6.001 Tutorial 5 Notes

Gerald Dalley
7–8 Mar 2005

Reminders

- Problem set due Tuesday
- Project 2 due Friday
- Grading
  - Quiz 1 = 12.5% of final grade
  - Project 1 ≈ 6% of final grade

`quote`

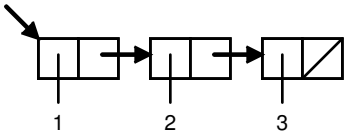
The `quote` special form just returns exactly the argument it was given – whatever the reader built.

Give a box-and-pointer diagram (as appropriate) for the results of evaluating each expression:

`(quote 1)`

1

`(quote (1 2 3))`



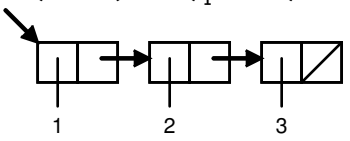
You can also use the sugared form of `quote`, `'`:

`'1`

`'1` ⇔ `(quote 1)` ⇔ 1

`'(1 2 3)`

`'(1 2 3)` ⇔ `(quote (1 2 3))` ⇔



Symbols

Symbols are another data type in Scheme. They are similar to strings but they are *interned*. When Scheme sees a symbol, it checks to see if it has ever seen a symbol with the same character string before. If it has not, it creates a new symbol. If it has, it returns a pointer to the symbol it created last time it saw the character string.

Because of this, any two symbols of the same name refer to the same *object*. Unlike strings, we can compare two symbols for equality in constant time, by checking if they point to the same object.

We use symbols for many purposes:

- Tagged data – see Tuesday's lecture
- Deferring evaluation – we'll see this later in the term.
- Giving names to values – `(define a b)` associates the value of `b` with the *symbol* `a`.

For each question, give the value and type of each expression. Assume they are evaluated in the given order.

`'3`

`3`, number

`'x`

`x`, symbol

`''x`

`(quote x)`, symbol

`(quote (3 4))`

`(3 4)`, list

`('+ 3 4)`

`error`, the `+` *symbol* is not an operator.

`(if '(= x 0) 7 8)`

`7`, number, the list `(= x 0)` is not `#f`, so true branch

Equality

We've talked about `=`, used to compare numbers for equality. We now have `eq?` and `equal?`.

`(eq? a b)` is the simplest test. It checks whether `a` and `b` are the same object (for C programmers, whether `a` and `b` are the same pointer). It works for booleans and symbols, but *not for numbers*!

```
(eq?  #t #t)
```
`#t`

```
(eq?  'foo 'foo)
```
`#t`

```
(eq?  'foo 'FOO)
```
`#t`

```
(eq?  (list 1 2 3) (list 1 2 3))
```
`#f`

`(equal? a b)` checks whether `a` and `b` print out the same and works for almost everything.

```
(equal?  #t #t)
```
`#t`

```
(equal (list 1 2 3) (list 1 2 3))
```
`#t`

What do these do?

```
(eq?  'x 'X)
```
`#t`, boolean, case insensitivity

```
(eq?  (list 1 2) (list 1 2))
```
`#f`, boolean, `list` builds a new list each time

```
(equal?  (list 1 2) (list 1 2))
```
`#t`, boolean the two lists print the same

In general, you should use = for numbers, eq? for symbols, and equals? for lists. Assuming we have evaluated the following:

```
(define lst '(1 2 3))
```

fill in the table below with `#t`, `#f`, `e` (for error), or `u` (for unspecified) if we assume that the actual text written in the `a` and `b` columns was copied and pasted into the given expressions:

a	b	(= a b)	(eq? a b)	(equal? a b)
1	1	#t	u	#t
3456789	3456789	#t	u	#t
#f	#f	e	#t	#t
"123"	"123"	e	u	#t
'a	'a	e	#t	#t
lst	lst	e	#t	#t
lst	'(1 2 3)	e	#f	#t
'(1 2 3)	'(1 2 3)	e	#f	#t
car	car	e	#t	#t

See `http://sicp.csail.mit.edu/Spring-2005/manuals/scheme-7.5.5/doc/scheme_4.html` for a complete description of the equality tests with many more examples.

Symbolic Boolean Expression Manipulation

A boolean formula is a formula containing boolean operations and boolean variables. A boolean variable is either `true` or `false`. `and`, `or`, and `not` are all boolean operations. For the purposes of this problem, `and` and `or` will be defined to take exactly two inputs.

Example formulas:

```
a
(not b)
(or b (not c))
(and (not a) (not c))
(not (or (not a) c))
(and (or a (not b)) (or (not a) c))
```

Assume that we have the following abstractions defined for boolean expressions:

- `(variable? exp)` → `boolean`
  Indicates whether the given boolean expression is a simple variable.
- `(make-variable sym)` → `exp`
  Converts a symbol into a expression. The returned expression is a variable.
- `(variable-name exp)` → `sym`
  Obtains the symbol associated with an expression. Assumes `exp` is a variable.

- `(not? exp)` → `boolean`
  Predicate for `not` expressions.
- `(make-not exp)` → `exp`
  Constructor for `not` expressions. `exp` should be an expression. It becomes the subexpression for the new `not` expression.
- `(not-operand exp)` → `exp`
  Returns the subexpression of a `not` expression.

- `(or? exp)` → `boolean`
  Predicate for `or` expressions.
- `(make-or exp1 exp2)` → `exp`
  Constructor for `or` expressions. `exp1` and `exp2` should be expressions. They become the two subexpressions for the new `or` expression.
- `(or-first exp)` → `exp`
  Returns the first subexpression of an `or` expression.
- `(or-second exp)` → `exp`
  Returns the second subexpression of an `or` expression.

- `(and? exp)` → `boolean`
  Predicate for `and` expressions.
- `(make-and exp1 exp2)` → `exp`
  Constructor for `and` expressions. `exp1` and `exp2` should be expressions. They become the two subexpressions for the new `and` expression.
- `(and-first exp)` → `exp`
  Returns the first subexpression of an `and` expression.
- `(and-second exp)` → `exp`
  Returns the second subexpression of an `and` expression.

For those who are curious, a simple implementation of these methods are given in this week's tutorial solutions.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Definitions for Dr. Scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Semi-standard functions and values

(define nil '())
(define (fold-right op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (fold-right op init (cdr lst)))))
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst))
         (cons (car lst)
               (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Error checking helpers
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (assert test-val pred)
  (if (pred test-val)
      #t
      (error "ASSERT FAILED: "
             test-val pred)))
(define (assert-equal test-val expected-val)
  (assert test-val
          (lambda (t)
            (equal? t expected-val))))
(define (assert-f test-val)
  (assert-equal test-val #f))
(define (assert-t test-val)
  (assert-equal test-val #t))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Boolean expressions abstraction
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Determines whether a given object is
;; a boolean expression
(define (expression? exp)
  (or (variable? exp) (not? exp)
      (or? exp)       (and? exp)))

;; Tells whether the given expression is
```

```
;; a simple variable
(define (variable? exp)
  (and (pair? exp) (eq? (car exp) 'var)))
;; Turns a symbol into a variable for a
;; boolean expression
(define (make-variable sym)
  (assert sym symbol?)
  (list 'var sym))
;; Returns the symbol associated with
;; the variable.
(define (variable-name exp)
  (assert exp variable?)
  (cadr exp))

;; Tells whether the given expression is
;; a not expression
(define (not? exp)
  (and (pair? exp) (eq? (car exp) 'not)))
;; Creates a not expression from a
;; subexpression
(define (make-not exp)
  (assert exp expression?)
  (list 'not exp))
;; Returns the subexpression associated
;; with a not expression
(define (not-operand exp)
  (assert exp not?)
  (cadr exp))

;; Tells whether a given expression is
;; an or expression.
(define (or? exp)
  (and (pair? exp) (eq? (car exp) 'or)))
;; Creates an or expression of two
;; subexpressions
(define (make-or exp1 exp2)
  (assert exp1 expression?)
  (assert exp1 expression?)
  (list 'or exp1 exp2))
;; Returns the first subexpression of an
;; or expression
(define (or-first exp)
  (assert exp or?)
  (cadr exp))
;; Returns the second subexpression of an
;; or expression
(define (or-second exp)
  (assert exp or?)
  (caddr exp))

;; Tells whether a given expression is
;; an and expression.
(define (and? exp)
  (and (pair? exp) (eq? (car exp) 'and)))
;; Creates an and expression of two
;; subexpressions
(define (make-and exp1 exp2)
  (assert exp1 expression?)
  (assert exp1 expression?)
  (list 'and exp1 exp2))
;; Returns the first subexpression of an
;; and expression
(define (and-first exp)
  (assert exp and?)
  (cadr exp))
```

```
;; Returns the second subexpression of an
;; and expression
(define (and-second exp)
  (assert exp and?)
  (caddr exp))
```

Assume that we also have available the following abstraction for a set, from last week:

- `empty-set`
  The special value for the empty set.
- `(set-contains? elm set)` → `boolean`
  Indicates whether the set already contains an element.
- `(set-add elm set)` → `set`
  Returns a new set that contains all elements of the old set and the given element.
- `(set-remove elm set)` → `set`
  Returns a new set that contains all elements in `set` except `elm`.
- `(set-intersection s1 s2)` → `set`
  Returns a new set that contains only the elements that are in `s1` and `s2`.
- `(set-union s1 s2)` → `set`
  Returns a new set that contains only the elements that are in either `s1` and/or `s2`.

An implementation of the `set` abstraction:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Definitions for Dr. Scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Semi-standard functions and values

(define nil '())
(define (fold-right op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (fold-right op init (cdr lst)))))
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst))
         (cons (car lst)
               (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Error checking helpers
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (assert test-val pred)
  (if (pred test-val)
      #t
      (error "ASSERT FAILED: " test-val pred)))
(define (assert-equal test-val expected-val)
  (assert test-val (lambda (t) (equal? t expected-val))))
(define (assert-f test-val)
  (assert-equal test-val #f))
(define (assert-t test-val)
  (assert-equal test-val #t))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Set abstraction
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Creates a new set with no elements
(define (new-set) '(set))

;; Determines whether the given object is a set
(define (set? set)
  (and (list? set)
       (not (null? set))
       (eq? (car set) 'set)))

(assert (new-set) set?)

;; Generates an error if the given object is not a set.
(define (set-check set)
  (if (set? set) #t (error "not a set")))

(set-check (new-set))

;; Returns a list containing all of the elements of
;; a set.
(define (set-to-list set)
  (set-check set)
  (cdr set))

(assert-equal (set-to-list (new-set)) nil)

;; Determines whether elm is contained in set using
```

```
;; equal? as a comparator
(define (set-contains? elm set)
  (set-check set)
  (fold-right (lambda (val acc) (or acc (equal? val elm)))
              #f
              (set-to-list set)))

(assert-f (set-contains? 123 (new-set)))

;; Returns a new set containing all elements in the original
;; plus the given element.  If the given element is already
;; present in the set, the new set is equivalent to the old
;; one.
(define (set-add elm set)
  (set-check set)
  (if (set-contains? elm set)
      set
      (append (list 'set)
              (cons elm (set-to-list set)))))

(assert-equal (set-to-list (set-add 123 (new-set))) '(123))
(define s (set-add 3 (set-add 2 (set-add 1 (new-set)))))
(assert-t (set-contains? 1 s))
(assert-t (set-contains? 2 s))
(assert-t (set-contains? 3 s))
(assert-f (set-contains? 4 s))

;; Creates a set given a list of elements
(define (list-to-set lst)
  (fold-right set-add (new-set) lst))

(assert-t (set-contains? 1 (list-to-set '(1 2 3 3))))
(assert-t (set-contains? 2 (list-to-set '(1 2 3 3))))
(assert-t (set-contains? 3 (list-to-set '(1 2 3 3))))
(assert-f (set-contains? 4 (list-to-set '(1 2 3 3))))
(assert-equal (length (set-to-list (list-to-set '(1 2 3 3)))) 3)

;; Returns a new set containing the same elements as
;; the original, except for elm.  If elm was not
;; present in the original, the new set is equivalent
;; to the old one.
(define (set-remove elm set)
  (set-check set)
  (list-to-set (filter (lambda (x) (not (equal? elm x)))
                       (set-to-list set))))

(assert-equal (length (set-to-list (set-remove 3 (list-to-set '(1 2 3 3 4 5))))) 4)
(assert-equal (length (set-to-list (set-remove 3 (set-remove 3 (list-to-set
  '(1 2 3 3 4 5)))))) 4)
(assert-equal (length (set-to-list (set-remove 1 (set-remove 2 (list-to-set
  '(1 2 3 3 4 5)))))) 3)
(assert-equal (length (set-to-list (set-remove 1 (set-remove 2 (set-remove 3
  (set-remove 4 (set-remove 5 (list-to-set '(1 2 3 3 4 5)))))))))) 0)

;; Determines whether set 1 and set 2 contain the same
;; elements.
(define (set-equal? s1 s2)
  (set-check s1) (set-check s2)
  (let* ((lst1 (set-to-list s1))
         (l1 (length lst1))
         (l2 (length (set-to-list s2))))
    (if (not (= l1 l2))
        #f
        (fold-right (lambda (val acc) (and acc (set-contains? val s2)))
                    #t
```

```
                    lst1))))

(assert-t (set-equal? (list-to-set (list 1 2 3)) (list-to-set (list 1 2 3))))
(assert-t (set-equal? (list-to-set (list 1 2 3)) (list-to-set (list 3 2 1))))
(assert-f (set-equal? (list-to-set (list 1 2 3)) (list-to-set (list 1 2 3 4))))
(assert-f (set-equal? (list-to-set (list 1 2 3)) (list-to-set (list 1 2))))
(assert-f (set-equal? (list-to-set (list )) (list-to-set (list 1 2 3))))
(assert-t (set-equal? (new-set) (new-set)))

;; Creates a new set containing all elements in either
;; s1 and/or in set s2.
(define (set-union s1 s2)
  (set-check s1) (set-check s2)
  (fold-right set-add s2 (set-to-list s1)))

(assert-t (set-equal? (set-union (new-set) (new-set)) (new-set)))
(assert-t (set-equal? (set-union (new-set) (list-to-set '(1 1 2))) (list-to-set '(2 1))))
(assert-t (set-equal? (set-union (list-to-set '(1 2 3)) (list-to-set '(1 1 2)))
                      (list-to-set '(1 2 3))))

;; Creates a new set containing only those elements
;; present in both s1 and s2.
(define (set-intersection s1 s2)
  (list-to-set (filter (lambda (elm) (set-contains? elm s2)) (set-to-list s1))))

(define s1 (new-set))
(define s2 (list-to-set (list 1 1 2)))
(define s3 (list-to-set (list 1 2 3)))
(define s4 (list-to-set (list 2 3 4)))
(define s5 (list-to-set (list 7 8 9)))
(assert-t (set-equal? (set-intersection s1 s1) (new-set)))
(assert-t (set-equal? (set-intersection s1 s2) (new-set)))
(assert-t (set-equal? (set-intersection s2 s2) s2))
(assert-t (set-equal? (set-intersection s2 s3) (list-to-set '(1 2))))
(assert-t (set-equal? (set-intersection s3 s4) (list-to-set '(2 3))))
(assert-t (set-equal? (set-intersection s2 s5) (new-set)))

;; Creates a new set containing only those items
;; in s1 and not in set s2.
(define (set-diff s1 s2)
  (list-to-set (filter (lambda (elm) (not (set-contains? elm s2)))
                       (set-to-list s1))))
(assert-t (set-equal? (set-diff s1 s1) (new-set)))
(assert-t (set-equal? (set-diff s2 s1) s2))
(assert-t (set-equal? (set-diff s3 s2) (list-to-set '(3))))
(assert-t (set-equal? (set-diff s4 s2) (list-to-set '(4 3))))
(assert-t (set-equal? (set-diff s5 s2) s5))
```

Given a formula, we'd like to be able to tell which variables it involves. `formula-variables` should return the `set` of variables used in the formula.

```
;; Returns the set of variables used in the given expression.
(define (formula-variables exp)
 (cond ((variable? exp)
        (set-add (variable-name exp) (new-set)))
       ((not? exp)
        (formula-variables (not-operand exp)))
       ((or? exp)
        (set-union (formula-variables (or-first exp))
                   (formula-variables (or-second exp))))
       ((and? exp)
        (set-union (formula-variables (and-first exp))
                   (formula-variables (and-second exp))))
       (else (error "unknown exp" exp))))
```

```
; Tests
(define va (make-variable 'a))
(define vb (make-variable 'b))
(define vc (make-variable 'c))

(define e1 va)
(define e2 (make-not vb))
(define e3 (make-and va vb))
(define e4 (make-or va vb))
(define e5 (make-and (make-not va) (make-not vc)))
(define e6 (make-and (make-not va) va))
(define e7 (make-and (make-or va (make-not vb)) (make-or (make-not va) vc)))

(assert-t (set-equal? (formula-variables e1) (list-to-set '(a))))
(assert-t (set-equal? (formula-variables e2) (list-to-set (list 'b))))
(assert-t (set-equal? (formula-variables e3) (list-to-set '(a b))))
(assert-t (set-equal? (formula-variables e4) (list-to-set '(b a))))
(assert-t (set-equal? (formula-variables e5) (list-to-set '(a c))))
(assert-t (set-equal? (formula-variables e6) (list-to-set '(a))))
(assert-t (set-equal? (formula-variables e7) (list-to-set '(a b c))))
```

Now, assume that we have a `set` of variable-value bindings. The elements of this list will be two-item lists, where the first item is the variable name, and the second element is the variable's value.

Write a procedure that takes a boolean expression and a set of bindings and determines the value of the expression given the bindings. For example,

```
(define va (make-variable 'a))
(define vb (make-variable 'b))
(define bindings (set-add '(a #t) (set-add '(b #f) empty-set)))
(formula-value bindings (make-and va vb)) ; --> #f
(formula-value bindings (make-or va vb)) ; --> #t
(formula-value bindings (make-and va (not vb))) ; --> #t
(formula-value bindings va) ; --> #t
```

```
;; Returns the value of the boolean expression exp given
;; the variable bindings.
(define (formula-value bindings exp)
  (cond ((variable? exp)
         (variable-value bindings (variable-name exp)))
        ((not? exp)
         (not (formula-value bindings (not-operand exp))))
        ((or? exp)
         (or (formula-value bindings (or-first exp))
             (formula-value bindings (or-second exp))))
        ((and? exp)
         (and (formula-value bindings (and-first exp))
              (formula-value bindings (and-second exp))))
        (else (error "unknown exp" exp))))

;; Looks up the variable's value in a set of bindings.
;; Note that a better way to do this would be to use
;; an associative data structure to store the bindings,
;; but we haven't seen them yet.
(define (variable-value bindings varname)
  (set-check bindings)
  (assert-equal (length (filter (lambda (x) (eq? (first x) varname))
                                (set-to-list bindings)))
                1)
  (fold-right (lambda (val acc) (if (eq? (first val) varname)
                                    (second val)
                                    acc))
              nil
              (set-to-list bindings)))
```

```
; Tests

(define va (make-variable 'a))
(define vb (make-variable 'b))
(define vc (make-variable 'c))

(define e1 va)
(define e2 (make-not vb))
(define e3 (make-and va vb))
(define e4 (make-or va vb))
(define e5 (make-and (make-not va) (make-not vc)))
(define e6 (make-and (make-not va) va))
(define e7 (make-and (make-or va (make-not vb)) (make-or (make-not va) vc)))

(define b (list-to-set '((a #t) (b #f) (c #f))))

(assert-t (formula-value b e1))
(assert-t (formula-value b e2))
(assert-f (formula-value b e3))
(assert-t (formula-value b e4))
(assert-f (formula-value b e5))
(assert-f (formula-value b e6))
(assert-f (formula-value b e7))
```

SAT

We'll now try take a sneak peak at a classical problem in complexity theory. As far as anyone knows, there is no algorithm for solving this problem with a polynomial (sub-exponential) order of growth. If you can prove whether this problem is doable (or provably not doable) in polynomial time, fame, fortune, and riches will be heaped upon you. For now, we'll just design a slow exponential time algorithm...

We start with some boolean expression. We want to see if there is any way to assign boolean values to each of the variables such that the entire expression is true. For example,

```
(define va (make-variable 'a))
(define vb (make-variable 'b))
(sat (make-and va vb)) ; --> #t
  ; If a and b are both assigned #t, then (and a b) is true
(sat (make-and va (make-not va))) ; --> #f
  ; No assignment of a will make (and a (not a)) true.
```

Implement the `sat` procedure. It should indicate whether there is any assignment to all the formula variables that results in the entire expression evaluating to `#t`.

```
;; Returns #t if there is some assignment of boolean values to
;; each of the formula variables in exp such that the entire
;; boolean expression evaluates to true.
(define (sat exp)
  ; Recursively takes one variable out of unassigned-vars and
  ; tries both possible bindings (#t and #f).  bindings is a set
  ; of bindings, where a binding is a 2-element list whose first
  ; element is the variable names and whose second element is the
  ; bound boolean value.
  (define (assign-vars unassigned-vars bindings)
    (if (null? unassigned-vars)
        ; If all variables have been assigned, see if the expression
        ; returns #t
        (formula-value bindings exp)
        ; If there are unassigned variables, try both possible assignments
        ; of the first remaining variable.  Recurse.
        (or (assign-vars (cdr unassigned-vars)
                         (set-add (list (car unassigned-vars) #t) bindings))
            (assign-vars (cdr unassigned-vars)
                         (set-add (list (car unassigned-vars) #f) bindings)))))
```

```
  (assign-vars (set-to-list (formula-variables exp)) (new-set)))

; tests
(define va (make-variable 'a))
(define vb (make-variable 'b))
(define vc (make-variable 'c))

(define e1 va)
(define e2 (make-not vb))
(define e3 (make-and va vb))
(define e4 (make-or va vb))
(define e5 (make-and (make-not va) (make-not vc)))
(define e6 (make-and (make-not va) va))
(define e7 (make-and (make-or va (make-not vb)) (make-or (make-not va) vc)))

(assert-t (sat e1))
(assert-t (sat e2))
(assert-t (sat e3))
(assert-t (sat e4))
(assert-t (sat e5))
(assert-f (sat e6))
(assert-t (sat e7))
```

Feedback

1. If you had to choose one thing to change about tutorial, what would it be?

2. Are the examples good? Bad? Unintelligible? Greek? Too easy? Too hard?

3. How are explanations in tutorial? Are there things that I do a poor job of explaining?

4. Are there things that you would like to see online that would be useful?
   (As a reminder — `http://people.csail.mit.edu/~dalleyg/6.001/`)

5. Are there things I could do to make 6.001 more pleasant? Preemptively — I can't get rid of projects, problem sets, quizzes, ... Sorry.

6. Any other comments?

Thanks for your feedback!