

CSc 372

Comparative Programming Languages

40 : Scheme — Metacircular Interpretation

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2013 Christian Collberg

Introduction

- In this lecture I'm going to show how you can define Scheme by writing a **metacircular interpreter** for the language, i.e. an interpreter for Scheme written in Scheme.
- Before we can do that, we first need to learn a few more things about the language

Let Expressions

- A **let-expression** binds names to values:

```
(let ((name1 value1) (name2 value2) ...)
      expression)
```

- The first argument to let is a list of (name value) pairs.
The second argument is the expression to evaluate.

```
> (let ((a 3) (b 4) (square (lambda (x)(* x x)))
        (plus +))
    (sqrt (plus (square a) (square b))))
```

5.0

- Let-expressions can be nested:

```
> (let ((x 5) (c 4))  
    (let ((v (* 4 x))  
          (t (* 2 c)))  
      (+ v t)))
```

28

Imperative Features

- Scheme is an **impure** functional language.
- I.e., Scheme has **imperative** features.
- I.e., in Scheme it is possible to program with **side-effects**.

(set! var value) Change the value of *var* to *value*.

(set-car! var value) Change the *car*-field of the cons-cell *var* to *value*.

(set-cdr! var value) Change the *cdr*-field of the cons-cell *var* to *value*.

Imperative Features...

- Example:

```
> (let ((x 2) (l '(a b)))  
    (set! x 3)  
    (set-car! l '(c d))  
    (set-cdr! l '(e))  
    (display x) (newline)  
    (display l) (newline))  
3  
((c d) e)
```

Dotted Pairs

- S-expressions are constructed using **dotted pairs**.
- It is implemented as a struct (called a **cons-cell**) consisting of two fields (the size of a machine word) called `car` and `cdr`.
- We can manipulate these fields directly:

```
> '(1 . 2)
(1 . 2)
> (cons "stacy's" "mom")
("stacy's" . "mom")
> '(1 . (2 . 3))
(1 2 . 3)
> (cons 1 2)
(1 . 2)
```

Dotted Pairs...

- When the second part of a dotted pair (the cdr-field) is a list, and the innermost cdr-field is the empty list, we get a “normal” Scheme list:

```
> '(1 . ())
```

```
(1)
```

```
> '(1 . (2 . ()))
```

```
(1 2)
```

```
> '(1 . (2 3))
```

```
(1 2 3)
```


Dotted Pairs...

- We can use `set-car!` and `set-cdr!` to manipulate the fields of a cons-cell directly:

```
> (define x '(1 . 2))  
> (set-car! x 'a)  
> x  
(a . 2)  
> (set-cdr! x '(2 3))  
> x  
(a 2 3)
```

Dotted Pairs...

- `(cons A B)` can be thought of as first creating a cons-cell on the heap (using `malloc`, for example), and then setting the `car` and `cdr` fields to `A` and `B`, respectively:

```
> (define x (cons 0 0))  
> x  
(0 . 0)  
> (set-car! x '1)  
> (set-cdr! x '())  
> x  
(1)
```

Loops

Scheme's "for-loop" `do` takes these arguments:

- 1 A list of triples (*var init update*) which declares a variable *var*, with an initial value *init*, and which gets updated using the expression *update*, on each iteration;
- 2 A pair (*termination_cond return_value*) which gives the termination condition and return value of the loop; and
- 3 a loop body:

```
(do ((var1 init1 update1)
     (var12 init2 update2)
     ...
    )
    (termination_cond return_value)
    loop_body
)
```

Loops...

- Sum the numbers 1 to 4, printing out intermediate results:

```
> (do ((i 1 (+ i 1))
      (sum 0 (+ sum i)))
      (= i 5) sum)
      (display sum)
      (newline)
)
0
1
3
6
10
```

Association Lists

- **Association lists** are simply lists of *key-value* pairs that can be searched sequentially:

```
> (assoc 'bob '((bob 22) (joe 32) (bob 3)))  
(bob 22)
```

- The list is searched from the beginning to end, returning the first pair with a matching key:

(assoc *key alist*) Search for *key*; compare using `equal?`.

(assq *key alist*) Search for *key*; compare using `eq?`.

(assv *key alist*) Search for *key*; compare using `eqv?`.

Association Lists...

```
> (define e '((a 1) (b 2) (c 3)))  
> (assq 'a e)  
(a 1)  
> (assq 'b e)  
(b 2)  
> (assq 'd e)  
#f  
> (assq (list 'a) '((a) (b) (c)))  
#f  
> (assoc (list 'a) '((a) (b) (c)))  
((a))  
> (assv 5 '((2 3) (5 7) (11 13)))  
(5 7)
```

- We can actually have more than one value:

```
> (assoc 'bob '((bob 5 male)
                (jane 32 'female)))
(bob 5 male)
```

- **Apply** returns the result of applying its first argument to its second argument.

```
> (apply + '(6 7))
```

```
13
```

```
> (apply max '(2 5 1 7))
```

```
7
```


- `(eval arg)` evaluates its argument.

```
> (eval '(+ 4 5))
```

```
9
```

```
> (eval '(cons 'a '(b c))) (a b c)
```

- `eval` and `quote` are each other's inverses:

```
> (eval '(+ 4 5))
```

```
(+ 4 5)
```

```
> (eval (eval '(+ 4 5)))
```

```
9
```

```
> (eval (eval (eval ' '(+ 4 5))))
```

```
9
```

Programs as Data

- Scheme is **homoiconic**, self-representing, i.e. programs and data are both represented the same (as S-expressions).
- This allows us to write programs that generate programs - useful in AI, for example.

```
> (define x 'car)
> (define y '(a b c))
> (define p (list x y))
> p
(car '(a b c))
> (eval p)
a
```

Evaluation Order

- So far, we have said that to evaluate an expression `(op arg1 arg2 arg3)` we first evaluate the arguments, then apply the operator `op` to the resulting values.
- This is known as `applicative-order` evaluation.
- Example:

```
(define (double x) (* x x))
```

```
> (double (* 3 4))  
⇒ (double 12)  
⇒ (+ 12 12)  
⇒ 24
```

Evaluation Order...

- This is not the only possible order of evaluation
- In **normal-order** evaluation parameters to a function are always passed unevaluated.
- This sometimes leads to extra work:

```
(define (double x) (* x x))
```

```
> (double (* 3 4))  
⇒ (+ (* 3 4) (* 3 4))  
⇒ (+ 12 (* 3 4))  
⇒ (+ 12 12)  
⇒ 24
```

Evaluation Order...

- Applicative-order can sometimes also lead to more work than normal-order:

```
(define (switch x a b c)
  (cond
    ((< x 0) a)
    ((= x 0) b)
    (> x 0) c)))
```

```
> (switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
```

- Here, applicative-order evaluates all the arguments, although only one value will ever be needed.

Evaluation Order...

- Ordinary Scheme functions (such as `+`, `car`, etc) use applicative-order evaluation.
- Some **special forms** (`cond`, `if`, etc) must use normal order since they need to consume their arguments unevaluated:

```
> (if #t (display 5) (display 6))
```

```
5
```

```
> (cond (#f (display 5))  
        (#f (display 6))  
        (#t (display 7)))
```

```
7
```

A Metacircular Interpreter

- One way to define the semantics of a language (the effects that programs written in the language will have), is to write a **metacircular interpreter**.
- I.e, we define the language by writing an interpreter for it, in the language itself.
- A metacircular interpreter for Scheme consists of two mutually recursive functions, `mEval` and `mApply`:

```
(define (mEval Expr)
  ...
)
(define (mApply Op Args)
  ...
)
```


A Metacircular Interpreter...

- We want to be able to call our interpreter like this:

```
> (mEval (+ 1 2))
```

```
3
```

```
> (mEval (+ 1 (* 3 4)))
```

```
13
```

```
> (mEval (quote (2 3)))
```

```
(2 3)
```

```
> (mEval (car (quote (1 2))))
```

```
1
```

A Metacircular Interpreter...

```
> (mEval (cdr (quote (1 2))))  
(2)  
> (mEval (cons (quote 5) (quote (1 2))))  
(5 1 2)  
> (mEval (null? (quote (1 2))))  
#f  
> (mEval (null? (quote ())))  
#t  
> (mEval (if (eq? 1 1) 5 6))  
5
```

A Metacircular Interpreter. . .

- `mEval` handles primitive special forms (lambda, if, const, define, quote, etc), itself.
- Note that, for these forms, we must use normal-order evaluation.
- For other expressions, `mEval` evaluates all arguments and calls `mApply` to perform the required operation:

A Metacircular Interpreter...

```
(define (mEval Expr)
  (cond
    [(null? Expr) '()]
    [(number? Expr) Expr]
    [(eq? (car Expr) 'if)
     (mEvalIf (cadr Expr)
              (caddr Expr)
              (cadddr Expr))]
    [(eq? (car Expr) 'quote) (cadr Expr)]
    [else (mApply (car Expr)
                  (mEvalList (cdr Expr)))]
  )
)
```

A Metacircular Interpreter...

- `mApply` checks if the operation is one of the builtin primitive ones, and if so performs the required operation:

```
(define (mApply Op Args)
  (case Op
    [(car) (caar Args)]
    [(cdr) (cdar Args)]
    [(cons) (cons (car Args) (cadr Args))]
    [(eq?) (eq? (car Args) (cadr Args))]
    [(null?) (null? (car Args))]
    [(+) (+ (car Args) (cadr Args))]
    [(*) (* (car Args) (cadr Args))]
  )
)
```

A Metacircular Interpreter...

- Some auxiliary functions:

```
(define (mEvalIf b t e)
  (if (mEval b) (mEval t) (mEval e))
)
```

```
(define (mEvalList List)
  (cond
    [(null? List) '()]
    [else (cons (mEval (car List))
                 (mEvalList (cdr List)))]
  )
)
```

A Metacircular Interpreter. . .

- Note that this little interpreter lacks many of Scheme's functions.
- We don't have `symbols`, `lambda`, `define`.
- We can't define or invoke user-defined functions.
- There are no way to define or lookup variables, local or global. To do that, `mEval` and `mApply` pass around **environments** (*association lists*) of variable/value pairs.

Readings and References

- Read Scott, pp. 592–606, 609–610