

Computer Science I

Professor Tom Ellman
Lecture 16

Higher Order Procedures

- A procedure is said to be “first-order” if none of its arguments is itself a procedure.
- A procedure is said to be “higher-order” if one or more of its arguments is a procedure.

The Importance of Higher Order Procedures

- They allow us to write very general procedures (e.g. FOR-ALL?, THERE-EXISTS?) that we can use over and over.
- They allow us to precisely define patterns of computation, such as flat recursion, deep recursion and the accumulator method.

The APPLY Procedure

Welcome to DrScheme.

```
> (apply cons '( a ( ) ) )  
(a)  
> (cons 'a '( ) )  
(a)  
> (apply + '(5 10 15))  
30  
> (+ 5 10 15)  
30  
> (apply - '(30 15 10))  
5  
> (- 30 15 10)  
5
```

The APPLY Procedure

- APPLY takes a procedure P and a list LST as inputs.
- If the length of LST is N, then P should accept N arguments.
- Suppose the value of LST is: (a1 a2 ... aN).
- Then (APPLY P LST) returns the same value as: (P 'a1 'a2 ... 'aN).
- APPLY applies P to the members of LST.
- (All at once).

The APPLY Procedure

```
(apply <proc> (list <arg1> <arg2> ... <argN>))
```



Is Equivalent To:



```
<proc> <arg1> <arg2> ... <argN>
```

The APPLY Procedure

```
(apply <proc> '(<arg1> <arg2> ... <argN>))
```



Is Equivalent To:



```
(<proc> '<arg1> '<arg2> ... '<argN>)
```

The APPLY Procedure

```
(apply <proc> <lst>)
```



Is Equivalent To:



```
(<proc> (get-nth 1 <lst>) (get-nth 2 <lst>) ... (get-nth N <lst>))
```

Using APPLY to Implement SUM

```
(define sum (lambda (lst) (apply + lst)))
```

Welcome to DrScheme.

```
> (sum '(1 2 3 4 5))
15
> (sum '())
0
```

Using APPLY to Implement FACTORIAL

```
(define factorial (lambda (n) (apply * (from-to 1 n))))
```

```
(define from-to
  (lambda (low high)
    (if (> low high) '() (cons low (from-to (+ low 1) high)))))
```

Welcome to DrScheme.

```
> (factorial 5)
120
> (factorial 0)
1
```

The MAP Procedure

```
(define map (lambda (fun lst) ...?...))
```

Welcome to DrScheme.

```
> (map list '(a b c))
((a) (b) (c))

> (map car '(a b c) (d e) (f))
(a d f)

> (map cdr '(a b c) (d e) (f))
((b c) (e) ())
```

The MAP Procedure

```
(define map (lambda (fun lst) ...?...))
```

Welcome to DrScheme.

```
> (map (lambda (x) (+ x 1)) '(1 2 3))
(2 3 4)

> (map (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
```

The MAP Procedure

- MAP takes a procedure P and a list LST as inputs.
- The procedure P should accept one argument.
- Suppose the value of LST is: (a1 a2 ... aN).
- Then (MAP P LST) returns the same value as (LIST (P a1) (P a2) ... (P aN)).
- MAP applies P to the members of LST.
- (One at a time).

The MAP Procedure

```
(map <proc> (list <arg1> <arg2> ... <argN>))
```



Is Equivalent To:



```
(list (<proc> <arg1>) (<proc> <arg2>) ... (<proc> <argN>))
```

The MAP Procedure

```
(map <proc> '<arg1> <arg2> ... <argN>))
```



Is Equivalent To:



```
(list (<proc> '<arg1>') (<proc> '<arg2>') ... (<proc> '<argN>'))
```

The MAP Procedure

```
(map <proc> <lst>)
```



Is Equivalent To:



```
(list (<proc> (get-nth 1 <lst>))  
      (<proc> (get-nth 2 <lst>))  
      ...  
      (<proc> (get-nth N <lst>)))
```

Definition of MAP Procedure

```
(define map  
  (lambda (p lst)  
    (if (null? lst)  
        '()  
        (cons (p (car lst)) (map p (cdr lst))))))
```

The FOR-ALL? Procedure

```
(define for-all? (lambda (lst pred?) ...?...))
```

Welcome to DrScheme.

```
> (for-all? '(5 2 9 4) positive?)  
#t  
> (for-all? '(5 2 -9 4) positive?)  
#f  
> (for-all? '(5 2 9 4) number?)  
#t  
> (for-all? '(5 b 9 4) number?)  
#f  
> (for-all? '() number?)  
#t
```

The FOR-ALL? Procedure

- FOR-ALL? takes a list LST and a predicate PRED? as inputs.
- The predicate PRED? should accept one argument.
- FOR-ALL? returns #t if (PRED? M) would return #t for every member M of LST.
- Otherwise, FOR-ALL? returns #f.
- Universal Quantification: $(\forall x)P(x)$.

Definition of FOR-ALL?

```
(define for-all?  
  (lambda (lst pred?)  
    (or (null? lst)  
        (and (pred? (car lst))  
              (for-all? (cdr lst) pred?))))))
```

Definition of FOR-ALL?

```
(define for-all?  
  (lambda (lst pred?)  
    (not (member? #f (map pred? lst)))))
```

The THERE-EXISTS? Procedure

```
(define there-exists? (lambda (lst pred?) ...?...))
```

Welcome to DrScheme.

```
> (there-exists? '(-5 -2 9 -4) positive?)  
#t  
> (there-exists? '(-5 -2 -9 -4) positive?)  
#f  
> (there-exists? '(a z 0 1) number?)  
#t  
> (there-exists? '(a b c d) number?)  
#f  
> (there-exists? '() number?)  
#f
```

The THERE-EXISTS? Procedure

- THERE-EXISTS? takes a list LST and a predicate PRED? as inputs.
- The predicate PRED? should accept one argument.
- THERE-EXISTS? returns #t if there is at least one member M of LST such that (PRED? M) would return #t.
- Otherwise, THERE-EXISTS? returns #f.
- Existential Quantification: $(\exists x)P(x)$.

Definition of THERE-EXISTS?

```
(define there-exists?  
  (lambda (lst pred?)  
    (and (not (null? lst))  
         (or (pred? (car lst))  
             (there-exists? (cdr lst) pred?))))))
```

Definition of THERE-EXISTS?

```
(define there-exists?  
  (lambda (lst pred?)  
    (member? #t (map pred? lst))))
```