

Computer Science I

Professor Tom Ellman
Lecture 4

Types of Values in Scheme

- Numbers: 1, 2, 17, 3.14159.
- Symbols: bill, hillary, al, tipper.
- Lists: (bill hillary al tipper), (bill hillary chelsea).
- Booleans: #t, #f.
- Strings: "To be or not to be.", "Vassar College".
- Procedures: #<procedure:+>, #<procedure:*>.

Boolean Values

- Sometimes we need to ask Scheme a simple TRUE / FALSE question:
 - "Is $3*3 + 4*4$ equal to $5*5$?"
 - "Is Superman first on our list of heroes?"
- Scheme answers TRUE / FALSE questions by returning a boolean value:
 - Scheme uses the "#t" to represent TRUE.
 - Scheme uses the "#f" to represent FALSE.

Welcome to DrScheme

```
> (define square3 (* 3 3))  
> (define square4 (* 4 4))  
> (define square5 (* 5 5))  
  
> (= square5 (+ square3 square4))  
#t
```

Welcome to DrScheme

```
> (define heroes '(hulk batman superman))  
  
> (equal? 'superman (car heroes))  
#f
```

The Boolean Values: #t and #f

Welcome to DrScheme

```
> #t  
#t  
> #f  
#f
```

- The values #t and #f evaluate to themselves.
- They do not need to be quoted.

Predicates

- A predicate is a procedure that returns a boolean value.
- A predicate is used to find the answer to a TRUE / FALSE question.
- Many (but not all) predicates have question marks at the ends of their names.

Predicates for Testing Equality

- Scheme has several different predicates for testing whether two data items are the same.
- The predicate “EQUAL?” is the most widely applicable equality predicate.
- It can test equality of all types of data.
- Other equality predicates can be more efficient, but they work only for special types of data.

```
Welcome to DrScheme
> (equal? 'foo 'foo)
#t
> (equal? 'foo 'bar)
#f
> (define fred 'foo)
> (define ethel 'bar)
> (equal? fred fred)
#t
> (equal? fred ethel)
#f
> (equal? fred 'fred)
#f
```

```
Welcome to DrScheme
> (equal? (+ 9 16) (+ 20 5))
#t
> (equal? (+ 9 16) (+ 30 5))
#f
> (equal? '(superman) (cons 'superman '()))
#t
> (equal? '(hulk batman) '(hulk batman))
#t
> (equal? '(hulk batman) '(batman hulk))
#f
```

Numeric Equality and Inequality

- Some Scheme predicates are designed only for comparing numeric data.
- They work when their arguments are numbers.
- They result in errors when applied to other types of data.

Numeric Equality

```
Welcome to DrScheme
> (= (+ 9 16) (+ 20 5))
#t
> (= (+ 9 16) (+ 30 5))
#f
> (= 'foo 'foo)
=: expects type <number> as 2nd argument,
given: foo; other arguments were: foo
```

Numeric Inequality

```
Welcome to DrScheme      Welcome to DrScheme
>(> 7 7)                 >(< 7 7)
#f                        #f
>(>= 7 7)                >(<= 7 7)
#t                        #t
>(> (+ 2000 4) 2002)     >(< (+ 2000 4) 2002)
#t                        #f
>(> 2002 (+ 2000 4))    >(< 2002 (+ 2000 4))
#f                        #t
```

Why are Predicates Useful?

- Sometimes we simply want to ask a TRUE or FALSE question.
- More often we need to ask a TRUE or FALSE question in order to decide how to solve a problem.
- We use a predicate in a boolean valued expression inside a Scheme program.

Finding the Larger of Two Numbers

```
(define maximum (lambda (x y) (if (>= x y) x y)))
```

```
Welcome to DrScheme
> (maximum 17 23)
23
> (maximum (- 36 3) (- 10 13))
33
```

Finding the Smaller of Two Numbers

```
(define minimum (lambda (x y) (if (<= x y) x y)))
```

```
Welcome to DrScheme
> (minimum 17 23)
17
> (maximum (- 36 3) (- 10 13))
-3
```

The “IF” Special Form

```
(if <condition> <consequent> <alternative>)
```

- The keyword “IF” indicates to Scheme that this whole expression is a special form.
- The condition is normally a boolean-valued expression.
- First Scheme evaluates <condition> to obtain a value of #t or #f.
- If the boolean value is #t, Scheme evaluates <consequent> and returns the result.
- If the boolean value is #f, Scheme evaluates <alternative> and returns the result.

Why is “(IF …)” Considered a Special Form?

- Scheme always evaluates the condition.
- Depending on the value of the condition Scheme evaluates either the consequent or the alternative, but not both.
- Suppose “IF” were the name of a procedure?
- We would expect Scheme to evaluate all the arguments, i.e., Scheme would evaluate both the consequent and the alternative.

Why Does This Matter?

```
Welcome to DrScheme
> (define foo (lambda (x) (if (= x 0) x (/ 1 x))))
> (foo 0)
0
```

- Suppose “IF” were the name of a procedure.
- An attempt to evaluate (foo 0) would lead to a division by zero error.

Type Predicates

- Used to test the type of a data object.
- For each type of data, we have a corresponding type predicate.

“NUMBER?”

```
Welcome to DrScheme
> (number? 7)
#t
> (number? (- 10 5))
#t
> (number? 'fred)
#f
> (number? '(batman superman))
#f
```

“SYMBOL?”

```
Welcome to DrScheme
> (symbol? 'fred)
#t
> (symbol? (car '(batman superman)))
#t
> (symbol? 7)
#f
> (symbol? '(batman superman))
#f
```

“PAIR?”

```
Welcome to DrScheme
> (pair? '(batman superman))
#t
> (pair? (cons 'superman '()))
#t
> (pair? '())
#f
> (pair? 'fred)
#f
> (pair? 7)
#f
```

What is the “PAIR?” Predicate?

- PAIR? returns #t if its argument was created using CONS.
- PAIR? also returns #t if its argument is a quoted list that could have been created using CONS.
- Otherwise, PAIR? returns #f.

“NULL?”

- NULL? returns #t if its argument is the empty list.
- NULL? returns #f otherwise.

```
Welcome to DrScheme
> (define empty-list '())

> (null? empty-list)
#t
> (null? '())
#t
```

```
Welcome to DrScheme
> (null? '(superman))
#f
> (null? 'fred)
#f
> (null? 0)
#f
```

Using “PAIR?” and “NULL?” to Define “LIST?”

```
(define list? (lambda (x)
  (if (null? x)
      #t
      (if (pair? x) #t #f))))
```

Redundant!!

The expression: (if (pair? x) #t #f)
... can be simplified to: (pair? x)

Using “PAIR?” and “NULL?” to Define “LIST?”

```
(define list? (lambda (x)
  (if (null? x)
      #t
      (pair? x))))
```

```
Welcome to DrScheme
> (list? '())
#t
> (list? '(batman superman))
#t
> (list? (cons 'batman (cons 'superman '())))
#t
> (list? 'fred)
#f
> (list? 7)
#f
```

The “PROCEDURE?” Predicate

```
Welcome to DrScheme
> (procedure? car)
#t
> (procedure? (lambda (x) (* x x)))
#t
> (procedure? 'car)
#f
> (procedure? '(lambda (x) (* x x)))
#f
```

The “AND” Special Form

(and <condition1> <condition2> ... <conditionN>)

- AND takes any number of conditions, each of which is normally a boolean expression.
- Scheme evaluates the conditions in order.
- If any condition evaluates to #f, then Scheme immediately returns #f as the value of the entire AND expression.
- If all conditions evaluate to #t, then Scheme returns #t as the value of the entire AND expression.

The “OR” Special Form

(OR <condition1> <condition2> ... <conditionN>)

- OR takes any number of conditions, each of which is normally a boolean expression.
- Scheme evaluates the conditions in order.
- If any condition evaluates to #t, then Scheme immediately returns #t as the value of the entire OR expression.
- If all conditions evaluate to #f, then Scheme returns #f as the value of the entire OR expression.

Welcome to DrScheme

```
> (and #f #f)
#f
> (and #f #t)
#f
> (and #t #f)
#f
> (and #t #t)
#t
> (and (< 5 7.5) (< 7.5 10))
#t
> (and (< 5 7.5) (< 7.5 5))
#f
```

Welcome to DrScheme

```
> (or #f #f)
#f
> (or #f #t)
#t
> (or #t #f)
#t
> (or #t #t)
#t
> (or (< 5 7.5) (< 7.5 10))
#t
> (or (< 5 7.5) (< 7.5 5))
#t
```

The “NOT” Predicate

(not <boolean-expression>)

- NOT takes one argument, which is normally a boolean expression.
- NOT returns #t if its argument evaluates to #f.
- NOT returns #f if its argument evaluates to #t.

Welcome to DrScheme

```
> (not #t)
#f
> (not #f)
#t
> (not (not #t))
#t
> (not (not #f))
#f
> (equal? 'superman) (cons 'superman '()))
#t
> (not (equal? 'superman) (cons 'superman '())))
#f
```

A Better Way of Using “PAIR?” and “NULL?” to Define “LIST?”

```
(define list?
  (lambda (x)
    (or (null? x) (pair? x))))
```

Class Time

```
(define class-time?
  (lambda (time) (and (>= time 1000) (<= time 1115))))

Welcome to DrScheme
> (class-time? 930)
#f
> (class-time? 1045)
#t
> (class-time? 1120)
#f
```

Class Time

```
(define class-time?
  (lambda (time) (if (>= time 1000)
                    (if (<= time 1115) #t #f)
                    #f)))

Welcome to DrScheme
> (class-time? 930)
#f
> (class-time? 1045)
#t
> (class-time? 1120)
#f
```

Class Day

```
(define class-day?
  (lambda (day) (or (equal? day 'tuesday)
                   (equal? day 'thursday))))

Welcome to DrScheme
> (class-day? 'monday)
#f
> (class-time? 'tuesday)
#t
> (class-time? 'thursday)
#t
```

Class Day

```
(define class-day?
  (lambda (day) (if (equal? day 'tuesday)
                   #t
                   (if (equal? day 'thursday)
                       #t
                       #f))))

Welcome to DrScheme
> (class-day? 'monday)
#f
> (class-time? 'tuesday)
#t
> (class-time? 'thursday)
#t
```

Lists of Specific Lengths

```
(define singleton? (lambda (x) (and (pair? x) (null? (cdr x)))))

(define doubleton? (lambda (x) (and (pair? x) (singleton? (cdr x)))))

(define tripleton? (lambda (x) (and (pair? x) (doubleton? (cdr x)))))
```

Ordered Lists

```
(define increasing-singleton?
  (lambda (x) (and (singleton? x) (number? (car x)))))

(define increasing-doubleton?
  (lambda (x) (and (doubleton? x)
                  (number? (car x))
                  (increasing-singleton? (cdr x))
                  (< (car x) (car (cdr x))))))

(define increasing-tripleton?
  (lambda (x) (and (tripleton? x)
                  (number? (car x))
                  (increasing-doubleton? (cdr x))
                  (< (car x) (car (cdr x))))))
```

Defining the TYPE-OF Function

- TYPE-OF will take any data as its argument.
- TYPE-OF will return a symbol.
- The symbol will indicate the data type of the argument.

```
(define type-of
  (lambda (item) (if (null? item)           OK, ... but there must
                    'empty-list           be a better way!
                    (if (pair? item)
                        'pair
                        (if (number? item)
                            'number
                            (if (symbol? item)
                                'symbol
                                (if (procedure? item)
                                    'procedure
                                    'some-other-type))))))))
```

A Better Way ...

```
(define type-of
  (lambda (item)
    (cond ((null? item) 'empty-list)
          ((pair? item) 'pair)
          ((number? item) 'number)
          ((symbol? item) 'symbol)
          ((procedure? item) 'procedure)
          (else 'some-other-type))))
```

The “COND” Special Form

```
(cond (<condition1> <consequent1>)
      (<condition2> <consequent2>)
      ...
      (<conditionN> <consequentN>)
      (else <alternative>))
```

- Scheme evaluates the conditions in order.
- As soon as Scheme finds one whose value is #t, Scheme immediately evaluates and returns the value of the corresponding consequent.
- If all conditions evaluate to #f, Scheme evaluates and returns the alternative.

A Note about Some Special Forms

- The conditions given to AND, OR, NOT, COND & IF are usually expressions that evaluate to the boolean values #t or #f.
- Non-boolean values may also be used as conditions.
- Scheme considers any value other than #f to be a “true value”.
- Scheme considers #f to be the only “false value”.

The Full Truth about AND

```
(and <condition1> <condition2> ... <conditionN>)
```

- In processing “(AND...)”, Scheme evaluates the conditions in order. If some condition evaluates to #f, then Scheme immediately stops and returns #f.
- If all conditions return “true values”, then Scheme returns the value of the last condition.

The Full Truth about OR

(or <condition1> <condition2> ... <conditionN>)

- In processing “(OR...)”, Scheme evaluates the conditions in order. If all conditions return #f, then Scheme returns #f.
- If some condition evaluates to a “true value”, then Scheme immediately stops and returns that value.