

6.001 Recitation 19: Interpretation

RI: Gerald Dalley, dalleyg@mit.edu, 25 Apr 2007
<http://people.csail.mit.edu/dalleyg/6.001/SP2007/>

Announcements

- Quiz 2: $A \geq 85$, $B \geq 70$, $C \geq 55$, $D \geq 40$ (drop date is tomorrow)
 - Reminder: `set!` is a special form that changes bindings in environments. Its first argument *must* be a name (symbol)¹.
- Friday: Prof. Szolovits substitutes; Project 4 is due.
- Recitation 16 notes update: quick explanation of advanced error handling with `raise` and `with-handlers`). None of this is required material for 6.001.
- `apply` actually allows for a variable-length argument list, as you may have noticed in Project 4. If you're curious about the full syntax, see the Scheme documentation (e.g. google `MIT Scheme apply`).

Interpretation

Today we'll be talking about interpreting computer languages. In particular, we'll continue developing an interpreter for a Scheme-like language. Typical interpreters have the following components:

Input Example	Interpreter Stage
"(square (+ 5 3))"	lexical analyzer (breaks up strings into tokens)
	parser (tokens → internal representations; sequences of tokens → trees)
	evaluator (processes the parse tree to get values)
64	printer (converts results into strings and puts them onto the screen)
"64"	

We'll be using Scheme's built-in lexical analyzer, parser, and printer, but write our own evaluator. Take 6.035 to learn about the extra pieces.

For today's recitation, we'll be using a version of the interpreter from the online tutor. See the extra handout for the code. For each practice problem, change the interpreter to support the given special forms, some of which should already be familiar to you from Scheme.

¹It turns out that Common LISP actually has a special form called `setf` that works like an assignment operator in many other languages. In particular, it allows for an expression as the first argument, e.g. if we evaluate `(define x '())` and then `(setf (car x) 10)`, then evaluating `x` will yield '(10). Allowing for this type of behavior is surprisingly complex. Ask your TA or recitation instructor (not in class) if you're curious about what it involves.

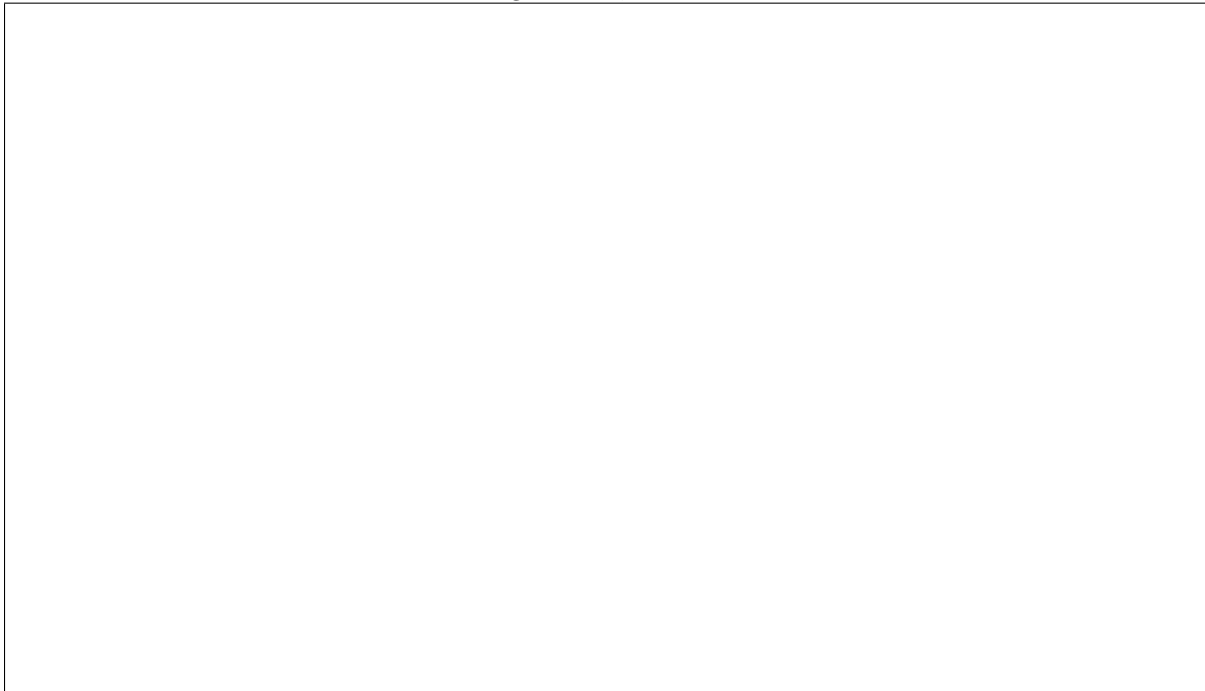
Mutation!

`(set!* var exp)` evaluates `exp` and assign its value to `var`. Although `set!` in Scheme has an undefined return value, your `set!*` should return `var`'s previous value.



Quotation

`(quote* expr)` returns `expr` without evaluating it. Also, make it work for `(quote expr)`.



Sugared procedure definitions

`(define* (name arg arg ...) body)` defines a procedure name with arguments `(arg arg ...)` and body `body`.



Who's smarter than a 360th version?

Earlier in the semester, we noted that DrScheme v360 has a “feature” allowing either of the following

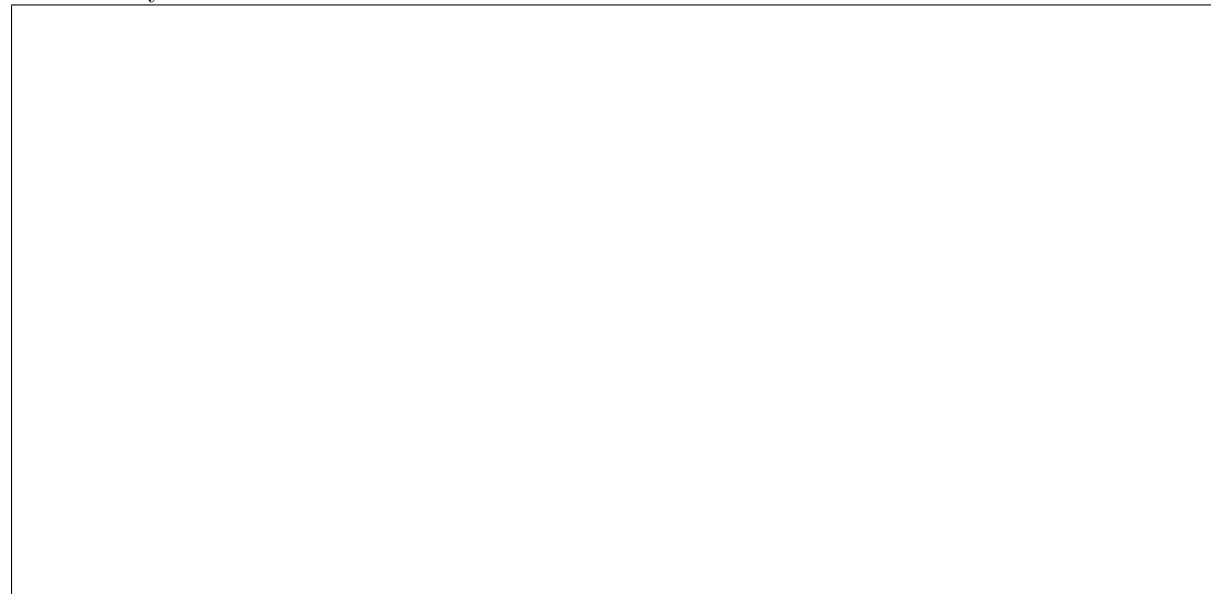
```
(define (quote x) 10)
```

```
(define 'x 10)
```

but this causes the `quote` special form to get clobbered. How could we prevent the shadowing of special forms in Scheme*? If someone tries to shadow a special form, for example with

```
(define* (quote* x) 10)
```

return the symbol `reserved-word-error`.



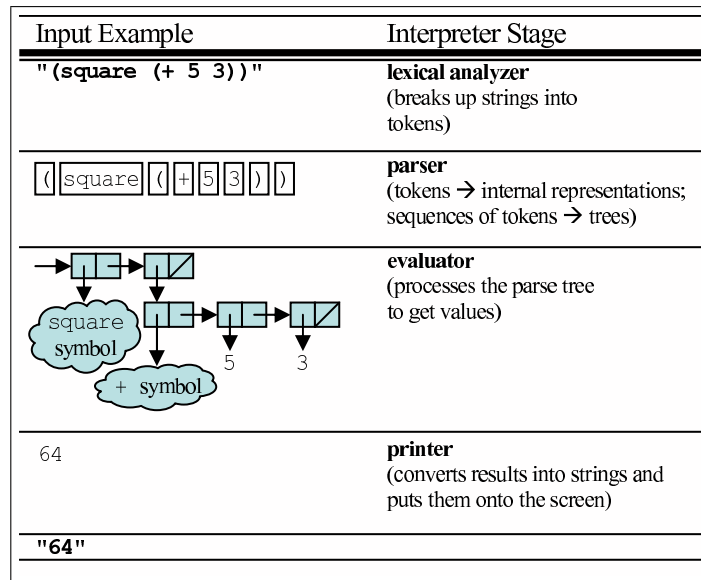
Solutions

Announcements

- Quiz 2: $A \geq 85$, $B \geq 70$, $C \geq 55$, $D \geq 40$ (drop date is tomorrow)
 - Reminder: `set!` is a special form that changes bindings in environments. Its first argument *must* be a name (symbol)².
- Friday: Prof. Szolovits substitutes; Project 4 is due.
- Recitation 16 notes update: quick explanation of advanced error handling with `raise` and `with-handlers`). None of this is required material for 6.001.
- `apply` actually allows for a variable-length argument list, as you may have noticed in Project 4. If you're curious about the full syntax, see the Scheme documentation (*e.g.* google MIT Scheme apply).

Interpretation

Today we'll be talking about interpreting computer languages. In particular, we'll continue developing an interpreter for a Scheme-like language. Typical interpreters have the following components:



We'll be using Scheme's built-in lexical analyzer, parser, and printer, but write our own evaluator. Take 6.035 to learn about the extra pieces.

For today's recitation, we'll be using a version of the interpreter from the online tutor. See the extra handout for the code. For each practice problem, change the interpreter to support the given special forms, some of which should already be familiar to you from Scheme.

²It turns out that Common LISP actually has a special form called `setf` that works like an assignment operator in many other languages. In particular, it allows for an expression as the first argument, *e.g.* if we evaluate `(define x '())` and then `(setf (car x) 10)`, then evaluating `x` will yield '(10). Allowing for this type of behavior is surprisingly complex. Ask your TA or recitation instructor (not in class) if you're curious about what it involves.

Mutation!

(set!* var exp) evaluates exp and assign its value to var. Although set! in Scheme has an undefined return value, your set!* should return var's previous value.

```
(load "eval-code.scm")
(display "SET.SCM_...") (newline)

; Added support for set!* special form
(define (eval exp env)
  (cond
    ((number?      exp) exp)
    ((symbol?      exp) (lookup exp env))
    ((define?      exp) (eval-define exp env))
    ((if?          exp) (eval-if exp env))
    ((lambda?      exp) (eval-lambda exp env))
    ((let?         exp) (eval-let exp env))
    ((set!?        exp) (eval-set exp env))
    ((application? exp) (apply* (eval (car exp) env)
                                 (map (lambda (e) (eval e env))
                                      (cdr exp)))))
    (else
     (error "unknown_ expression_" exp))))

; Need to be able to check for the set!* special form...
(define (set!? exp)
  (tag-check exp 'set!*))

; Handles (set!* var val-expr)
(define (eval-set exp env)
  (let* ((var      (second exp))
        (val-expr (third exp))
        (binding  (lookup-binding var env))
        (old-val  (second binding)))
    (set-binding-value! binding (eval val-expr env))
    old-val))

; Does the same thing as lookup, but returns the full binding instead of just the value.
(define (lookup-binding name env)
  (if (null? env)
      (error "unbound_ variable:_ " name)
      (let ((binding (table-get (car env) name)))
        (if (pair? binding)
            binding
            (lookup name (cdr env))))))

(define (set-binding-value! binding new-value)
  (set-car! (cdr binding) new-value))

; Quick test
(eval '(define* x 10) GE) ; -> undefined
(eval '(set!* x 20) GE) ; -> 10
(eval 'x GE) ; -> 20
```

Quotation

`(quote* expr)` returns `expr` without evaluating it. Also, make it work for `(quote expr)`.

```
(load "eval-code.scm")
(display "QUOTE.SCM_" (newline))

; Added quote* special form
(define (eval exp env)
  (cond
    ((number?      exp) exp)
    ((symbol?      exp) (lookup exp env))
    ((define?      exp) (eval-define exp env))
    ((if?          exp) (eval-if exp env))
    ((lambda?      exp) (eval-lambda exp env))
    ((let?         exp) (eval-let exp env))
    ((quote?       exp) (eval-quote exp env))
    ((application? exp) (apply* (eval (car exp) env)
                                 (map (lambda (e) (eval e env))
                                      (cdr exp)))))
    (else
     (error "unknown expression" exp)))

; Here we'll do a tag check for quote* and quote. This way
; we can take advantage of Scheme's complex parsing of quotes
; using apostrophe.
(define (quote? exp)
  (or (tag-check exp 'quote*)
      (tag-check exp 'quote)))

; Handles expressions like (quote* (1 2 3))
(define (eval-quote exp env)
  (second exp))

; Quick tests
(eval '(quote* x)      GE) ; -> x
(eval ''x             GE) ; -> x
(eval '(quote* (x y z)) GE) ; -> (x y z)
```

Sugared procedure definitions

(define* (name arg arg ...) body) defines a procedure name with arguments (arg arg ...) and body body.

```
(load "quote.scm")
(display "PROCEDURE-DEFINE.SCM_" (newline))

; One of the nice things in normal Scheme is sugared define+lambda syntax
; (define (my-proc param1 param2 ... paramn) body)
; Let's allow for this syntax in our Scheme* interpreter too. To do this,
; we'll directly handle this form; we won't desugar it.

; Handles procedure defines and variable defines
(define (eval-define exp env)
  (if (procedure-define? exp)
      (eval-procedure-define exp env)
      (eval-variable-define exp env)))

; Tests if the expression is a sugared define+lambda, e.g.
; (define (foo p1 p2) ...)
(define (procedure-define? exp)
  (and (define? exp)
       (pair? (second exp))))

; This is just a renamed version of the old eval-define
(define (eval-variable-define exp env)
  (let ((name (cadr exp))
        (defined-to-be (caddr exp)))
    (table-put! (car env) name (eval defined-to-be env))
    'undefined))

; Handles evaluation of defines like:
; (define (foo p1 p2) ...)
(define (eval-procedure-define exp env)
  (let ((name (first (second exp)))
        (params (cdr (second exp)))
        (body (third exp)))
    (table-put! (car env) name (make-compound params body env))
    'undefined))

(eval '(define* x* 10) GE) ; -> undefined
(eval '(define* double* (lambda* (x) (plus* x x))) GE) ; -> undefined
(eval '(double* x*) GE) ; -> 20
(eval '(define* (double2* x) (plus* x x)) GE) ; -> undefined
(eval '(double2* x*) GE) ; -> 20
```

Who's smarter than a 360th version?

Earlier in the semester, we noted that DrScheme v360 has a “feature” allowing either of the following

```
(define (quote x) 10)
```

```
(define 'x 10)
```

but this causes the `quote` special form to get clobbered. How could we prevent the shadowing of special forms in Scheme*? If someone tries to shadow a special form, for example with

```
(define* (quote* x) 10)
```

return the symbol `reserved-word-error`.

```
(load "procedure-define.scm")
(display "FIXING-QUOTE-PROC-DEF.SCM□...") (newline)

; Handle (define* (proc-name param1 param2 ...) body)
; while disallowing the overriding of special forms.
(define (eval-procedure-define exp env)
  (let ((name (first (second exp)))
        (params (cdr (second exp)))
        (body (third exp)))
    (if (memq name '(define* if* lambda* let* quote quote* set!*))
        'reserved-word-error
        (begin
          (table-put! (car env) name (make-compound params body env))
          'undefined))))

(eval '(define* x 10) GE) ; -> undefined
(eval '(define* 'x 10) GE) ; -> reserved-word-error
(eval '(define* (quote* x) 10) GE) ; -> reserved-word-error
(eval '(define* (define* x) 10) GE) ; -> reserved-word-error
```