Principles of Programming Languages
Lecture 11

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig
Concepts of PL by Sebesta
Prog. in Prolog by Clocksin and Mellish
PL Concepts and Constructs by Sethi
ML for the Working Prog. By Paulson
PL Pragmatics by Scott

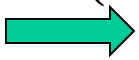
Scheme Practice 4

Write a Scheme function rm-dupl that is called as (rm-dupl L). L is a list of atomic values. rm-dupl removes duplicate values in L, returning a list in which each value only appears once (the order of values is unimportant). You can use member? For example:

```
]=> (rm-dupl '(1 2 3 4))
(1 2 3 4)
]=> (rm-dupl '(1 1 1 1))
(1)
]=> (rm-dupl '(a b c b c a))
(b c a)
```

```
(define (remove-duplicates lst)
    (cond
        ((null? lst)
                '())
        ((member? (car lst) (cdr lst))
                (remove-duplicates (cdr lst)))
        (else
                (cons (car lst) (remove-duplicates (cdr lst))))))
```

Scheme Practice 4 - trace



```
(define (remove-duplicates lst)
      (cond
            ((null? lst)
                        '())
            ((member? (car lst) (cdr lst))
                        (remove-duplicates (cdr lst)))
            (else
                        (cons (car lst) (remove-duplicates (cdr lst))))))

]=> (remove-duplicates '(b b c b))
```

Scheme Practice 4 - trace

```
(define (remove-duplicates lst)
    (cond
        ((null? lst)
                '())
        ((member? (car lst) (cdr lst))
                (remove-duplicates (cdr lst)))
        (else
                (cons (car lst) (remove-duplicates (cdr lst))))))

]=> (remove-duplicates '(b b c b))
```

Scheme Practice 4 - trace

```
(define (remove-duplicates lst)
        (cond
                ((null? lst)
                        '())
                ((member? (car lst) (cdr lst))
                    (remove-duplicates (cdr lst)))
                (else
                        (cons (car lst) (remove-duplicates (cdr lst))))))

]=> (remove-duplicates '(b b c b))
        (remove-duplicates '(b c b))
```

Scheme Practice 4 - trace

```
(define (remove-duplicates lst)
        (cond
                ((null? lst)
                        '())
                ((member? (car lst) (cdr lst))
                    (remove-duplicates (cdr lst)))
                (else
                        (cons (car lst) (remove-duplicates (cdr lst))))))

]=> (remove-duplicates '(b b c b))
                (remove-duplicates '(b c b))
                        (remove-duplicates '(c b))
```

Scheme Practice 4 - trace

```
(define (remove-duplicates lst)
        (cond
                ((null? lst)
                        '())
                ((member? (car lst) (cdr lst))
                        (remove-duplicates (cdr lst)))
                (else
                        (cons (car lst) (remove-duplicates (cdr lst))))))
```

```
]=> (remove-duplicates '(b b c b))
        (remove-duplicates '(b c b))
                (remove-duplicates '(c b))
                        (cons c  (remove-duplicates(b))
```

Scheme Practice 4 - trace

```
(define (remove-duplicates lst)
        (cond
                ((null? lst)
                        '())
                ((member? (car lst) (cdr lst))
                        (remove-duplicates (cdr lst)))
                (else
                        (cons (car lst) (remove-duplicates (cdr lst))))))

]=> (remove-duplicates '(b b c b))
                (remove-duplicates '(b c b))
                        (remove-duplicates '(c b))
                                (cons c  (remove-duplicates(b))
                                        (cons b (remove-duplicates( '() ))
```

Scheme Practice 4 - trace

```
(define (remove-duplicates lst)
        (cond
                ((null? lst)
                  ⇒  '())
                ((member? (car lst) (cdr lst))
                                (remove-duplicates (cdr lst)))
                (else
                                (cons (car lst) (remove-duplicates (cdr lst))))))

]=> (remove-duplicates '(b b c b))
                (remove-duplicates '(b c b))
                                (remove-duplicates '(c b))
                                                (cons c  (remove-duplicates(b))
                                                                (cons b (remove-duplicates( '() ))
```

Scheme Practice 4 - trace

```
(define (remove-duplicates lst)
        (cond
                ((null? lst)
                        '())
                ((member? (car lst) (cdr lst))
                        (remove-duplicates (cdr lst)))
                (else
                        (cons (car lst) (remove-duplicates (cdr lst))))))
```

```
]=> (remove-duplicates '(b b c b))
        (remove-duplicates '(b c b))
                (remove-duplicates '(c b))
                        (cons c  (remove-duplicates(b))
                                (cons b ('() ) )
```

Scheme Practice 4 - trace

```
(define (remove-duplicates lst)
        (cond
                ((null? lst)
                        '())
                ((member? (car lst) (cdr lst))
                        (remove-duplicates (cdr lst)))
                (else
                        (cons (car lst) (remove-duplicates (cdr lst))))))
```

```
]=> (remove-duplicates '(b b c b))
        (remove-duplicates '(b c b))
                (remove-duplicates '(c b))
                        (cons c  (remove-duplicates(b))
                                (cons b ('() ) )
                         (cons c  '(b))
```

Scheme Practice 4 - trace

```
(define (remove-duplicates lst)
      (cond
            ((null? lst)
                  '())
            ((member? (car lst) (cdr lst))
                  (remove-duplicates (cdr lst)))
            (else
                  (cons (car lst) (remove-duplicates (cdr lst))))))
```

```
]=> (remove-duplicates '(b b c b))
            (remove-duplicates '(b c b))
                  (remove-duplicates '(c b))
                        (cons c  (remove-duplicates(b))
                              (cons b ('() ) )
                         (cons c  '(b))
                  (c b)
            (c b)
      (c b)
```

Software Verification

Software Verification

- **Software crisis**
  - Quality of software?

- **Enforce roles/responsibilities**

  Architects ←→ Programmers ←→ QA

  - QA test the program on many different inputs
  - However, subtle bugs may remain undiscovered, only to appear at random, inconvenient or dangerous moments.

- **Implement quality processes (e.g. CMM,ISO…)**
  - However, bugs may remain undiscovered

- **Use Algebraic proofs**
  - Especially useful for safety-critical software (e.g. Air control,….)

Proving Properties of Programs

- **Example:**

```
(define (append X Y)
        (if (null? X ) Y
            (cons (car X) (append (cdr X) Y ))))

(define (length X)
        (if (null? X) 0
            (+ 1 (length (cdr X)))))
```

– **Prove the following:**

Theorem: (length (append X Y)) = (length X) + (length Y)
for all lists X and Y

Software Verification: prove outline

- **Use mathematical induction on the length of X**
  - A two-part method of proving a theorem involving an integral parameter.
    - First the theorem is verified for the smallest admissible value of the integer.
    - Then it is proven that if the theorem is true for any value of the integer, it is true for the next greater value. The final proof contains the two parts.

- **First, prove that the theorem is true for lists of length 0 [Basis]**

- **Then, prove that <u>if</u> the theorem is true for lists of length N, <u>then</u> it is also true for lists of length N+1 [Inductive Step]**

- **This implies that the theorem is true for lists of any length (i.e. for any list)**

Software Verification: structural induction

- **Actually, we will use a variation of induction that emphasizes the structure of lists, not their length.**

- **First, prove that the theorem is true for X = ' () i.e. when X has length 0 [Basis]**

- **Then, prove that if the theorem is true for X = L, then it is also true for X = (cons E L) [Inductive Step]**
  - Note: if L has length N, then (cons E L) has length N + 1

Software Verification: preliminaries

- **Before using induction (or any other technique) to prove a complex property of a program, write down the basis properties that can be trivially verified by inspecting the program code**

- **The inductive proof should only use these properties of the code.**

- **The code itself plays no other role in a proof of correctness and can be henceforth ignored.**

- ***Note:*** **If the basic properties are wrong, then the entire proof is wrong. So be sure to get them right!!**

Basic properties of append

- **Code:**

```
(define (append X Y)
        (if (null? X ) Y
            (cons (car X)  (append (cdr X) Y ))))
```

- **Using X = ' ()**

(append ' () Y) = Y **[1]**

- **Using X = (cons E L)**

(append (cons E L) Y) = (cons E (append L Y)) **[2]**

- ***Note:***
  - if X = (cons E L), then (car X) = E, (cdr X) = L

Basic Properties of Length

- **Code:**

```
(define (length X)
        (if (null? X) 0)
            (+ 1 (length (cdr X)))))
```

- **Using X = ' ()**

(length ' ()) = 0 **[3]**

- **Using X = (cons E L)**

(length (cons E L)) = 1+ (length L) **[4]**

- ***Note:***
  - if X = (cons E L), then (cdr X) = L

Summary of basic properties

- **(append ' () Y) = Y** **[1]**
- **(append (cons E L) Y) = (cons E (append L Y))** **[2]**
- **(length ' ()) = 0** **[3]**
- **(length (cons E L)) = 1+ (length L)** **[4]**
- **Using these basic properties, we shall prove (by induction) a more complex property:**

  Theorem: (length (append X Y)) = (length X) + (length Y)

Software Verification: basis proof

- **Using induction on the structure of X**

- **Basis:** **when X = ' ()**

(length (append X Y))
= (length (append '() Y))
= (length Y) [by 1]
= 0 + (length Y)
= (length ' ()) + (length Y) [by 3]
= (length X) + (length Y)

Therefore, the theorem is true when X = ' ()

(append ' () Y) = Y **[1]**

(append (cons E L) Y) = (cons E (append L Y)) **[2]**

(length ' ()) = 0 **[3]**

(length (cons E L)) = 1+ (length L) **[4]**

Software Verification: inductive step

- **Suppose that the theorem holds for X = L , i.e. suppose that**

  (length (append L Y)) = (length L) + (length Y)

  {*Inductive hypothesis*}

- **Now, prove that the theorem holds for X = (cons E L)**

Proof of inductive step

- **If X = (cons E L) then**

	(length (append X Y))	
=	(length (append (cons E L) Y))	
=	(length (cons E (append L Y)))	[by 2]
=	1 + (length (append L Y))	[by 4]
=	1 + (length L) + (length Y)	[by inductive hypothesis]
=	(length (cons E L)) + (length Y)	[by 4]
=	(length X) + (length Y)	

(append ' () Y) = Y [1]

(append (cons E L) Y) = (cons E (append L Y)) [2]

(length ' ()) = 0 [3]

(length (cons E L)) = 1+ (length L) [4]

Summary of proof of theorem

- **For <u>any</u> List, Y,**
- **Basis:** **the theorem holds for X = ' ()**
- **Inductive Step:** **if the theorem holds for X = L, then it holds for X = (cons E L)**
- **By the principle of structural induction, the theorem holds for <u>any</u> lists X and Y**
- **Theorem:**

  (length (append X Y)) = (length X) + (length Y)

Proving properties of programs

- **Example 2:**

```
(define (member A X)
  (cond   ((null? X) #f)
          ((equal? A (car X)) #t)
          (else (member A (cdr X)))))
```

– **Prove the following:**

Theorem: If (member A X) then
(member A (append X Y))
for any A, and any lists X and Y.

Outline of proof

- As before, we will use structural induction on X
- First, prove that the theorem is true for X=' () [basis]
- Then, prove that if the theorem is true for X = L, then it is true for X = (cons E L) [inductive step]
- However, this time proof will be more complex, because the program can terminate its recursion in two ways.

Basic properties of member

- **Code:**

```
(define (member A X)
    (cond   ((null? X) #f)
            ((equal? A (car X)) #t)
            (else (member A (cdr X)))))
```

- **Using X = ' ()**

(member A ' ()) = #f **[5]**

- **Using X = (cons A L)**

(member A (cons A L)) = #t **[6]**

- **Note:**
  - (car X) = A

Basic properties of member – cont'd

- **Code:**

```
(define (member A X)
    (cond   ((null? X) #f)
            ((equal? A (car X)) #t)
            (else (member A (cdr X)))))
```

- **Using X = (cons E L) where E != A**

(member A (cons E L)) = (member A L) **[7]**

- **Note:**
  - (car X) = E, (cdr X) = L

Summary of basic properties

- **member Code:**
  - (member A ' ()) = #f [5]
  - (member A (cons A L)) = #t [6]
  - If E!=A then
    (member A (cons E L)) = (member A L) [7]

- **append Code:**
  - (append ' () Y) = Y [1]
  - (append (cons E L) Y) = (cons E (append L Y)) [2]

Proof of theorem

- **Theorem:** If (member A X) then
  (member A (append X Y))

- **Proof: by induction on X**

- **Basis:** **if X = ' () then**

(member A X)
= (member A ' ())
= #f [by 5]

Therefore, the premise of the theorem is false.

Therefore, the theorem is trivially true (for X = ' ())

Inductive step

- **Suppose the theorem is true for X=L, i.e.**

  if (member A L) then

  (member A (append L Y))

  {Inductive hypothesis}

- **Now, prove the theorem is true for X = (cons E L)**

Proof of inductive step

- **Let X = (cons E L) , there are two cases**

- **Case 1: E = A**

Therefore, **X = (cons A L)**

Therefore, (member A (append X Y))

= (member A (append (cons A L) Y))

= (member A (cons A (append L Y))) [by 2]

= #t [by 6]

Therefore, the conclusion of the theorem is true

Therefore, the theorem itself is trivially true for X = (cons A L)

(append (cons E L) Y) = (cons E (append L Y)) **[2]**

(member A (cons A L)) = #t **[6]**

Proof of inductive step – cont'd

- **Let X = (cons E L), there are two cases**

- **Case 2: X = (cons E L) where E != A**

if(member A X) = #t then

	(member A (cons E L))	
=	??	
=	(member A L)	
=	(member A (append L Y))	[by inductive hypothesis]
=	(member A (cons E (append L Y)))	[by 7]
=	(member A (append (cons E L) Y))	[by 2]
=	(member A (append X Y))	

Therefore, the theorem is true when X = (cons E L) and E != A

(append (cons E L) Y) = (cons E (append L Y)) [2]
If E!=A then (member A (cons E L)) = (member A L) [7]

Summary of proof of theorem

- **For any A, and any list Y**

- **Basis**

  The theorem holds for X = ' ()

- **Inductive Step:**

  If the theorem holds for X = L, then it holds for X = (cons E L)

  Therefore, by the principle of structural induction, the theorem holds for any A and any lists X and Y.

- **Theorem:** If (member A X) then

  (member A (append X Y))

Software Verification

- **Interested to learn more…**
  - CSC465: Formal Methods in Software Design

Possible Directions for PLs

- **Logic**
  - Program
    - declarative English-like statements
    - Control embedded in interpreter
  - Testing:
    - Evaluate query answer against known answers

- **Functional**
  - Program
    - Pure functions
  - Testing
    - Proof it!

Mixing Imperative & Logic/functional

- **Logic**
  - Use plugin (e.g. swi-prolog, jiprolog, InterProlog, JPL…)

- **Functional**
  - Problems:
    - Global variables
    - Pass-by-reference
    - Polymorphism

What is polymorphism?

- **Biology:**
  - occurs when two or more clearly different phenotypes exist in the same population of a species

- **Computer science:**
  - polymorphism is a programming language feature that allows values of different data types to be handled using a uniform interface

Polymorphism

```
class Circle {
    int _nCenter;
    Circle( ) {
        _nCenter = 10
    }
    draw(int nX, int nY){

   }
}
```

```
98:  …..
99: mov Ax,10
100: draw …
…..
```

```
class ScreenManager {
    Circle_circle;
    void refresh (  ){
            _circle.draw( );
   }
}
```

```
10:  …..
20:  …..
30:  …..
40: jump 100
```

Polymorphism

```
class Shape{

    draw(int nX, int nY){

   }
}
```

```
class Circle inherits Shape{

    draw(int nX, int nY){

   }
}
```

```
class Rectangle inherits Shape{

    draw(int nX, int nY){

   }
}
```

```
class ScreenManager {
    Shape _shape;
    void refresh (  ){
            _shape.draw( );
   }
}
```

Polymorphism

```
class Circle {
    int _nCenter;
    Circle( ) {
        _nCenter = 10
    }
    draw(int nX, int nY){

   }
}
```

```
98:  …..
99: mov Ax,10
100: draw …
…..
```

```
class ScreenManager {
    Shape  _shape;
    void refresh (  ){
            _shape.draw( );
   }
}
```

```
10:  …..
20:  …..
30:  …..
40: jump ???
```