CSc 520

Principles of Programming Languages

9: Scheme — Metacircular Interpretation

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg
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 this 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**.

- \( \text{set! } \text{var } \text{value} \) Change the value of \text{var} to \text{value}.
- \( \text{set-car! } \text{var } \text{value} \) Change the \text{car}-field of the cons-cell \text{var} to \text{value}.
- \( \text{set-cdr! } \text{var } \text{value} \) Change the \text{cdr}-field of the cons-cell \text{var} to \text{value}.
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:

```lisp
> '(1 . 2)
(1 . 2)
> (cons "stacy's" "mom")
("stacy's" . "mom")
> '(1 . (2 . 3))
(1 2 . 3)
> (cons 1 2)
(1 . 2)
```
When the second part of a dottend pair (the \textit{cdr}-field) is a list, and the innermost \textit{cdr}-field is the empty list, we get a “normal” Scheme list:

\begin{verbatim}
> ' (1 . ())
(1)
> ' (1 . (2 . ()))
(1 2)
> ' (1 . (2 3))
(1 2 3)
\end{verbatim}
Dotted Pairs...

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

```scheme
> (define x '(1 . 2))
> (set-car! x 'a)
> x
  (a . 2)
> (set-cdr! x '(2 3))
> x
  (a 2 3)
```
(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” \texttt{do} takes these arguments:

1. A list of triples \((\text{var init update})\) which declares a variable \texttt{var}, with an initial value \texttt{init}, and which gets updated using the expression \texttt{update}, on each iteration;

2. A pair \((\text{termination_cond return_value})\) which gives the termination condition and return value of the loop; and

3. a loop body:

\[
\begin{align*}
\text{(do ((var1 init1 update1)} \\
& \quad (\text{var12 init2 update2}) \\
& \quad \cdots \\
& ) \\
& \text{(termination_cond return_value)} \\
& \text{loop_body}
\end{align*}
\]
Loops...

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

```scheme
> (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:

\[
> \text{(assoc 'bob '(((bob 22) (joe 32) (bob 3)) (bob 22))}
\]

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

- \text{(assoc key alist)}  Search for key; compare using equal?.
- \text{(assq key alist)}  Search for key; compare using eq?.
- \text{(assv key alist)}  Search for key; compare using eqv?.

[13]
Association Lists...

```lisp
> (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)
```
Association Lists...

We can actually have more than one value:

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

(bob 5 male)
Apply

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...

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.

```scheme
> (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 \((\text{op} \ \text{arg1} \ \text{arg2} \ \text{arg3})\) we first evaluate the arguments, then apply the operator \(\text{op}\) to the resulting values.

This is known as **applicative-order** evaluation.

Example:

\[
\text{(define (double } x) \ (* \ x \ x))
\]

\[
> \ (\text{double } (* \ 3 \ 4)) \quad \Rightarrow \ (\text{double } 12) \quad \Rightarrow \ (+ \ 12 \ 12) \quad \Rightarrow \ 24
\]
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.
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:

```scheme
(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
mEval handles \textbf{primitive special forms} (\texttt{lambda}, \texttt{if}, \texttt{const}, \texttt{define}, \texttt{quote}, etc), itself.

Note that, for these forms, we must use normal-order evaluation.

For other expressions, \texttt{mEval} evaluates all arguments and calls \texttt{mApply} to perform the required operation:
(define (mEval Expr)
  (cond
   [(null? Expr) '(())]
   [(number? Expr) Expr]
   [(eq? (car Expr) 'if)
     (mEvalIf (cadr Expr) (caddr Expr) (cadddr Expr))]
   [(eq? (car Expr) 'quote) (cadadr 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:

\[
\text{(define (mApply Op Args)}
\text{(case Op)}
\text{[ (car) (caar Args)]}
\text{[ (cdr) (cdar Args)]}
\text{[ (cons) (cons (car Args) (cadr Args))]}\]
\text{[ (eq?) (eq? (car Args) (cadr Args))]}\]
\text{[ (null?) (null? (car Args))]}\]
\text{[ (+) (+ (car Args) (cadr Args))]}\]
\text{[ (*) (* (car Args) (cadr Args))]}\]
\text{)}
\text{)}
\text{)}
Some auxiliary functions:

\[
\begin{align*}
(\text{define } (\text{mEvalIf } b \ t \ e) & \quad (\text{if } (\text{mEval } b) \ (\text{mEval } t) \ (\text{mEval } e)) \\
(\text{define } (\text{mEvalList } List) & \quad (\text{cond} \\
& \quad \quad [(\text{null? } List) \ '()] \\
& \quad \quad [\text{else} \ (\text{cons } (\text{mEval } (\text{car } List)) \\
& \quad \quad \quad (\text{mEvalList } (\text{cdr } List)))] \\
\end{align*}
\]
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