6.001 Recitation 10: Symbols and Quotation

RI: Gerald Dalley, dalleyg@mit.edu, 14 Mar 2007
`http://people.csail.mit.edu/dalleyg/6.001/SP2007/`

Announcements / Notes

- Happy π Day (3/14)!
- Exam: you should have received it in tutorial

Symbols and Strings

In many (nearly all, really) languages, variable names cannot be operated upon. In Scheme, we have the concept of symbols as first-class objects. Symbols are essentially references to variable names, but in Scheme we can also do the same things with symbols that we can do with anything else... we can bind them to other names, we can pass them to procedures, we can return them from procedures, *etc.*

	Strings	Symbols
Can have spaces?		
Must be legal names (can't start with a number, can't have spaces or some special characters)?		
Uses double quotes?		
What is the value of a (string/symbol)?		
How much space is required[1]?		
How long does it take to test equivalence?		

Why do we need (or want) both?

- by using symbols, we can make things that print out like valid Scheme expressions, and
- in many situations, symbols are both faster and smaller.

Scheme-ing Additions

`(quote` *expr*`)`: returns whatever the reader built for *expr*. This provides one way of producing symbols.

`(string->symbol` *str*`)`: returns a symbol whose name is *str*. This the second way of producing symbols.

`(symbol->string` *sym*`)`: returns the name of symbol *sym*.

`(eq?` *v1 v2*`)`: returns true of *v1* and *v2* are bitwise identical. "Works on" symbols, booleans, and pairs. Doesn't "work on" numbers and strings.

`(eqv?` *v1 v2*`)`: like `eq?`, except it "works on" numbers.

`(equal?` *v1 v2*`)`: returns true if *v1* and *v2* print out the same. "Works on" almost everything.

[1] ...for typical implementations of Scheme. There are many potential and actual caveats to this rule.

What's in a name: to quote or not to quote

Give printed value (or unspecified, error, or procedure where appropriate). Assume x is bound to 5.

```
'3
'x
''x
(quote (3 4))
('+ 3 4)
(if '(= x 0) 7 8)
(eq? 'x 'X)
(eq? (list 1 2) (list 1 2))
(eq? '(1 2) '(1 2))
(equal? (list 1 2) (list 1 2))
(quote 1 2 3)
(quote (list (list 1 2)))
(list (quote (list 1 2)))
```

```
(car (quote (list)))
(cadr '(+ (* 1 2) (/ 8 4)))
(map car (list '(3) '(4) '(5)))
(filter pair?
     '(a (b c d) e (f) (g h)))
(cons 'a '(b c d))
(append 'a '(b c d))
(quote '(b c d))
(cons 'a '(b))
(cons '(a) '(b))
(list 'a 'b)
(list '(a) '(b))
(append '(a) '(b))
(car ''a)
```

Revealing Membership

Write `(member elt lst)` that returns `#f` if `elt` is not in the list and returns the tail of the list starting with the first occurence of the element otherwise. `elt` might be a list!

```
(define (member elt lst)
```

How many times will this occur?

Write `(occurrences elt lst)` that returns the number of times `elt` appears in the list. `elt` might be a list!

```
(define (occurrences elt lst)
```

Chasing squirrels up a tree

A tree is a list of lists. Write `(tree-occurrences sym tree)` that returns the number of times the symbol `sym` appears in the tree. Assume that the tree only contains symbols.

```
(define (tree-occurrences sym tree)
```

Brainteasers

So far, we've claimed that `define` must be a special form. With the syntax we currently use, that's actually true. Can you think of an alternative syntax for define that would allow it to be implemented as a regular primitive procedure that is evaluated with the combination rule from the substitution model? Can `define` be a compound procedure?

Feedback

Year: Programming Experience:
Section:

1. In general, how is recitation?

   Great! Good OK I don't attend

2. Recitation pace compared to your optimal pace?

   Too fast Fast OK Slow Zzzzz

3. Problem solving balance?

   More individual problem solving OK More pontificating

4. How often have you worked out and/or looked up the solutions to problems at the end of the handout that we haven't covered in class?

   Every one! sometimes Mmmm...never

5. How many people's names will I really remember by the end of the semester?

6. Name something you dislike about recitation:

7. Name something you like about recitation

8. Any other comments / suggestions for improvement:

9. What did you think of quiz 1?

6.001 Recitation 10: Symbols and Quotation

RI: Gerald Dalley, dalleyg@mit.edu, 14 Mar 2007
http://people.csail.mit.edu/dalleyg/6.001/SP2007/

Announcements / Notes

- Happy π Day (3/14)!
- Exam: you should have received it in tutorial

Symbols and Strings

In many (nearly all, really) languages, variable names cannot be operated upon. In Scheme, we have the concept of symbols as first-class objects. Symbols are essentially references to variable names, but in Scheme we can also do the same things with symbols that we can do with anything else... we can bind them to other names, we can pass them to procedures, we can return them from procedures, *etc.*

	Strings	Symbols
Can have spaces?	yes	no[2]
Must be legal names (can't start with a number, can't have spaces or some special characters)?	no	yes
Uses double quotes?	yes	no
What is the value of a (string/symbol)?	the string itself	value bound to that symbol
How much space is required[3]?	$\Theta(n)$	$\Theta(1)$
How long does it take to test equivalence?	$\Theta(n)$	$\Theta(1)$

Why do we need (or want) both?

- by using symbols, we can make things that print out like valid Scheme expressions, and
- in many situations, symbols are both faster and smaller.

Scheme-ing Additions

`(quote` *expr*`)`: returns whatever the reader built for *expr*. This provides one way of producing symbols.

`(string->symbol` *str*`)`: returns a symbol whose name is *str*. This the second way of producing symbols.

`(symbol->string` *sym*`)`: returns the name of symbol *sym*.

`(eq?` *v1 v2*`)`: returns true of *v1* and *v2* are bitwise identical. "Works on" symbols, booleans, and pairs. Doesn't "work on" numbers and strings.

`(eqv?` *v1 v2*`)`: like `eq?`, except it "works on" numbers.

`(equal?` *v1 v2*`)`: returns true if *v1* and *v2* print out the same. "Works on" almost everything.

[2]There are actually some tricky ways in to create symbols with spaces, but we won't talk about them.
[3]...for typical implementations of Scheme. There are many potential and actual caveats to this rule.

What's in a name: to quote or not to quote

Give printed value (or unspecified, error, or procedure where appropriate). Assume x is bound to 5.

`'3`	`3`
`'x`	`x`
`''x`	`(quote x)`
`(quote (3 4))`	`(3 4)`
`('+ 3 4)`	Error: the symbol `+` is not a procedure
`(if '(= x 0) 7 8)`	`7`: the list `(= x 0)` is not `#f`
`(eq? 'x 'X)`	It depends on whether you turned off case sensitivity
`(eq? (list 1 2) (list 1 2))`	`#f`
`(eq? '(1 2) '(1 2))`	`#f` in DrScheme, but the R5RS specification says `#t` is okay too
`(equal? (list 1 2) (list 1 2))`	`#t`
`(quote 1 2 3)`	Error: `quote` takes exactly one expression
`(quote (list (list 1 2)))`	`(list (list 1 2))`
`(list (quote (list 1 2)))`	`((list 1 2))`
`(car (quote (list)))`	`list`
`(cadr '(+ (* 1 2) (/ 8 4)))`	`(* 1 2)`
`(map car (list '(3) '(4) '(5)))`	`(3 4 5)`
`(filter pair? '(a (b c d) e (f) (g h)))`	`((b c d) (f) (g h))`
`(cons 'a '(b c d))`	`(a b c d)`
`(append 'a '(b c d))`	Error: the symbol `a` is not a list
`(quote '(b c d))`	`'(b c d)`
`(cons 'a '(b))`	`(a b)`
`(cons '(a) '(b))`	`((a) b)`
`(list 'a 'b)`	`(a b)`
`(list '(a) '(b))`	`((a) (b))`
`(append '(a) '(b))`	`(a b)`
`(car ''a)`	`quote`

Revealing Membership

Write `(member elt lst)` that returns `#f` if `elt` is not in the list and returns the tail of the list starting with the first occurence of the element otherwise. `elt` might be a list!

```
(define (member elt lst)
  (cond ((null? lst) #f)
        ((equal? (car lst) elt) lst)
        (else (member elt (cdr lst)))))

(member '(apple pear) '(x (apple sauce) y apple pear))  ; => #f
(member '(apple sauce) '(x (apple sauce) y apple pear)) ; => ((apple sauce) y apple pear)
; note: member is actually a built-in procedure
```

How many times will this occur?

Write `(occurrences elt lst)` that returns the number of times `elt` appears in the list. `elt` might be a list!

```
(define (occurrences elt lst)
  ; a standalone version works...
  (cond ((null? lst)              0)
        ((equal? elt (car lst)) (+ 1 (occurrences elt (cdr lst))))
        (else                     (occurrences elt (cdr lst)))))

(occurrences 'b '(a b c d a b c d)) ; => 2
(occurrences 'e '(a b c d a b c d)) ; => 0
```

```
(occurrences '(a b c) '((a b c) ((a b c)) d a (a b c) b c d)) ; => 2

(define (occurrences elt lst)
  ; but using member makes it even easier...
  (let ((first-occurrence (member elt lst)))
    (if first-occurrence
        (+ 1 (occurrences elt (cdr first-occurrence)))
        0)))
(occurrences 'b '(a b c d a b c d)) ; => 2
(occurrences 'e '(a b c d a b c d)) ; => 0
(occurrences '(a b c) '((a b c) ((a b c)) d a (a b c) b c d)) ; => 2
```

Chasing squirrels up a tree

A tree is a list of lists. Write `(tree-occurrences sym tree)` that returns the number of times the symbol `sym` appears in the tree. Assume that the tree only contains symbols.

```
(define (tree-occurrences sym tree)
  (cond ((null? tree) 0)
        ((pair? tree) (+ (tree-occurrences sym (car tree))
                         (tree-occurrences sym (cdr tree))))
        ((eq? sym tree) 1)
        (else 0)))

(tree-occurrences 'a '((a b c) (d e f) (a b a))) ; => 3
(tree-occurrences 'a '((a b c)))                 ; => 1
(tree-occurrences 'd '((a b c)))                 ; => 0
```

Brainteasers

So far, we've claimed that `define` must be a special form. With the syntax we currently use, that's actually true. Can you think of an alternative syntax for define that would allow it to be implemented as a regular primitive procedure that is evaluated with the combination rule from the substitution model?
We could use a syntax such as `(define 'x 10)` *where the type of the* `define` *procedure is* `symbol,A -> B` *and* `B` *is some undefined type.*
Can `define` be a compound procedure?

No, unless we provided some other way of supplying the Scheme interpreter with symbol-to-value bindings.