Programming Abstractions

Lecture 3: Expression evaluation, conditionals, and equality

Stephen Checkoway

Expression evaluation

Expression evaluation

Scheme evaluates s-expressions to produce values

- The value of `'()` is `'()`
- The value of a variable is the value bound to it
  - E.g., the variable `null` is bound to `'()`
- The value of an atom is the atom itself
- The value of a non-null list depends on the head of the list
  - If the head is one of a specific set of symbols (e.g., define, lambda, λ, and let), it's a *special form*. Each special form has its own way of being evaluated
  - Otherwise, it's procedure application

Procedure evaluation

`(foo 1 2 #t)` applies the procedure bound to the variable `foo` to the arguments `1`, `2`, and `#t`

- `(+ 1 2 3)` — applies + to `1`, `2`, and `3`, performing addition
- `(* 5 (- x y) (/ z 8))` — computes 5(*x* - *y*)(*z* / 8)
- `(list 32 5 8)` — creates the list `'(32 5 8)`
- `(list-ref (list 32 5 8) 2)` — returns the element of `'(32 5 8)` at index `2` namely `8`

Note that `(1 2 3)` is invalid because `1` isn't a special form nor is it a procedure

Procedure evaluation order

```
(s-exp0 s-exp2 ... s-expn)
```

Scheme evaluates each of the s-expressions in turn

- `s-exp0` must evaluate to a procedure value
- `s-exp1` through `s-expn` are evaluated to produce values
- Then, the procedure is applied to the *n* arguments

```
(+ (* 2 3) 8)
```

- `+` evaluates to the addition procedure
- `(* 2 3)` is evaluated
  - `*` evaluates to the multiplication procedure
  - `2` and `3` evaluate to themselves
  - multiplication procedure is applied to `2` and `3`, producing `6`
- `8` evaluates to itself
- addition procedure is applied to 6 and 8, producing 14

Using quote ' to prevent evaluation

`(quote (1 2 3))` evaluates to the list `(1 2 3)`

`'(1 2 3)` is shorthand for this and is how DrRacket will display lists

`'()` is null and how DrRacket displays the empty list

We can quote identifiers (e.g., variable names) and use them as symbols rather than the value the identifier is bound to (if any)

- `'red`, `'green`, `'blue`, `'+`, `'list`, etc

E.g., `(+ 1 2)` evaluates to `3`, `'(+ 1 2)` evaluates to the list `'(+ 1 2)`

```
(define (p x y)
  (printf "x: ~v\ny: ~v\n" x y))
```

Procedure `p` prints out its two arguments. E.g., `(p "foo" '(3 2))` prints

```
x: "foo"
y: '(3 2)
```

What does `(p (+ 3 5)`
`(list (- 2 1) '(- 2 1)))` print?

A.
```
x: (+ 3 5)
y: (list (- 2 1) '(- 2 1))
```

B.
```
x: 8
y: '((- 2 1) '(- 2 1))
```

C.
```
x: 8
y: '(1 1)
```

D.
```
x: 8
y: '(1 (- 2 1))
```

Conditionals

If expression

`(if test-exp then-exp else-exp)`

If `test-exp` evaluates to anything other than `#f`, then the whole expression evaluates to the evaluation of `then-exp`

If `test-exp` evaluates to `#f`, then the whole if expression evaluates to the evaluation of `else-exp`

Examples

-
```
(if (= x y)
    (+ x 2)
    y)
```
-
```
(if (empty? lst)
    "The list is empty"
    "The list is not empty")
```

Predicates

Racket has a bunch of procedures that return #t if its argument satisfies some property

- `(zero? x)`, `(positive? x)`, `(negative? x)`
- `(empty? x)` returns `#t` if `x` is the empty list
- `(list? x)` returns `#t` if `x` is a list
- `(number? x)` returns `#t` if `x` is a number
- `(pair? x)` returns `#t` if `x` is a pair (including a nonempty list)
- `(symbol? x)`
- `(string? x)`

Conditional expressions

`(cond [test-exp1 exp1] … [test-expn expn])`

Evaluates the `test-exp` expressions in turn

The first one that evaluates to something other than `#f` has its corresponding `exp` evaluated which becomes the value of the whole expression

We can use `else` as the last test expression

```
(cond [(zero? x) 0]
      [(> x 0) 1]
      [else -1])
```

Some examples

```
(define (signum x)
  (cond [(zero? x) 0]
        [(> x 0) 1]
        [else -1]))

(define (length lst)
  (cond [(empty? lst) 0]
        [else (+ 1 (length (rest lst)))]))
```

Recursion!

There are no loops in Scheme; instead, we use recursion *everywhere*!

Common pattern when working with lists

```
(define (sum-positives lst)
  (cond [(empty? lst) 0] ; If lst is empty, return 0
        [(> (first lst) 0)
         (+ (first lst) (sum-positives (rest lst)))] ; recursion
        [else (sum-positives (rest lst))])) ; recursion
```

Note

- List functions `empty?, first, rest`
- Base case 0
- Recursive calls using the `rest` of the list, combined with the `first` element

What does this procedure do?

```
(define (foo lst)
  (cond [(empty? lst) #t]
        [(zero? (first lst)) #f]
        [else (foo (rest lst))]))
```

A. Returns `#t` if `lst` is empty and `#f` otherwise

B. Returns `#t` if `lst` contains a `0` and `#f` otherwise

C. Returns `#f` if `lst` contains a `0` and `#t` otherwise

D. Runs forever because `foo` is called on the rest of `lst`

```
(define (fun lst1 lst2)
  (cond [(empty? lst1) lst2]
        [else (cons (first lst1)
                    (fun (rest lst1) lst2))]))
```

What is the result of `(fun '(1 2 3 4) '(a b c))`?

A. `'(1 2 3 4 a b c)`

B. `'(4 3 2 1 a b c)`

C. `'(1 2 3 4 c b a)`

D. `'(4 3 2 1 c b a)`

E. `'(a b c)`

Equality

Scheme's equality operators

`=`, `eq?`, `eqv?`, and `equal?`

`(= a b)` — compares only numbers, cannot be used for anything else

`(equal? a b)` — compares structures recursively

- you almost always want this one and not either of the two below

`(eq? a b)` — compares if a and b refers to the same object in memory

- This can be used on symbols `(eq? 'foo 'foo)` returns `#t`, but is otherwise not good
- `(eq? 2.0 (+ 1.0 1.0))` can (and in DrRacket does) return `#f`!

`(eqv? a b)` — like `eq?` but also works with characters and numbers

Testing for equality

Moral

Are you dealing with numbers? Use =

Are you dealing with anything else? Use `equal?`

(You can use `eq?` or `eqv?` with symbols like `(eq? sym 'foo)` to determine if the variable `sym` has as its value the symbol `'foo`)

Let's write some Racket!

`(remove-numbers lst)` — Remove all of the numbers from lst

- `(remove-numbers '(foo 3 5 (6 8))) => '(foo (6 8))`
- We can use `(number? x)` to test if `x` is a number

`(filter pred lst)` — Takes a predicate (i.e., `(pred x)` returns `#t` or `#f`) and a list and returns a list consisting only of the elements satisfying the predicate

- `(filter (λ (x) (> x 0)) '(1 2 -8 3 0 -4)) => '(1 2 3)`
- `(filter number? '(a 1 b 2 c 3)) => '(1 2 3)`