env)?
  - ***primitive procedures*** are just applied
    - (+ 2 3)
  - ***user-defined functions*** keep a reference to their compilation environment

Primitive Representation

- Representing primitives, i.e., functions not defined by the user
  - a name
  - a Scheme function
- So: `(magic-primitive-tag symbol function)`

Primitive code

```scheme
(define magic-primitive-tag '*primitive*)

(define (%primitive? proc)
  (and (pair? proc)
       (eq? (car proc) magic-primitive-tag)))

(define (%make-primitive symbol function)
  (list magic-primitive-tag symbol function))


;; accessing
(define (%primitive-symbol prim)
  (cadr prim))

(define (%primitive-function prim)
  (caddr prim))
```

%apply definition

```
(define (%apply proc largs)
  (cond ((%primitive? proc) (%apply-primitive proc largs))
        ((%closure? proc) (%apply-procedure proc largs))
        (else
           (error "Bad ! Un-apply-able object !" proc))))

(define (%apply-primitive primitive largs)
  (apply (%primitive-function primitive) largs))

(define (%apply-procedure proc largs)
  ;; apply a user-defined procure: evaluate proc body
  ;; in the closure environment extended with
  ;; proc arguments and largs
  (%eval (%closure-body proc)
         (%extend-env (%closure-args proc) largs
                                 (%closure-env proc))))
```

%eval

- Nearly finished:

```
(define (%eval expr env)
  (cond ((%constant? expr) expr)
        ((symbol? expr) (%lookup expr env))
        ((%special? expr) (%eval-special expr env))
        (else (%apply (%eval (car expr) env)
                      (%eval-list (cdr expr) env)))))

(define (%eval-list l env)
  ;; list * env -> list of val
  (if (null? l)
      '()
      (cons
         (%eval (car l) env)
          (%eval-list (cdr l) env))))
```

- Now we just need to add the special forms...

Special Forms

```
(define (%eval-special expr env)
  (case (car expr)
    ((if)       (%eval-if (cdr expr) env))
    ((lambda)   (%eval-lambda (cdr expr) env))
    ((quote)    (%eval-quoted (cdr expr)))
    ((begin)    (%eval-sequence (cdr expr) env))
    ((let)      (%eval-let (cdr expr) env))
    ((define)   (%eval-define (cdr expr) env))
    ((set!)     (%eval-set! (cdr expr) env))
    ((letrec)   (%eval-letrec expr env))
    (else (error "Cannot evaluate unknown Special Form"
                      (car expr)))))
```

if

```
(define (%eval-if expr env)
  ;; evaluates the three-element list expr
  ;; expr = (e1 e2 e3)
  ;; where first element represents the conditional
  ;; second is the true case, and
  ;;third is the false case
  (let
    ((bool (%eval (car expr) env)))
      (if bool
        (%eval (cadr expr) env)
        (%eval (caddr expr) env))))
```

lambda

- Simply creates a closure!
- Code:

```
(define (%eval-lambda expr env)
  ;; expr = ((x) (+ x 2))
  (%make-closure (car expr) (cadr expr) env))
```

quote

```
(define (%eval-quoted expr)
  ;; expr = (...)
  ;; '(+ 2 3) <=> (quote (+ 2 3))
  ;; => expr = ((+ 2 3))
  (car expr))
```

begin

- Evaluate sequence of expressions
- Return the value of the last expression

```
(define (%eval-sequence lexpr env)
  ;; lexpr = (e1 e2 ...en)
  (if (null? lexpr) '()
      (let ((expr (car lexpr))
            (rest (cdr lexpr)))
        (if (null? rest)
            (%eval expr env)
            (begin
              (%eval expr env)
              (%eval-sequence rest env))))))
```

Let

- Not essential but handy to obtain local variable
- Remember:

```
(let ((x 3) (+ x x)) <=> ((lambda (x) (+ x x)) 3)
```

- We evaluate the body in an environment extended with the new variables bound to their values.
- Code:

```
(define (%eval-let expr env)
  ;; expr = (((x 3) (y 4)) x)
  (let ((lvars (map car (car expr)))
        (lvals (map cadr (car expr)))
        (body (cdr expr)))
    (%eval-sequence body
       (%extend-env lvars (%eval-list lvals env) env))))
```

define

- Change existing value or create a new binding
- Code:

```
(define (modify-env! id val env)
  (let ((binding (%binding id env)))
    (if (not binding)
        (set-car! env
              (cons (%make-binding id val) (car env)))
        (set-cdr! binding val))))

(define (%eval-define expr env)
  ;; expr = (id expression)
  (modify-env! (car expr) (%eval (cadr expr) env) env)
  'undefined)
```

set!

- Modifies value of existing binding
- Code

```
(define (%eval-set! expr env)
  ;; expr = (id expression)
  (let ((binding (%binding (car expr) env)))
    (if (not binding)
        (error "Identifier not defined" (car expr))
        (set-cdr! binding (%eval (cadr expr) env)))))
```

letrec

- Remember:

```
(let ((fac (lambda (n)
                  (if (= 0 n)
                      1
                      (* n (fac (- n 1)))))))
    (fac 5))
  => error: fac unknown!
```

- Works with letrec
  - expressions should be evaluated in an environment that already contains the variables that will be linked to the expressions

letrec

```
(define (%eval-letrec expr env)
  ;; expr = (((x 3) (y 4)) body)
  (let ((lvars (map car (car expr)))
        (lvals (map cadr (car expr)))
        (body (cdr expr)))
    (%eval-sequence body
        (%extend-env-rec lvars lvals env ))))

(define (%extend-env-rec lids lexp env)
  ;; return a new environment in which
  ;; lexp have been evaluated in an extended
  ;; environment in which lids were predefined.
  (let ((envRec (%extend-env lids lexp env)))
    (let ((newBindingTable (car envRec)))
      (for-each (lambda (binding exp)
                  (set-cdr! binding (%eval exp envRec)))
                newBindingTable lexp)
      envRec)))
```

The icing on the cake

- We now can evaluate expressions
- So let's finish the rest:
  - establish the top environment
  - define the read-eval-print loop

%top-level-env

```scheme
(define %top-level-env '())

(define %primitive-symbols
  '(+ - * / = < > equal? null? cons car cdr))

(define %primitive-functions
  (list + - * / = < > equal? null? cons car cdr))

(define (%initialize-top-level-env)
  (set! %top-level-env
     (%extend-env
        %primitive-symbols
        (map
          %make-primitive
          %primitive-symbols
          %primitive-functions)
        %top-level-env)))
```

%Sch R-E-P

```
(define (%read)
  (printf "? ")
  (read))

(define (%print val)
  (printf "=> ~a\n" val))

(define (%rep)
  ;; the read-eval-loop - enter quit to quit !sch
  (let ((expr (%read)))
    (if (eq? 'quit expr)
        ()
        (begin
          (%print (%eval expr %top-level-env))
          (%rep)))))

(define (%sch)
  (%initialize-top-level-env)
  (%rep))
```

Dynamic Scoping

- Current %Sch uses lexical scoping
- Let's change it to have dynamic scoping
  - like most of the Lisp environments
- Where is the impact?

Lexical scoping revisited...

- Remember: Variables occuring free in the procedure are looked up in the current environment
  - e.g. they are bound dynamically
- Example

```
(let ( (x 5) )
   (define (foo y)
       (+ y x))
   (foo 4))
(define x 600)
=> 605
```

%eval dynamically scoped

- Closures no longer needed
  - no need to capture environment
- Evaluating expressions:

```
(define (%eval expr env)
  (cond
        ((%self-evaluating? expr) expr)
        ((symbol? expr) (%lookup expr env))
        ((%special? expr) (%eval-special expr env))
        (else
         (%apply (%eval (car expr) env)
                 (%eval-list (cdr expr) env)
                 env))))
```

But needs an environment!

%apply dynamically scoped

```
(define (%apply proc largs env)
  (cond ((%primitive? proc) (%apply-internal proc largs))
        ((%user-function? proc)
                         (%apply-function proc largs env))
        (else
            (error "Bad ! Un-apply-able object !" proc))))


(define (%apply-function proc largs env)
  ;; apply a closure: evaluate proc body in
  ;; extended current environment with
  ;; proc arguments and largs
  (%eval (%function-body proc)
         (%extend-env (%function-args proc) largs env)))
```

%eval-lambda dynamically scoped

```
(define (%eval-lambda-lisp expr)
  ;; expr = ((x) (+ x 2))
  (%make-function (car expr) (cadr expr)))


;; so functions needed, but no closures
;; represent function with special tag, args and body
;; so does not need env!

(define magic-function-tag '*function*)

(define (%user-function? func)
  (and (pair? func)
       (eq? (car func) magic-function-tag)))

(define (%make-function args body )
  (list magic-function-tag args body))

(define (%function-args func)
  (cadr func))

(define (%function-body func)
  (caddr func))
```

Wrap-up

- We have defined a Scheme interpreter in Scheme
  - no need for parser
  - had detailed look at (nested) environments
  - implemented lexical and dynamic scoping
    - difference between closures and functions

References

- http://www.ulb.ac.be/di/rwuyts/INFO020_2003/