Functional programming with Common Lisp

Dr. Constantinos Constantinides, P.Eng.

Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
cc@cse.concordia.ca

August 23, 2018

Expressions and functions

- Expressions are written as lists, using prefix notation. Prefix notation is a form of notation for logic, arithmetic, and algebra. It places operators to the left of their operands.
- For example, the (infix) expression $14 \; - \; (2 \; \times \; 3)$ is written as $(-\ 14\ (\times\ 2\ 3))$.
- The first element in an expression list is the name of a function and the remainder of the list are the arguments:

$$(function - name \quad arguments)$$

Arity of functions

- The term *arity* is used to describe the number of *arguments* or *operands* that a function takes.
- A *unary* function (arity 1) takes one argument. A *binary* function (arity 2) takes two arguments.
- A ternary *function* (arity 3) takes three arguments, and an `n`-ary function takes `n` arguments.
- *Variable arity* functions can take any number of arguments.

```
(+ 1 2 3 4) ; Equivalent to infix (1 + 2 + 3 + 4).
            ; Returns 10.

(* 2 3 4)   ; Equivalent to infix (2 * 3 * 4). Returns 24.

(< 1 3 2)   ; Equivalent to (1 < 3 < 2).
            ; Returns NIL (false).
```

Prohibiting expression evaluation

- The subexpressions of a procedure application are evaluated, whereas the subexpressions of a quoted expression are not.

```
 (/ (* 2 6) 3)    ; Returns 4.
'(/ (* 2 6) 3)    ; Returns (/ (* 2 6) 3).
```

Boolean operations

- Lisp supports Boolean logic with operators `and`, `or`, and `not`. The two former have variable arity, and the last one is a unary operator.
- The `or` Boolean operator evaluates its subexpressions from left to right and stops immediately (without evaluating the remaining expression) if any subexpression evaluates to *true*.
- In the example below the `or` function will return *true* which is the value of `(> x 3)`.
- Note that the values *true*/*false* are denoted in Lisp by `t`/`nil` respectively.

```
> (let ((x 5))
(or (< x 2) (> x 3)))
T
```

Boolean operations /cont.

- The `and` Boolean operator evaluates its subexpressions from left to right and stops immediately (without evaluating the remaining expression) if any subexpression evaluates to *false*.
- In the example below the `and` function will return `nil` which is the value of `(< x 3)`.

```
> (let ((x 5))
(and (< x 7) (< x 3)))
NIL
```

- Consider another example:

```
>(or (and (= 1 1) (< 5 6)) (not (> 3 1)))
T
```

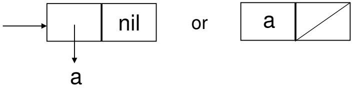
Constructing lists

- We have three (built-in) functions to create a list which are summarized below:
  1. `cons`: creates a list by adding an element as the head of an existing list.
  2. `list`: creates a list comprised of its arguments.
  3. `append`: creates a list by concatenating existing lists.

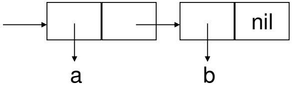
Function cons

- A list in Lisp is singly-linked where each node is a pair of two pointers, the first one pointing to a data element and the second one pointing to the tail of the list with the last node's second pointer pointing to the empty list.

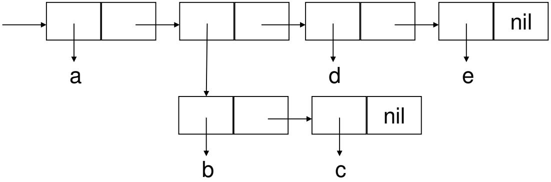
```
> (cons 'a '())
(A)
```




```
> (cons 'a (cons 'b `()))
(A B)
```




```
> (cons 'a (cons (cons 'b (cons 'c '()))
                 (cons 'd  (cons 'e '()))))
(A (B C) D E)
```



Function `list`

- Lists can be created directly with the `list` function, which takes any number of arguments, and it returns a list composed of these arguments.

```
(list 1 2 'a 3)               ; Returns (1 2 A 3).
(list 1 '(2 3) 4)             ; Returns (1 (2 3) 4).
(list '(+ 2 1) (+ 2 1))       ; Returns ((+ 2 1) 3).
(list 1 2 3 (list 'a 'b 4) 5) ; Returns (1 2 3 (a b 4) 5).
```

Function append

- The append function takes any number of list arguments and it returns a list which is the concatenation (join) of its arguments:

```
(append '(1 2) '(3 4))               ; Returns (1 2 3 4).
(append '(1 2 3) '() '(a) '(5 6)) ; Returns (1 2 3 a 5 6).
(append '(1 2 3 '(a b c)) '() '(d) '(4 5))
; Returns (1 2 3 (QUOTE (a b c)) d 4 5).
```

- Note that append expects as its arguments only lists. The following call to append will cause an error since the first argument, 1, is not a list.

```
> (append 1 '(4 5 6))
Error: 1 is not of type LIST.
```

Function `append` for list concatenation

- To create a list `(1 4 5 6)` we must first transform 1 into a list:

```
> (append (list 1) '(4 5 6))
(1 4 5 6)
```

Accessing a list

- Only two operations are available. We can only access either the head of a list, or the tail of a list.
- Operation `car` (also: `first`) takes a list as an argument and returns the head of the list. For example,

```
(car '(a s d f))    ; Returns a.
(car '((a s) d f)) ; Returns (a s).
```

- Operation `cdr` (also: `rest`) takes a list as an argument and returns the tail of the list. For example,

```
(cdr '(a s d f))      ; Returns (s d f).
(cdr '((a s) d f))    ; Returns (d f).
(cdr '((a s) (d f))) ; Returns ((d f)).
```

Accessing a list /cont.

- In the following example, we are interested in accessing the second element in a list.
- The second element is the head of the tail of the list:

```
(car (cdr '(1 (3 5) (7 11)))) ; Returns (3 5).
```

Predicate functions

- A function whose return value is intended to be interpreted as truth or falsity is called a predicate. The built-in function `listp` returns *true* if its argument is a list. For example,

```
(listp '(a b c)) ; Returns T (true).
(listp 7)        ; Returns NIL (false).
```

- Other common predicate functions include the following:

Predicate	Description
(`numberp` *argument*)	Returns *true* if *argument* is a number.
(`zerop` *argument*)	Returns *true* if *argument* is zero.
(`evenp` *argument*)	Returns *true* if *argument* is an even number.
(`oddp` *argument*)	Returns *true* if *argument* is an odd number.

Control flow: Single selection

- The simplest single conditional is `if`.

(`if` *testExpression*
thenExpression)

(`if` *testExpression*
thenExpression
elseExpression)

- The *testExpression* is a predicate while the *thenExpression* and the (optional) *elseExpression* are expressions.

Control flow: Multiple selection

- Multiple selection can be formed with a `cond` expression which contains a list of clauses where each clause contains two expressions, called condition and answer. Optionally, we can have an `else`.

```
( cond  (question  answer)
        ...
        (else  answer)   ; Optional.
      )
```

- Conditions are evaluated sequentially.
- For the first condition that evaluates to *true*, Lisp evaluates the corresponding answer, and the value of the answer is the value of the entire `cond` expression.
- If the last condition is `else` and all other conditions fail, the answer for the `cond` expression is the value of the last answer expression.
- We can also use `t` (true) in place of `else`.

Variables and binding

- *Binding* is a mechanism for implementing lexical scope for variables.
- The `let` syntactic form takes two arguments: a list of bindings and an expression (the body of the binding) in which to use these bindings.

$$
\begin{array}{l}
(\ \texttt{let} \\
\qquad (\ (binding_1) \\
\qquad\quad (binding_2) \\
\qquad\quad \ldots \qquad) \\
\qquad (expression)\ \ )
\end{array}
$$

where $(binding_n)$ is of the form $(variable_n\ value)$.

Variables and binding /cont.

- The let values are computed and bindings are done in parallel, which requires all of the definitions to be independent.
- In the example below, x and y are let-bound variables; they are only visible within the body of the let.

```
(let ((x 2) (y 3))
     (+ x y))

; Returns 5.
```

Context and nested binding

- An operator like `let` creates a new lexical context.
- Within this context there are new variables, and variables from outer contexts may become invisible.
- A binding can have different values at the same time:

```
(let ((a 1))
   (let ((a 2))
      (let ((a 3))
         ...)))
```

- Here, variable `a` has three distinct bindings by the time the body (marked by ...) executes in the innermost `let`.

Context and nested binding /cont.

- The inner binding for a variable shadows the outer binding and the region where a variable binding is visible is called its scope.
- Consider the following example:

```
(let ((x 1))                ; x is 1.
     (let ((x (+ x 1))) ; x is 2.
     (+ x x)))              ; Returns 4.
```

Context and nested binding /cont.

- What if we want the value of one new variable to depend on the value of another variable established by the same expression?
- In that case we have to use a variant called `let*`.
- A `let*` is functionally equivalent to a series of nested lets. Consider the following example:

```
(let* ((x 10)
       (y (* 2 x))) ; Not legal for let.
      (* x y))

; Returns 200.
```

Defining functions

- We can define new functions using `defun`. A function definition looks like this:

```
( defun  name ( formal parameter list )
         body )
```

Example: Defining functions

- Consider function `absdiff` takes two arguments and returns their absolute difference:

```
(defun absdiff (x y)
  (if (> x y)
    (- x y)
    (- y x)))
```

- We can execute the function as follows:

```
> (absdiff 3 5)
2
```

Higher-order functions

- *Higher-order functions* are functions which do at least one of the following:
  1. Take one or more functions as their arguments.
  2. Return a function.
- The derivative function in calculus is a common example, since it maps a function to another function, e.g.

$$\frac{d}{dx}\left(x^2\right) = 2x$$

Example: Higher-order functions

- As an example, consider function `sort` which takes as an argument a list, constructed through function `list`, and the comparison operator greater-than ($>$) and returns a sorted list.

```
>(sort (list 5 0 7 3 9 1 4 13 23) #'>)
(23 13 9 7 5 4 3 1 0)
```

Common higher-order functions in Lisp

- `mapcar` takes as its arguments a function and one or more lists and applies the function to the elements of the list(s) in order.

  ```
  ; Multiplication applies to successive pairs.
  > (mapcar #'* '(2 3) '(10 10))
  (20 30)
  ```

- `funcall` takes as its arguments a function and a list of arguments (does not require arguments to be packaged as a list), and returns the result of applying the function to the elements of the list.

  ```
  > (funcall #'+ 1 3 4)   ; Equivalent to (+ 1 3 4).
  8
  ```

Common higher-order functions in Lisp /cont.

- `apply` works like `funcall`, but requires that the last argument is a list.

```
> (apply #'+ 3 4 '(1 3 4))
15
```

Anonymous functions

- An *anonymous function* is one that is defined, and possibly called, without being bound to an identifier.
- Unlike functions defined with `defun`, anonymous functions are not stored in memory.
- The general syntax of an anonymous function in Lisp (also called *lambda expression*) is

  (`lambda` (*formal parameter list*) (*body*))

  where *body* is an expression to be evaluated.

Anonymous functions /cont.

- An anonymous function can be applied in the same way that a named function can, e.g.

```
> ((lambda (x) (* x x)) 3)
9
```

Anonymous functions: Example

- Consider a function that takes a list as an argument and returns a new list whose elements are the elements of the initial list multiplied by 2.
- We can perform the multiplication with an anonymous function, and deploy `mapcar` to apply the anonymous function to the elements of the list as follows:

```
>  (mapcar (lambda (n) (* n 2)) '(2 3 5 7))
(4 6 10 14)
```

Side effects in Common Lisp

- Common Lisp is not a pure functional language as it allows side effects.

Variables and assignments

- A variable is *global* if it is visible everywhere as opposed to a *local variable* which is visible only within the code block in which it is defined.
- A global variable is accessible everywhere except in expressions that create a new local variable with the same name.
- Inside code blocks, local values are always looked for first. If a local value for the variable does not exist, then a global value is sought.
- If no global value is found then the result is an error. We use `setq` to assign a global variable and `setf` to assign both global and local variables. The general format is

  (`setf` *place value*)

  and it is used to assign a new value to a place (variable).

Examples: Variables and assignments

```
> (setf x '(a b c))
(A B C)
> (car x)
A
> (cdr x)
(B C)
> (cdr (cdr (cdr x)))
NIL
> (setf x (append x '(d e)))
(A B C D E)
```

Examples: Variables and assignments /cont.

- Variables are essentially pointers.
- Function `eql` will return *true* if its arguments point to the same object, whereas function `equal` returns *true* if its arguments have the same value.

Examples: Variables and assignments /cont.

```
> x
(A B C D E)
> (setf y '(a b c d e))
(A B C D E)
> (eql x y)
NIL
> (equal x y)
T
> (setf z x)
(A B C D E)
> (eql x z)
T
> (equal x z)
T
> (eql y z)
NIL
> (equal y z)
T
```

Defining recursive functions

- In problem solving, the deployment of *recursion* implies that the solution to a problem depends on solutions to smaller instances of the same problem.
- Recursion refers to the practice of defining an object, such as a function or a set, in terms of itself. Every recursive function consists of:
  - One or more *base cases*, and
  - One or more *recursive cases* (also called *inductive cases*).

Defining recursive functions

- Each recursive case consists of:
  1. Splitting the data into smaller pieces (for example, with `car` and `cdr`),
  2. Handling the pieces with calls to the current method (note that every possible chain of recursive calls must eventually reach a base case), and
  3. Combining the results into a single result.

Defining recursive functions /cont.

- A mathematical function uses only recursion and conditional expressions.
- A mathematical conditional expression is in the form of a list of pairs, each of which is a *guarded expression*. Each guarded expression consists of a predicate guard and an expression:

$$\mathit{functionName}(\mathit{arguments}) = \mathit{expression}_1 - \mathit{predicateGuard}_1, \ldots$$

  which implies that the function is evaluated by $\mathit{expression}_n$ if $\mathit{predicateGuard}_n$ is true.

Example: $f : \mathbb{N} \to \mathit{lists}(\mathbb{N})$

- Suppose we need to define the function $f : \mathbb{N} \to \mathit{lists}(\mathbb{N})$ that accepts an integer argument and returns a list, such that

$$f(n) = \langle n, n-1, ..., 0 \rangle$$

- In this and similar problems, we can transform the definition of $f(n)$ into a computable function using available operations on the underlying structure (list).
- We can use *cons* as follows:

$$\begin{aligned} f(n) &= \langle n, n-1, ..., 1, 0 \rangle \\ &= \mathit{cons}(n, \langle n-1, ..., 1, 0 \rangle) \\ &= \mathit{cons}(n, f(n-1)). \end{aligned}$$

Example: $f : \mathbb{N} \to \mathit{lists}(\mathbb{N})$ /cont.

- We can therefore define f recursively by

$$f(0) = \langle 0 \rangle.$$
$$f(n) = \mathit{cons}(n, f(n-1)), \;\; \mathit{for} \;\; n > 0.$$

Example: $f : \mathbb{N} \to \mathit{lists}(\mathbb{N})$ /cont.

- We can visually show how this works with a technique called "unfolding the definition" (or "tracing the algorithm"). We can unfold this definition for $f(3)$ as follows:

$$\begin{aligned} f(3) &= \mathit{cons}(3, f(2)) \\ &= \mathit{cons}(3, \mathit{cons}(2, f(1))) \\ &= \mathit{cons}(3, \mathit{cons}(2, \mathit{cons}(1, f(0)))) \\ &= \mathit{cons}(3, \mathit{cons}(2, \mathit{cons}(1, \langle 0 \rangle))) \\ &= \mathit{cons}(3, \mathit{cons}(2, \langle 1, 0 \rangle)) \\ &= \mathit{cons}(3, \langle 2, 1, 0 \rangle) \\ &= \langle 3, 2, 1, 0 \rangle. \end{aligned}$$

Example: $f : \mathbb{N} \to \mathit{lists}(\mathbb{N})$ /cont.

- We can now build function `bsequence` as follows:

```
(defun bsequence (n)
  (if (= n 0)
    (cons 0 '())
    (cons n (bsequence(- n 1)))))
```

Example: Our own version of append

- Consider function append2 which takes as its arguments two lists lst1 and lst2 and returns a new list which forms a concatenation of lst1 and lst2.
  - Base case: If lst1 is empty, then return lst2.
  - Recursive case: Return a list containing as its first element the head of lst1 with its tail being the concatenation of the tail of lst1 with lst2.

```
(defun append2 (lst1 lst2)
  (if (null lst1)
    lst2
    (cons (car lst1) (append2 (cdr lst1) lst2))))
```

Example: sum

- Consider function `sum` which takes a list `lst` as its argument and returns the summation of its elements.
  - Base case: If the list is empty, then sum is `0`.
  - Recursive case: Add the head element to the sum of the elements of the tail.
- We can unfold this definition for $sum(\langle 2,4,5\rangle)$ as follows:

$$\begin{aligned} sum(\langle 2,4,5\rangle) &= 2 + sum(\langle 4,5\rangle) \\ &= 2 + 4 + sum(\langle 5\rangle) \\ &= 2 + 4 + 5 + sum(\langle\rangle) \\ &= 2 + 4 + 5 + 0 \\ &= 11 \end{aligned}$$

Example: sum /cont.

```
(defun sum (lst)
  (cond ((null lst) 0)
        (t (+ (car lst) (sum (cdr lst))))))
```

Example: sum /cont.

- We can trace the execution of `(sum '(1 2 3 4 5))` as follows:

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

Example: Finding the last element in a list

- Consider a function `last2` which takes a list `lst` as its argument and returns the last element in the list.
  - Base case: If the list has one element (its tail is the empty list), then return this element.
  - Recursive case: Return the last element of the tail of the list.

```
(defun last2 (lst)
  (cond ((null lst) nil)
        ((null (cdr lst)) (car lst))
        (t (last2 (cdr lst)))))
```

Example: Reversing a list

- Consider function `reverse2` which takes a list as its argument and returns the reversed list.
  - Base case: If the list is empty, then return the empty list.
  - Recursive case: Recur on the tail of the list and the head of the list.

```
(defun reverse2 (lst)
  (cond ((null lst) '())
    (t (append (reverse2 (cdr lst)) (list (car lst))))))
```

Example: cube-list

- Consider a function called `cube-list`, which takes as argument a list of numbers and returns the same list with each element replaced with its cube.

```
(defun cube-list (lst)
  (cond ((null lst) nil)
        (t (let ((elt (car lst)))
          (cons (* elt elt elt)
                (cube-list (cdr lst)))))))
```

Example: Interleaving the elements of two lists

- Consider function `interleave` which takes two lists `lst1` and `lst2` as its arguments and returns a new list whose elements correspond to lists `lst1` and `lst2` interleaved, i.e. the first element is the from `lst1`, the second is from `lst2`, the third from `lst1`, etc.
  - Base cases:
    1. If `lst1` is empty, then return `lst2`.
    2. If `lst2` is empty, then return `lst1`.
  - Recursive case: Concatenate the head of `lst1` with a list containing the concatenation of the head of `lst2` with the interleaved tails of `lst1` and `lst2`.

Example: Interleaving the elements of two lists /cont.

```
(defun interleave (lst1 lst2)
  (cond ((null lst1) lst2)
        ((null lst2) lst1)
        (t (cons (car lst1)
             (cons (car lst2)
                (interleave (cdr lst1) (cdr lst2)))))))
```

Example: Removing the first occurrence of an element in a list

- Consider function `remove-first-occurrence` which takes as arguments a list `lst` and an element `elt`, and returns `lst` with the first occurrence of `elt` removed.
- Base cases:
  1. If `lst` is empty, then return the empty list.
  2. If the head of `lst` is the symbol we want to remove then return the tail of `lst`.
- Recursive case: Keep the head of `lst` and recur on the tail of `lst`.

Example: Removing the first occurrence of an element in a list /cont.

```
(defun remove-first-occurrence (lst elt)
  (cond ((null lst) nil)
        ((equal (car lst) elt) (cdr lst))
        (t (cons (car lst)
                 (remove-first-occurrence (cdr lst) elt)))))
```

Example: Removing the first occurrence of an element in a list /cont.

- Let us trace the execution of `(remove-first-occurrence '(a e b c d e) 'e)`:

```
(remove-first-occurrence '(a e b c d e) 'e)
   = (cons 'a ((remove-first-occurrence '(e b c d e) 'e))
   = (cons 'a '(b c d e))
   = '(a b c d e)
```

Example: Removing all occurrences of an element in a list

- Consider function `remove-all-occurrences` which takes as arguments a list `lst` and an element `elt`, and returns `lst` with all occurrences of `elt` removed.
- Base case: If `lst` is empty, return the empty list.
- Recursive cases: There are two cases to consider when the list is not empty.
  1. When the head of the list is the same as `elt`, ignore the head of the list and recur on removing `elt` from the tail of the list.
  2. When the head of the list is not the same as `elt`, keep the head and recur on removing `elt` from the tail of the list.

Example: Removing all occurrences of an element in a list /cont.

```
(defun remove-all-occurrences (lst elt)
  (if (null lst)
    nil
    (if (equal (car lst) elt)
      (remove-all-occurrences (cdr lst) elt)
      (cons (car lst) (remove-all-occurrences (cdr lst) elt)))))
```

Example: Merge two lists

- Consider function `merge2` which takes as its arguments two sorted lists of non-repetitive numbers and returns a merged list with no redundancies.
- Base cases:
  1. If `lst1` is empty, then return `lst2`.
  2. If `lst2` is empty, then return `lst1`.
- Recursive cases:
  1. If the head of `lst1` equals to the head of `lst2` then ignore this element and recur on the tail of `lst1` and `lst2`.
  2. If the head of `lst1` is less than the head of `lst2`, then keep this element and recur on the tail of `lst1` and `lst2`.
  3. Otherwise keep the head of `lst2` and recur on `lst1` and the tail of `lst2`.

Example: Merge two lists /cont.

```
(defun merge2 (lst1 lst2)
  (cond ((null lst1) lst2)
        ((null lst2) lst1)
        ((= (car lst1) (car lst2)) (merge2 (cdr lst1) lst2))
        ((< (car lst1) (car lst2))
                  (cons (car lst1) (merge2 (cdr lst1) lst2)))
        (t (cons (car lst2) (merge2 lst1 (cdr lst2))))))
```

Example: The Fibonacci sequence /cont.

- We define function `fibonacci` which takes as its argument a nonnegative integer k and returns the k^{th} Fibonacci number F_k.

```
(defun fibonacci (k)
  (if (or (zerop k) (= k 1))
    k
    (+ (fibonacci (- k 1)) (fibonacci (- k 2)))))
```

- The program is rather slow. The reason for this is that F_k and F_{k-1} both must compute F_{k-2}.

Some guidelines on defining functions

- Unless the function is trivial, break the logic into cases (multiple selection) with `cond`.
- When handling lists, you would normally adopt a recursive solution. Treat the empty list as a base case.
- Normally you would operate on the head of a list (accessible with `car`) and recur on the tail of the list (accessible with `cdr`).
- To delete the head of the list, simply recur on the tail of the list.
- To keep the head of the list as is, use `cons` to place it as the head of the returning list (whose tail is determined by the recursive call).
- Use `else` (or `t`) to cover remaining (and to protect against forgotten) cases.

Example: Determining a subset relation

- Consider function `issubsetp` which takes as arguments two lists representing sets, `set1` and `set2`, and returns true if `set1` is a subset of `set2`. Otherwise, it returns false (nil).
- Base case: If `set1` is empty, then return true.
- Recursive case: If the first element of `set1` is a member of `set2`, then recur on the rest of the elements of `set1`, otherwise return false (nil).

```
(defun issubsetp (set1 set2)
  (if (null set1)
    t
    (if (member (car set1) set2)
      (issubsetp (cdr set1) set2)
      nil)))
```

Example: Determining set union

- Consider function `setunion` which takes as its arguments two lists `lst1` and `lst2` representing sets and returns the set union.
- Base cases:
  1. If `lst1` is empty, then return `lst2`.
  2. If `lst2` is empty, then return `lst1`.
- Recursive cases:
  1. If the head of `lst1` is a member of `lst2`, then ignore this element and recur on the tail of `lst1`, and `lst2`.
  2. If the head of `lst1` is not a member of `lst2`, return a list which is the concatenation of this element with the union of the tail of `lst1` and `lst2`.

Example: Determining set union /cont.

```
(defun setunion (lst1 lst2)
  (cond
    ((null lst1) lst2)
    ((null lst2) lst1)
    ((member (car lst1) lst2)(setunion (cdr lst1) lst2))
    (t (cons (car lst1) (setunion (cdr lst1) lst2)))))
```

We can execute the function as follows:

```
> (setunion '(a b c d) '(a d))
(B C A D)
```

Example: Determining set intersection

- Consider function `setintersection` which takes as its arguments two lists `lst1` and `lst2` representing sets, and returns a new list representing a set which forms the intersection of its arguments.
- Base case: If either list is empty, then return the empty set.
- Recursive cases:
  1. If the head of `lst1` is a member of `lst2`, then keep this element and recur on the tail of `lst1` and `lst2`.
  2. If the head of `lst1` is not a member of `lst2`, ignore this element and recur on the tail of `lst1` and `lst2`.

Example: Determining set intersection /cont.

```
(defun setintersection (lst1 lst2)
  (cond
    ((null lst1) '())
    ((null lst2) '())
    ((member (car lst1) lst2)
             (cons (car lst1)(setintersection (cdr lst1) lst2)))
    (t (setintersection (cdr lst1) lst2))))
```

We can execute the function as follows:

```
> (setintersection '(a b c) '())
NIL
> (setintersection '(a b c) '(a d e))
(A)
```

Example: Determining set difference

- Consider function `setdifference` which takes as its arguments two lists `lst1` and `lst2` representing sets and returns the set difference.
- Base case: If `lst1` is empty, then return the empty set. If `lst2` is empty, then return `lst1`.
- Recursive cases:
  1. If the head of `lst1` is a member of `lst2`, then ignore this element and recur on the tail of `lst1`, and `lst2`.
  2. If the head of `lst1` is not a member of `lst2`, keep this element and recur on the tail of `lst1` and `lst2`.

Example: Determining set difference /cont.

```
(defun setdifference (lst1 lst2)
  (cond
    ((null lst1) '())
    ((null lst2) lst1)
    ((member (car lst1) lst2)(setdifference (cdr lst1) lst2))
    (t (cons (car lst1) (setdifference (cdr lst1) lst2)))))
```

We can execute the function as follows:

```
> (setdifference '(a b c) '(a d e f))
(B C)
```

Example: Determining set symmetric difference

- Consider function `setsymmetricdifference` which takes as its arguments two lists representing sets and returns a list representing their symmetric difference.
- We can define this function as the difference between the union and the intersection sets, i.e.

$$A \oplus B = (A \cup B) \setminus (A \cap B)$$

```
(defun setsymmetricdifference (lst1 lst2)
  (setdifference (union lst1 lst2)(intersection lst1 lst2)))
```

Example: Determining set symmetric difference /cont.

- Alternatively we can say

$$A \oplus B = (A \backslash B) \ \cup \ (B \backslash A)$$

```
(defun setsymmetricdifference2 (lst1 lst2)
  (union (setdifference lst1 lst2)(setdifference lst2 lst1)))
```

Example: Determining set symmetric difference /cont.

- We can now run the function as follows:

```
> (setsymmetricdifference '(a b c d e f) '(d e f g h))
(H G A B C)
> (setsymmetricdifference2 '(a b c d e f) '(d e f g h))
(H G A B C)
> (setsymmetricdifference '(a b (cd) e) '(e (f h)))
((F H) A B (CD))
> (setsymmetricdifference2 '(a b (cd) e) '(e (f h)))
((F H) A B (CD))
```

Example: Transforming a bag to a set

- Consider function `bag-to-set` which takes as its argument a list representing a bag and returns the corresponding set.
- Base case: If the list is empty, then return the empty list.
- Recursive cases:
  1. If the head of the list is a member of the tail of the list, then ignore this element and recur on the tail of the list.
  2. If the head of the list is not a member of the tail of the list, keep the head element and recur on the tail of the list.

Example: Transforming a bag to a set /cont.

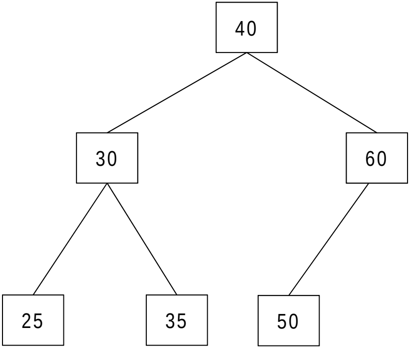
```
(defun bag-to-set (bag)
  (cond ((null bag) '())
        ((member (car bag) (cdr bag)) (bag-to-set (cdr bag)))
        (t (cons (car bag) (bag-to-set(cdr bag))))))
```

Representing trees

- We can use a list to represent a non-empty tree as $\langle atom, \langle l - list \rangle, \langle r - list \rangle \rangle$, where *atom* is the root of the tree, and $\langle l - list \rangle$ and $\langle r - list \rangle$ represent the left and right subtrees respectively.

Example: Binary tree

- Consider the binary tree below.



Example: Binary tree - Translating the representation into Lisp

We can represent the entire tree as one single list:

```
'(40                    ; Root.
    (30                 ; Root of left subtree.
        (25 () ())
        (35 () ())
    )
    (60                 ; Root of right subtree.
        (50 () ())
        ()
    )
 )
```

or `'(40 (30 (25 () ())(35 () ()))(60 (50 () ())()))`

Accessing parts of the tree

- Recall that the entire tree is represented by the list $\langle atom, \langle l - list \rangle, \langle r - list \rangle \rangle$.
- We can obtain the root of the tree by getting the head of the list:

```
> (car '(40 (30 (25 () ())(35 () ()))(60 (50 () ())())))
40
```

Bubble sort

- Consider the implementation of function `bubble-sort` which takes as its argument a list, and returns the same list with its elements sorted in ascending order.
- We first need to build some auxiliary functions, the first one is `bubble` which performs one iteration, thus placing one element in its proper position.

```
(defun bubble (lst)
  (cond ((or (null lst) (null (cdr lst))) lst)
     (( < (car lst) (car (cdr lst)))
            (cons (car lst) (bubble (cdr lst))))
     ( t (cons (car (cdr lst))
            (bubble (cons (car lst) (cdr (cdr lst))))))))
```

Bubble sort /cont.

- Another auxiliary function is `is-sortedp` which returns True or False on whether or not its list argument is sorted.

```
(defun is-sortedp (lst)
  (cond ((or (null lst) (null (cdr lst))) t)
    ((< (car lst) (car (cdr lst))) (is-sortedp (cdr lst)))
    (t nil)))
```

Bubble sort /cont.

- We can now put everything together and define `bubble-sort` as follows:

```
(defun bubble-sort (lst)
  (cond ((or (null lst) (null (cdr lst))) lst)
        ((is-sortedp  lst) lst)
        (t (bubble-sort (bubble lst)))))
```

Linear search

- If x appears in L, then we would like to return its position in the list.

```
(defun search (lst elt pos)
  (if (equal (car lst) elt)
    pos
    (search (cdr lst) elt (+ 1 pos))))

(defun linear-search (lst elt)
  (search lst elt 1))
```

Binary search

- Recall that we can use a list to represent a non-empty tree as $\langle atom, \langle l - list \rangle, \langle r - list \rangle \rangle$, where *atom* is the root of the tree and $l - list$ and $r - list$ represent the left and right subtrees respectively.

```
(defun binary-search (lst elt)
 (cond ((null lst) nil)
   ((= (car lst) elt) t)
   ((< elt (car lst)) (binary-search (car (cdr lst)) elt))
   ((> elt (car lst))
       (binary-search (car (cdr (cdr lst))) elt))))
```