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)
```

```
(quote (1 2 3))
```

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

```
'1
```

```
'(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
```

```
'x
```

```
''x
```

```
(quote (3 4))
```

```
('+ 3 4)
```

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

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)

(eq?  'foo 'foo)

(eq?  'foo 'FOO)

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

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

```
(equal?  #t #t)

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

What do these do?

```
(eq?  'x 'X)

(eq?  (list 1 2) (list 1 2))

(equal?  (list 1 2) (list 1 2))
```

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			
3456789	3456789			
#f	#f			
"123"	"123"			
'a	'a			
lst	lst			
lst	'(1 2 3)			
'(1 2 3)	'(1 2 3)			
car	car			

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.

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`.

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)))
```

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)
```

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)
```

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!