Tutorial 3

Week of October 3, 2005

1 Good Programming Style

```
Please read the following document.  Pay particular attention to
the issue of programming-by-contract and what is expected in
your code:
   http://www.cs.toronto.edu/~sheila/324/f05/assns/marking.html

Please ensure that you:
- proper indentation
- meaningful procedure and argument names
- document your code w/ pre- and post-conditions
```

2 Conditional Control structures

```
(if <condition>
    <then-expression>
    <else-expression>)

 Example:
    1 ]=> (if (> 3 2)
              'foo
              'bar)
    ;Value: foo

    1 ]=> (if (= 3 2)
              'foo
              'bar)
    ;Value: bar

    1 ]=> (if (> 3 2)
              'foo)
    ;Value: foo

    1 ]=> (if (= 3 2)
              'foo)
    ;Unspecified return value   <---- generally, a bad thing to have

    1 ]=> (if (> 3 2)
              'foo
              bar)       <---- bar is not evaluated
    ;Value: foo                called ``Lazy evaluation''

    1 ]=> (if (= 3 2)
              'foo
               bar)
   ;Unbound variable: bar  <---- bar is evaluated => ERROR
```

```
(cond  ( <condition1> <expression1> )
       ( <condition2> <expression2> )
                ...
       ( <conditionN-1> <expressionN-1> )
       ( else <expressionN> ))


 1 ]=> (cond  ( (< 2 2) 'foo )
              ( (> 2 2) 'bar ))
 ;Unspecified return value   <--- generally, not a good thing

 1 ]=> (cond  ( (< 2 2) 'foo )
              ( (> 2 2) 'bar )
              ( (= 2 2) 'foobar)) <--- not a good thing
 ;Value: foobar                        unnecessary evaluation

 1 ]=> (cond  ( (< 2 2) 'foo )
              ( (> 2 2) 'bar )
              (else 'foobar))  <--- much better now
 ;Value: foobar

 1 ]=> (cond  ( (> 3 2) 'foo )
              ( (< 3 2)  bar  )  <---- bar is NOT evaluated
              ( else    'foobar))      Lazy evaluation again
 ;Value: foo

 1 ]=> (cond  ( (< 3 2) foo )   <--- foo is NOT evaluated
              ( (> 3 2) 'bar )
              (else 'foobar))
 ;Value: bar

 1 ]=> (cond  ( (< 3 2) foo ) <--- foo is NOT evaluated
              ( (= 3 2) bar ) <--- bar is NOT evaluated
              (else   foobar)) <--- foobar is evaluated => ERROR
;Unbound variable: foobar...
```

3 Lists

```
(cons <arg1> <arg2>) ,
     where <arg1> and <arg2> are arbitrary, but both are necessary
(list <arg1> <arg2> ... <argN>) ,
     where <arg1> <arg2> ... <argN> are arbitrary, neither is necessary
(append <arg1> <arg2> ... <argN>)
     where <arg1> <arg2> ... <argN-1> are lists and <argN> is
     arbitrary, neither is necessary
```

Draw pictures of:

1. () can come from (list)

   Picture: ()

2. (1) can come from (list 1)

3. (1 . 2)
   Note the spaces around the "."
   can come from (cons 1 2)

4. (1 . ())
   It is the same as (1) !!!
   can come from (cons 1 ())

5. (()) can come from (list ()) or from (cons () ())

6. ((1 2) 3 () ((4) 5) 6) could come from:

```
( list  '(1 2)   3   ()   (list (list 4) 5)   6)
           ^                     ^
           |                     |
  could be (list 1 2)     could be '( (4) 5)
```

7. ((1 2) 3 () ((4) . 5) 6)
   could come from (list (list 1 2) 3 () (cons '(4) 5) 6)

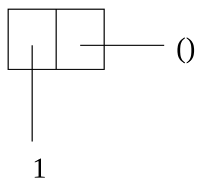

Figure 1: (1)

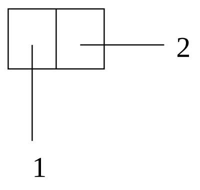

Figure 2: (1 . 2)

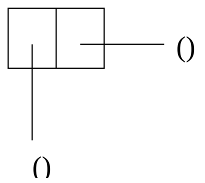

Figure 3: (())

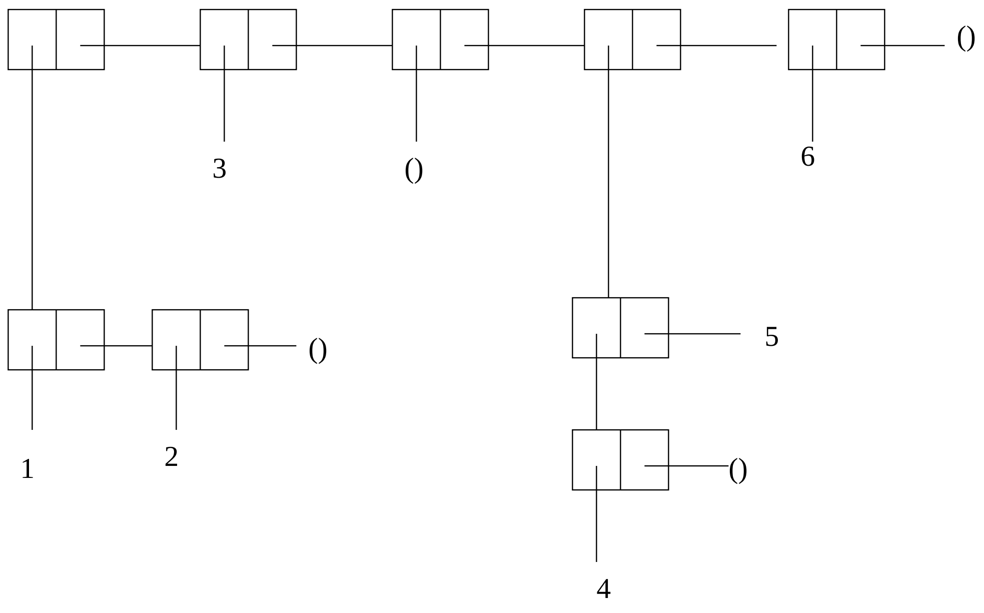

Figure 4: ((1 2) 3 () ((4) . 5) 6)

4 Recursive procedures

Marks will be deducted in A2 if you don't include Pre, Post, Args, etc., or if the latter are unclear or underspecified.

1. Write a procedure **sum-list-large** that takes a list of numbers and computes the sum of all numbers greater than 2 in the list. Return 0 if there are no such numbers in the input list.

```
;; (sum-list-large lst) return the sum of all numbers that are
;;  greater than 2 in lst
;; Args: lst - a list of numbers
;; Pre: lst is flat list of numbers; lst can be empty
;; Post: none
;;Return: if lst contains numbers greater than 2, their sum
;;        ow, 0
(define (sum-list-large lst)
  (cond ((null? lst) 0)
        ((> (car lst) 2) (+ (car lst) (sum-list-large (cdr lst))))
        (else (sum-list-large (cdr lst)))))
```

2. Write a procedure that takes a non-negative integer n and an object as input and returns a list of n objects.

```
  E.g., (make-list 7 '())  returns  (() () () () () () ())
        (make-list 3 'csc324) returns ( csc324 csc324 csc324 )
```

```
;; (make-list n object) returns a list of n objects
;; Args: n - the number of times object appears in the result list
;;       object - each element of the resulting list
;; Pre: n - non-negative integer
;; Post: none
;; Return: a list of n objects
(define (make-list n object)
  (if (= n 0)
      '()
      (cons object (make-list (- n 1) object)))))
```

5 More Examples

1. Write a procedure **member?** that takes an object and a flat list as inputs and tests whether the object is an element of the input list.

```
;; (member? elt lst) tests whether elt appears in lst
;; Args: elt - element to be tested for membership
;;       lst - the list to be tested for containing elt
;; Pre: lst - a flat list
;;      equal? is appropriate to test for equality of elt with
;;      elements of lst
;; Post: none
;; Return: true, if elt appears in lst
;;         false, otherwise
(define (member? elt lst)
   (cond  ( (null? lst) () )
          ( (equal? elt (car lst)) #t)
          ( else (member? elt (cdr lst)))))
```

2. Write a procedure **intersect** that computes the intersection of two lists. In other words, given two lists as arguments, it returns a list of elements contained in both lists.

```
Example:
]=> (intersect '(1 2 3 4) '(10 2 4 100) )
;Value: (2 4)

]=> (intersect '(john david) '(david 2 sky 4) )
;Value: (david)
```

```
; (intersect lst1 lst2) returns a list of elements contained
;; in both lst1 and lst2
;; Parameters: lst1 and lst2 are lists
;; Preconditions: none
;; Postconditions: none
;; Return values: a list of elements contained both in lst1 and lst2
(define (intersect lst1 lst2)
    (cond ((null? lst1) ())
          ((member? (car lst1) lst2)
             (cons (car lst1) (intersect (cdr lst1) lst2)))
          (else (intersect (cdr lst1) lst2))))
```

3. Write a procedure **union** that computes the union of two lists. In other words, given two lists as arguments, it returns a list of elements contained in either of the two lists, but does not create duplicates.

```
Example:
]=> (union '(1 2 3 4) '(10 2 4 100) )
;Value: (1 2 3 4 10 100)

]=> (union '(john david) '(david 2 sky 4) )
;Value: (john david 2 sky 4)
; (union lst1 lst2) returns a list of elements contained
;; in either lst1 or lst2, but does not create duplicates
;; Parameters: lst1 and lst2 are lists
;; Preconditions: none
;; Postconditions: none
;; Return values: a list of elements contained both in lst1 and lst2
(define (union lst1 lst2)
    (cond ((null? lst1) lst2)
          ((member? (car lst1) lst2) (union (cdr lst1) lst2)))
          (else (cons (car lst1) (union (cdr lst1) lst2))))
```

6 Proofs

Recall the procedure **factorial**.

```
;; (factorial n) returns n!
;; Args: n - a number, factorial of which is returned
;; Pre: n is an integer, n >=0
;; Post: none
;; Return: n!
(define (factorial n)
  (if (= n 0)
      1
     (* n (factorial (- n 1)))))
```

We want to prove that $(factorial\ n) = n!\ \forall n \in \mathbf{N},\ n \geq 0$. Define $P(n)$ to stand for $(factorial\ n) = n!$. We prove by induction on n:

1. Base case:

   ```
      (factorial 0)                       [definition of (factorial n)]
   == (if (= 0 0)
           1
           (* 0 (factorial (- 0 1))))    [evaluation of (= 0 0)]
   == (if #t
           1
           (* 0 (factorial (- 0 1))))    [evaluation of if structure]
   == 1                                   [definition of factorial]
   == 0!
   ```

   We thus conclude that $P(0)$ is true.

2. Inductive step:

   Assume $P(i)$ for an arbitrary $i \in \mathbf{N}$, $i \geq 0$. In other words, we assume that $(factorial\ i) = i!$ for an arbitrary $i \in \mathbf{N}$, $i \geq 0$. This is our inductive hypothesis (IH).

   ```
      (factorial (i+1))                       [definition of (factorial n)]
   == (if (= (i+1) 0)
            1
   ```

```
         (* (i+1) (factorial (- (i+1) 1)))) [arithmetic]
== (if (= i -1)
          1
         (* (i+1) (factorial i)))           [evaluation of (= i -1)
                                             according to IH i>=0]
== (if #f
          1
         (* (i+1) (factorial i)))           [evaluation of if structure]
== (* (i+1) (factorial i))                  [IH]
== (* (i+1) i!)                             [definition of factorial]
== (i+1)!
```

We thus conclude that $P(i) \Longrightarrow p(i+1)$ for any $i \geq 0$, $i \in \mathbf{N}$.

Thus, by the Principle of (weak) Induction, we conclude that $P(n)$ is true for all $n \in \mathbf{N}$, $n \geq 0$. In other words, $(factorial\ n) = n!\ \forall n \in \mathbf{N}$, $n \geq 0$.