CMPU 101 § 55

Computer Science I

Problem-Solving and Abstraction

Prof. Jonathan Gordon

Lecture 4

Procedural abstraction, *continued*

Making a single-item list

```
Welcome to DrRacket.
> (define embed (lambda (x) ...?...))
> (embed 'fred)
'(fred)
> (embed (embed 'fred))
'((fred))
```

Making a single-item list

```
Welcome to DrRacket.
> (define embed (lambda (x) (cons x '())))
> (embed 'fred)
'(fred)
> (embed (embed 'fred))
'((fred))
```

Cyclically permuting a list

```
Welcome to DrRacket.
> (define lst '(a b c))
> (define permute (lambda (lst) ...?...))
> (permute lst)
'(c a b)
```

Cyclically permuting a list

```
Welcome to DrRacket.
> (define lst '(a b c))
> (define permuted-lst
    (list (third lst)
          (first lst)
          (second lst)))
> permuted-lst
'(c a b)
```

Cyclically permuting a list

```
Welcome to DrRacket.
> (define lst '(a b c))
> (define permute
    (lambda (lst)
      (list (third lst)
            (first lst)
            (second lst))))
> (permute lst)
'(c a b)
```

Cyclically permuting a list

```
Welcome to DrRacket.
> (define lst '(a b c))
> (define permute
    (lambda (lst)
      (list (first (rest (rest lst)))
            (first lst)
            (first (rest lst)))))
> (permute '(a b c))
'(c a b)
```

Shuffling two lists

```
Welcome to DrRacket.
> (define list1 '(a b))
> (define list2 '(c d))
> (define shuffle
    (lambda (lst1 lst2)
      ...?...))
> (shuffle list1 list2)
'(a c b d)
```

Shuffling two lists

```
Welcome to DrRacket.
> (define list1 '(a b))
> (define list2 '(c d))
> (define shuffle
    (lambda (lst1 lst2)
      (list (first list1)
            (first list2)
            (first (rest list1))
            (first (rest list2)))))
> (shuffle list1 list2)
'(a c b d)
```

Procedural abstraction

Captures a pattern in commonly occurring expressions.

Defines a procedure that allows user to use the pattern over and over.

Gives a name to the procedure.

When the user uses the name, she does not need to remember the details of the pattern.

The user needs only to remember what arguments the procedure accepts, and what value the procedure returns.

When writing functions, include

Contract

Documentation

Test cases

Data types and predicates

Types of values in Racket

Numbers: `1`, `2`, `17`, `3.14159`.

Symbols: `barack`, `michelle`, `bo`.

Lists: `(barack michelle)`, `(malia sasha)`.

Booleans: `#t`, `#f`.

Strings: `"To be or not to be"`, `"Vassar College"`.

Procedures: `#<primitive:+>`, `#<primitive:*>`.

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 "`#t`" to represent ***true***.
- Scheme uses "`#f`" to represent ***false***.

George Boole

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

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

The Boolean values: **#t** and **#f**

```
> #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

Racket 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.

```
> (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
```

```
> (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 Racket 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 DrRacket
> (= (+ 9 16) (+ 20 5))
#t
> (= (+ 9 16) (+ 30 5))
#f
> (= 'foo 'foo)
=: expects type <number> as 1st argument,
given: 'foo; other arguments were: 'foo
```

Numeric inequality

```
> (> 7 7)
#f
> (>= 7 7)
#t
> (> (+ 1998 4) 2000)
#t
> (> 2000 (+ 1998 4))
#f
```

```
> (< 7 7)
#f
> (<= 7 7)
#t
> (< (+ 1998 4) 2000)
#f
> (< 2000 (+ 1998 4))
#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.

> In these cases, we are writing *conditional expressions* – more next time!

Acknowledgments

This lecture incorporates material from:

Tom Ellman

Luke Hunsberger