Functional Programming with Common Lisp

Dr. Constantinos Constantinides, P.Eng.

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

May 24, 2021

Lists in functional programming

- A list is the central notion of functional programming.
- We will adopt the Lisp programming language to model, construct and manipulate lists.
- We will adopt a variant of Lisp called Common Lisp.

List representations

```
()              ; The empty list.

(1 3 5 7)       ; A list of four elements,
                ; the numbers 1, 3, 5, and 7.

((1 2)(3 4))    ; A list of two elements, the list (1 2) and
                ; the list (3 4).

(((1 2)(3 4))) ; A list of one element, the list ((1 2)(3 4)).

(a (b 1) 2)     ; A list with three elements: the symbol a,
                ; the list (b 1) and the number 2.
```

Data types

- Every programming language has data types and ways of combining and abstracting them.
- For any data type, we are concerned with:
  1. The values of the type.
  2. The operations on that type.
  3. How the values are represented.

Simple and composite data types

- Data types can be *simple* or *composite*.
- A simple data type (or *atom*) can be a boolean (*true*/*false*), a number (e.g. integer, complex, etc.) or a symbol (a sequence of characters).
- A list is a composite data type.

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:

$$(\mathit{functionName} \quad \mathit{arguments})$$

Expressions and functions /cont.

- When an expression is evaluated, it produces a value (or list of values), which then can be embedded into other expressions.
- In $(- 14 (\times 2\ 3))$, `(* 2 3)` will invoke the $*$ (multiplication) function on the arguments 2 and 3 returning 6 which will in turn become the second argument to the invocation of the `-` (subtraction) function which will return 8.
- This shows that we can invoke Lisp as a calculator.

Expressions and functions /cont.

- As in arithmetic, we can nest expressions.
- Nested expressions are evaluated by reducing the innermost parenthesized expressions to numbers, followed by the next layer, and so on.
- Unlike in regular arithmetic where multiplication has priority over addition the evaluation of prefix expressions is unambiguous. For example, the expression

$$\frac{a - b \times c}{d \times e + f}$$

  is translated in prefix notation as

$$(/\ (-\ a\ (*\ b\ c))\ (+\ (*\ d\ e)\ f))$$

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.

Arity of functions /cont.

- *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*.
- 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
```

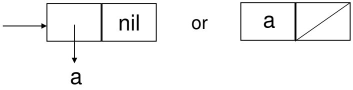
Constructing lists

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

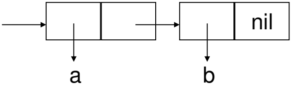
Function cons

- 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.

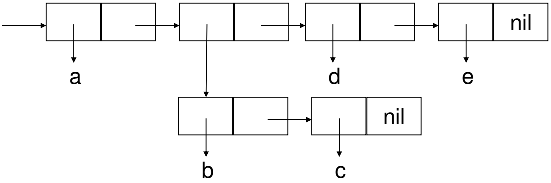
```
> (cons 'a '())
(A)
```




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




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



Function `cons` /cont.

- For an element h and a list L, $cons(h, L)$ denotes the list whose head is h and whose tail is L. Consider the following examples:

$$cons(a, \langle\rangle) = \langle a \rangle$$

$$cons(a, \langle b, c \rangle) = \langle a, b, c \rangle$$

- For any non-empty list L, operations *cons*, *head* and *tail* are related as:

$$cons(head(L), tail(L)) = L$$

Function `cons` /cont.

- The `cons` function takes two arguments and it creates a new list: the first argument becomes the head of the new list and the second argument becomes the tail of the new list.
- If an element is added to an empty list, then `cons` is essentially used to create a list, as in the first of the examples below:

```
(cons 'a '())          ; Returns (a).
(cons 1 '(2 3))        ; Returns (1 2 3).
(cons '(1 2) '(3 4)) ; Returns ((1 2) 3 4).
```

Function `cons`: Adding an element to the front of a list /cont.

- For example, the list `(a)` can be constructed (and represented) as `(cons 'a '())` or `(cons 'a nil)`.

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

Function `cons`: Adding an element to the front of a list /cont.

- The list `(a b)` can be constructed as

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

  or

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

- The list `(a (b c) d e)` can be constructed as

```
> (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.

Advanced mathematical operations

- Lisp provides a number of built-in advanced mathematical operations. For example, `(sqrt a)` returns $\sqrt{a}$, `(expt a b)` returns a^b and `(log a)` returns the natural logarithm of a.

```
> (sqrt 9)
3.0
> (expt 2 3)
8
> (log 10)
2.3025852
```

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.

Example: Single selection

```
(if (listp '(a b c))
    (+ 3 7)
    (+ 1 3))
; Returns 10.
```

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.

$$
\begin{array}{l}
(\ \texttt{let} \\
\qquad\quad (\ (\mathit{binding}_1) \\
\qquad\qquad\ (\mathit{binding}_2) \\
\qquad\qquad\quad \ldots \qquad\quad) \\
\qquad\quad (\mathit{expression}) \quad)
\end{array}
$$

where $(\mathit{binding}_n)$ is of the form $(\mathit{variable}_n\ \mathit{value})$.

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
```

Example: Obtaining the third element from a list

- Consider function `third2` (apparently function `third` is built-in) which takes a list as an argument and returns its third element. The third element is the head of the tail of the tail of the original list.

  ```
  (defun third2 (lst)
    (car (cdr (cdr lst))))
  ```

- We can execute the function as follows:

  ```
  > (third2 '(a b c d))
  C
  > (third2 '(a (b c) (d e f) (g)))
  (D E F)
  ```

Developing variable arity functions with rest parameters

- We can write a function of variable arity and we can do this through a *rest* parameter.
- The token `&rest` before the last parameter in the parameter list, makes this last parameter a list that will contain all the remaining arguments.
- In the following example, we define function `construct-list` that takes any number of arguments and places them in a list. Notice that in the case where no second (or third etc.) argument is provided, the list represented by `args` is empty.

```
(defun construct-list (thing &rest args)
  (cons thing args))
```

Developing variable arity functions with rest parameters /cont.

- Consider the function:

```
(defun construct-list (thing &rest args)
  (cons thing args))
```

- We can execute the function as follows:

```
> (construct-list 'a)
(A)
> (construct-list 'a '())
(A NIL)
> (construct-list 'a 'b 'c 'd)
(A B C D)
> (construct-list 'a '(b c))
(A (B C))
```

Developing variable arity functions with optional parameters

- An *optional parameter* (as opposed to *required*) is one that can be omitted.
- Additionally, an optional parameter can have a default value.
- The implicit default value is `nil`, but we can provide an explicit default.
- In the next example, we leave the default implicit value for the optional parameter `arg`:

```
(defun make-quote (thing &optional arg)
  (list thing arg))
```

```
>(make-quote 'all)
(ALL NIL)
```

Developing variable arity functions with optional parameters /cont.

- Let us now modify the function slightly and also provide an explicit default value to parameter `arg` which we specify by enclosing it in a list with the parameter:

```
(defun make-quote (thing &optional (arg 'die))
  (list thing 'men 'must arg))
```

- We can execute the function as follows:

```
> (make-quote 'all)
(ALL MEN MUST DIE)
> (make-quote 'all 'serve)
(ALL MEN MUST SERVE)
```

Keyword parameters

- A more flexible kind of optional parameter is the *keyword parameter*.
- In a parameter list, all parameters after the `&key` symbol are optional.
- Additionally, they can be identified not by their position in the parameter list, but by symbolic tags that precede them.
- In the following example, function `make-pairs` takes four optional paremeters that combines into a list of two pairs:

```
(defun make-pairs (&key a b c d)
  (list (list a b) (list c d)))
```

- We now can execute the function by passing arguments under symbolic tags that would correspond to the function parameters:

```
> (make-pairs :c 3 :a 5 :d 1 :b 9)
((5 9) (3 1))
```

Keyword parameters /cont.

- As the implicit default is `nil`, consider the following example:

```
> (make-pairs)
((NIL NIL) (NIL NIL))
```

- Consider another execution where we combine implicit defaults and symbolic tags:

```
> (make-pairs :a 7 :d 6)
((7 NIL) (NIL 6))
```

- To specify explicit defaults we have to modify our function:

```
(defun make-pairs (&key a b c (d 'last))
  (list (list a b) (list c d)))
```

- Finally, consider the example where we combine implicit and explicit defaults, and symbolic tags:

```
>  (make-pairs :a 7)
((7 NIL) (NIL LAST))
```

Side effects

- In Computer Science, a function or expression is said to produce a *side effect* if it modifies some state in addition to its return value.
- For example, a function might modify some global variable, modify one of its arguments, write data to a display or file, or read some data from other side-effecting functions.

Pure functions

- A function may be described as *pure* if both of the following statements about the function hold:
- The function always evaluates the same result value given the same argument value(s).
- The evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices.

Examples: Pure and impure functions

- A function `length(string)` is pure because it returns the size of a string.
- A function `today()` is impure because at different times it will yield different results.
- A function `print(arg)` is impure because it causes output as an effect.

Pure functions /cont.

- Pure functions allow optimization of expressions through common subexpression elimination.
- For example, consider $y = f(x) \times f(x)$. The evaluation of $f(x)$ can be costly. A compiler can perform an optimization by factoring out $f(x)$ if it is pure, transforming the program to

$$z = f(x)$$
$$y = z \times z$$

  thus eliminating the second evaluation of $f(x)$.
- If a function is impure, common subexpression elimination is not possible. For example, in $y = \mathit{random}() \times \mathit{random}()$, then the second call to $\mathit{random}()$ cannot be eliminated, because its return value will (most likely) be different from that of the first call.

Referential transparency

- An expression is said to be referentially transparent (as opposed to referentially opaque) if it can be replaced with its value without changing the program (in other words, yielding a program that has the same effects and output on the same input).
- Since referential transparency involves the concept of determinacy (producing the same result for each input), all referentially transparent functions are determinate.
- If all functions involved in the expression are pure functions, then the expression is referentially transparent. In pure functional programming, referential transparency is enforced for all functions.

Examples where referential transparency holds

- `(* 5 5)` can be replaced by 25.
- $sin(x)$ will always give the same result for any given x.

Examples where referential transparency does not hold

- The expression `x++` in languages such as C++ or Java is not transparent, as it changes the value of `x`.
- `System.out.println("Hello world")` cannot be replaced by its value (say, `0`) since *Hello world* will not be displayed.
- Function `today()` cannot be replaced by its value (say, “June 27, 2009”) since it will not yield the same result the day after.

Conditions for referential transparency

- Being side-effect free is necessary but not sufficient for referential transparency.
- Referential transparency implies that an expression (such as a function call) can be replaced with its value; this requires that the expression has no side effects and is determinate.

Idempotence

- Idempotence is a property of a mathematical operation that has the same effect if used multiple times as it does if used only once. For example, the absolute value, $abs()$, function is idempotent, as

$$\begin{aligned} abs(x) &= abs(abs(x)) \\ &= abs(abs(abs(x))) \\ &= ...for\ all\ x. \end{aligned}$$

- In other words, applying *abs* exactly once yields the same result as repeatedly applying *abs* any number of times.

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)
```

Function composition

- We can construct a new function by combining simpler functions.
- Many times we use *composition* of functions even though we may not refer to it explicitly as such.
- The composition of two functions f and g is the function denoted by $f \circ g$ is defined as
$$(f \circ g)(x) = f(g(x))$$
- The composition makes sense only for values of x in the domain of g such that $g(x)$ is in the domain of f.

Examples: Function composition

1. For the list $L = \langle a, b \rangle$, $head(tail(L))$ is a valid function composition, whereas $tail(head(L))$ would not be a valid function composition.
2. For $f(x) = x + 2$ and $g(x) = x^2 - 1$, then $(f \circ g)(x)$ yields $(x^2 - 1) + 2$.

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
```

Copying a function: copy-list

- The function `copy-list` takes a list and returns a copy of it.

```
> (setf w (copy-list x))
(A B C D E)
> (eql x w)
NIL
> (equal x w)
T
```

Copying a function: Our version of copy-list

- We can define our own function to copy a list, as follows:

```
(defun copy-list2 (lst)
  (if (atom lst)
    lst
    (cons (car lst) (copy-list2 (cdr lst)))))
```

```
> (setf k '(a b (cd) (e f g)))
(A B (CD) (E F G))
> (setf l (copy-list2 k))
(A B (CD) (E F G))
> (eql k l)
NIL
> (equal k l)
T
```

Creating a modifying a list

- We can use `setf` to modify a list. Consider the example below:

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

Control flow: loops

- The `loop` form repeats until some condition is satisfied or when an explicit exit statement is encountered.
- This form allows you not to specify a condition, thus creating an infinite loop as follows:

```
(loop (print "Inside an infinite loop!"))
```

- The above is obviously bad programming.

Forcing an exit from a loop

- A `return` from anywhere inside the `loop` will cause control to exit the loop; any value you specify becomes the value of the loop form.
- The example below will display "Inside a loop" and return 7.

```
(loop
  (print "Inside a loop.")
  (return 7)
  (print "Will never reach here."))
```

return

- `return` can be used in a conditional form to determine when the loop should terminate, as follows:

```
(let ((n 0))
  (loop
    (when (> n 3) (return))
    (print n) (write (* n n n))
    (incf n)))
```

```
0 0
1 1
2 8
3 27
NIL
```

dotimes

- The `dotimes` form repeats for some fixed number of iterations:

```
(dotimes (n 3)
  (print n)
  (write (* n n n)))
```

```
0 0
1 1
2 8
NIL
```

Blocks: progn

- There are three basic operations for creating blocks of code: `progn`, `block`, and `tagbody`.
- With `progn`, the expressions within its body are evaluated in order, and the value of the last is returned:

```
(progn
   (format t "x")
   (format t "y")
   (+ 1 2))
```

```
xy
3
```

Blocks: block

- A `block` is like a `progn` with a name and an emergency exit.
- The first argument should be a symbol and it becomes the name of the block.
- At any point within the body you can halt evaluation and return a value immediately by using `return-from` with the block's name.
- The second argument to `return-from` is returned as the value of the block named by the first.
- Expressions after the `return-from` are not evaluated.

```
(block my-label
   (format t "Inside a block.")
   (return-from my-label 'Exit)
   (format t "We will never see this."))
```

```
Inside a block.
Exit
```

Blocks: tagbody with go

- Within `tagbody` you can use go, a statement which instructs execution to jump to the line containing an atom which appears inside the body and interpreted as a label. Consider the following example:

```
(tagbody
    (setf x 0)
  top
    (setf x (+ x 1))
    (format t "~A" x)
    (if (< x 10) (go top)))
```

```
1 2 3 4 5 6 7 8 9 10
NIL
```

- The statement go is found (usually by its semantic synonym `goto`) in many programming languages.
- It causes an unconditional jump of execution to another statement, identified by a label or a line number (depending on the language).

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 /cont.

- Each recursive case consists of:
- Splitting the data into smaller pieces (for example, with `car` and `cdr`),
- Handling the pieces with calls to the current method (note that every possible chain of recursive calls must eventually reach a base case), and
- 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: $f : \mathbb{N} \rightarrow \mathit{lists}(\mathbb{N})$

- Suppose we need to define the function $f : \mathbb{N} \rightarrow \mathit{lists}(\mathbb{N})$ 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

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

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

- We can execute the function as follows:

```
> (bsequence 9)
(9 8 7 6 5 4 3 2 1 0)
> (bsequence 13)
(13 12 11 10 9 8 7 6 5 4 3 2 1 0)
> (bsequence 0)
(0)
> (bsequence 3)
(3 2 1 0)
```

Example: Prime numbers

- An integer $p > 1$ is called *prime* if it cannot be the product of two integers greater than 1, or alternatively if its only positive factors are 1 and itself.
- Positive integers which can be expressed as the product of two integers greater than 1 are called *composite*.

Example: Greatest common divisor

- The *greatest common divisor (gcd)* of two integers a and b (not both zero) is the largest integer d that is a divisor both of a and of b.
- Consider function `gcd`:

```
(defun gcd (a b)
  (cond ((equal a b) a)
        ((> a b) (gcd (- a b) b))
        (t (gcd a (- b a)))))
```

Example: Relative primality

- Two numbers are *relatively prime* (or *coprime*) if their greatest common divisor (gcd) is 1.
- Consider a predicate function `coprimep` which determines whether or not two positive integer numbers *a* and *b* are coprime.

```
(defun coprime (a b)
  (equal (gcd a b) 1))
```

- We can now run the function as follows:

```
>(coprime 35 64)
T
```

Example: Division remainder

- Consider function `remainder`, which takes as arguments two positive non-zero numbers, n and m, and returns the remainder of the division n/m.
- Base case: If $n < m$ then return n.
- Recursive case: Return the remainder of $(n - m)$ and m.

```
(defun remainder (n m)
  (cond ((< n m) n)
        (t (remainder (- n m) m))))
```

- We can now run the function as follows:

```
> (remainder 3 5)
3
> (remainder 5 3)
2
```

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: Our own version of append /cont.

- We can execute the function as follows:

```
> (append2 '() '(a))
(a)
> (append2 '(a b c) '(d e f))
(a b c d e f)
```

- We can trace the execution of `(append2 '(a b c) '(d e f))` as follows:

```
(append2 '(a b c) '(d e f))
   = cons ('a (append2 '(b c) '(d e f)))
   = cons ('a (cons 'b (append2 '(c) '(d e f))))
   = cons ('a (cons 'b (cons 'c (append2 '() '(d e f)))))
   = cons ('a (cons 'b (cons 'c '(d e f))))
   = '(a b c d e f)
```

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.

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

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

- We can execute the function as follows:

```
> (sum '(1 3 5 7 11 13))
40
> (sum '(0 2 0 0 3))
5
```

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: Finding the last element in a list /cont.

- We can execute the function as follows:

```
> (last2 '(a b 3 4 c d 5 6))
6
> (last2 '(a b (c d 1)))
(C D 1)
```

Example: length2

- Consider a recursive function `length2` which takes a list `lst` as its argument and returns the length of `lst`.
- In defining `length2` we need to verify two things:
  - Base case: If the list is empty, then the length of the list is `0`.
  - Recursive case: Add 1 to the length of the tail.

```
(defun length2 (lst)
  (if (null lst)
    0
    (+ 1 (length2 (cdr lst)))))
```

Example: length2 /cont.

- We can execute the function as follows:

```
> (length2 '(a d c 1 2 3))
6
> (length2 '(a (bc) (1 2 3)))
3
```

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: Reversing a list /cont.

- We can execute the function as follows:

```
> (reverse2 '(a b c d))
(D C B A)
```

Example: Multiplying the elements of a list

- Consider function `product` which takes a list `lst` as its argument and returns the product of its elements.
- This function is very similar to `sum`.
  - Base case: If the list is empty, then the product is 1 (by convention).
  - Recursive case: Multiply the head of `lst` to the product of the elements of the tail.

```
(defun product (lst)
  (cond ((null lst) 1)
        (t (* (car lst) (product (cdr lst))))))
```

Example: Multiplying the elements of a list /cont.

- We can execute the function as follows:

```
> (product '(3 5 7))
105
```

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: cube-list /cont.

- We can execute the function as follows:

```
> (cube-list '(1 3 5))
(1 27 125)
```

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: Interleaving the elements of two lists /cont.

- We can execute the function as follows:

```
> (interleave '() '(1))
(1)
> (interleave '(a b c) '(1 2 3))
(A 1 B 2 C 3)
> (interleave '(a b c d) '(1))
(A 1 B C D)
> (interleave '(a b c) '(1 2 3 4 5))
(A 1 B 2 C 3 4 5)
```

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)))))
```

We can execute the function as follows:

```
> (remove-first-occurrence '(a e b c d e) 'e)
(A B C D E)
```

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

- We can execute the function as follows:

```
> (remove-all-occurrences '(z a z b z z c) 'z)
(A B C)
```

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: Merge two lists /cont.

- We can execute the function as follows:

```
> (merge2 '(2 4 5 8) '(1 9))
(1 2 4 5 8 9)
```

Higher-order recursion

- When a recursive call is the last step in the definition of a recursive method, this is referred to as *tail recursion*.
- All the above examples fall into this category.
- When a recursive function makes more than a single recursive call, we say that the function uses *higher-order recursion*.
- This can be *binary recursion* (two recursive calls, each to solve two similar halves of the problem) or *multiple recursion* (potentially many recursive calls).

Example: The Fibonacci sequence

- The Fibonacci sequence is defined as

$$
\begin{aligned}
F_0 &= 0. \\
F_1 &= 1. \\
F_i &= F_{i-1} + F_{i-2} \textit{ for } i > 2.
\end{aligned}
$$

- We can unfold this definition for F_5 as follows:

$$
\begin{aligned}
F_5 &= F_4 + F_3 \\
&= (F_3 + F_2) + (F_2 + F_1) \\
&= ((F_2 + F_1) + (F_1 + F_0)) + ((F_1 + F_0) + 1) \\
&= (((F_1 + F_0) + F_1) + (F_1 + F_0)) + ((F_1 + F_0) + 1) \\
&= 8
\end{aligned}
$$

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 a subset relation /cont.

- We can now run the function as follows:

```
> (issubsetp '() '(a))
T
> (issubsetp '(a b c) '(a b c d))
T
```

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))
```

Bags (multisets)

- A *bag* (or *multiset*) is a structure which contains a collection of elements. Like a set, the ordering of the elements in not important in a bag. However, unlike a set, repetitions are allowed in a bag.
- Note that since order is not important and repetitions are allowed,

$$\{a, b, b, c\} = \{c, a, b, b\}$$

$$\{a, b, c\} \neq \{c, a, b, b\}$$

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))))))
```

We can execute the function as follows:

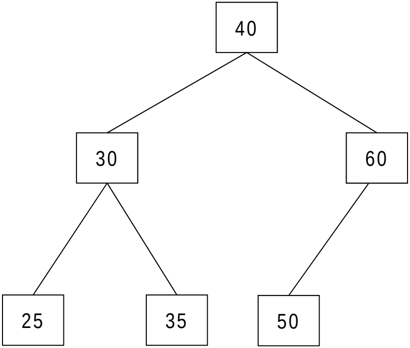
```
> (bag-to-set '(a a b c))
(A B C)
> (bag-to-set '(a a a b b c b a))
(C B A)
> (bag-to-set '(a b c d))
(A B C D)
```

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

```
'(40                  ; Root.
  (...)               ; Left subtree.
  (...)               ; Right subtree.
 )
```

Example: Binary tree - Translating the representation into Lisp

The left subtree of 40 can be represented as

```
(30                   ; Root of left subtree of 40.
  (...)
  (...)
)
```

Example: Binary tree - Translating the representation into Lisp /cont.

The left and right subtrees of 30 can be represented as

```
(25 () ())            ; Left subtree of 30.
(35 () ())            ; Right subtree of 30.
```

where their respective left and right subtrees are null, represented by the empty list.

Example: Binary tree - Translating the representation into Lisp /cont.

The right subtree of 40 can be represented as

```
(60
  (...)                 ; Left subtree of 60.
  ()                    ; Right subtree of 60.
)
```

where the left subtree of 60 can be represented as

```
(50 ()())
```

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
```

Accessing parts of the tree /cont.

- We can obtain the left subtree, $l - list$, of the tree by getting the head of the tail of the list:

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

Accessing parts of the tree /cont.

- We can obtain the right subtree, $r - list$, of the tree by getting the head of the tail of the tail of the list:

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

Example: Exponentiation

- The exponentiation operation, a^n, involves two numbers, the base a and the exponent n. When n is a positive integer, exponentiation corresponds to repeated multiplication.
- We can define $power(a, n)$ as follows:

$$\begin{aligned}
power(a, 0) &= 1 & & \\
power(a, 1) &= a & &= a \times power(a, 0) \\
power(a, 2) &= a \times a & &= a \times power(a, 1)
\end{aligned}$$

- We can then define a recursive pattern as follows:
- Base case: $power(a, 0) = 1$
- Recursive case: $power(a, n) = a \times power(a, n - 1)$

Example: Exponentiation /cont.

- We can unfold the definition of $\mathit{power}(3,4)$ as follows:

$$\begin{aligned}
\mathit{power}(3,4) &= 3 \times \mathit{power}(3,3) \\
&= 3 \times 3 \times \mathit{power}(3,2) \\
&= 3 \times 3 \times 3 \times \mathit{power}(3,1) \\
&= 3 \times 3 \times 3 \times 3 \times \mathit{power}(3,0) \\
&= 3 \times 3 \times 3 \times 3 \times 1 \\
&= 81.
\end{aligned}$$

Example: Exponentiation /cont.

- We can now define function `power` as follows:

```
(defun power (a n)
  (if (zerop n)
    1
    (* a (power a (- n 1)))))
```

- We can execute the function as follows:

```
> (power 3 0)
1
> (power 3 2)
9
> (power 3 4)
81
```

Example: Cartesian system

- For two points (x_1, y_1) and (x_2, y_2), the distance between them is given by
$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Example: Cartesian system /cont.

- A point on the Cartesian plane can be represented as a two-element list.
- The first element of the list represents the x coordinate and it can be obtained by the head of the list.
- The second element of the list defines the y coordinate and it can be accessed as the head of the tail of the list.
- We can define function `second2` to take as its argument a Cartesian point and return the y coordinate:

```
(defun second2 (lst)
  (car (cdr lst)))
```

Example: Cartesian system /cont.

- We can now use `second2` as an auxiliary to function `distance`, which takes as arguments two two-atom lists, each one representing a point on the Cartesian plane. The function returns the distance between the points.
- To improve readability, we will use `first` in place of the (admittedly less readable) `car`.

```
(defun distance (p1 p2)
  (sqrt (+ (expt (- (first p1) (first p2)) 2)
           (expt (- (second2 p1) (second2 p2)) 2))))
```

Example: Cartesian system /cont.

- We can execute the function as follows:

```
> (distance '(0 0) '(2 2))
2.828427
```

Example: Factorial

- The factorial of an integer number is defined as follows:
- Base case: If the number is zero, return `1`.
- Recursive case: Return the product between `n` the factorial of `n - 1`.

Example: Factorial /cont.

- Consider the unfolding for $f(5)$ as follows:

$$\begin{aligned}
\mathit{factorial}(5) &= 5 \times \mathit{factorial}(4) \\
&= 5 \times 4 \times \mathit{factorial}(3) \\
&= 5 \times 4 \times 3 \times \mathit{factorial}(2) \\
&= 5 \times 4 \times 3 \times 2 \times \mathit{factorial}(1) \\
&= 5 \times 4 \times 3 \times 2 \times 1 \times \mathit{factorial}(0) \\
&= 5 \times 4 \times 3 \times 2 \times 1 \times 1 \\
&= 120.
\end{aligned}$$

Example: Factorial /cont.

- We can now define function `factorial` as follows:

```
(defun factorial (n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

- We can now execute the function as follows:

```
>(factorial 5)
120
```

Sorting

- *Sorting* is a technique that puts the elements of an ordered collection in a certain order.

Bubble sort

- Bubble sort is based on successive pairwise comparisons between elements of a collection performed possibly over many iterations.
- Each iteration results in a single element eventually ending up in its proper position (like a bubble moving up).
- We can demonstrate this with an example: Consider the collection $(9, 8, 13, 6)$. The first iteration will work as follows:

Collection	Observations and actions
$(\boxed{9,\ 8}, 13, 6)$	Compare 1st with 2nd. Not in order. Swap them!
$(8, \boxed{9,\ 13}, 6)$	Compare 2nd with 3rd. In order.
$(8, 9, \boxed{13,\ 6})$	Compare 3rd with 4th. Not in order. Swap them!
$(8, 9, 6, \mathbf{13})$	One element (13) has reached its proper position. We have reached the end of the collection and the end of the current iteration.

- Note that the collection is not yet sorted and more iterations are required.

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))))))))
```

- We can test the function as follows:

```
> (bubble '(3 2 1))
(2 1 3)
```

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)))
```

- We can test the function as follows:

```
> (is-sortedp '(2 1 3))
NIL
> (is-sortedp '(1 2 3))
T
```

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)))))
```

- We can execute the function as follows:

```
> (bubble-sort '(4 2 7 5 9))
(2 4 5 7 9)
```

Searching

- *Searching* is a technique to determine whether or not a given element appears in a sorted list of elements.
- We will deploy a list to perform searching.

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))
```

- We can execute the function as follows:

```
> (linear-search '(4 6 1 5 8 9) 9)
6
> (linear-search '(a (bc) d) '(bc))
2
```

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))))
```

Binary search /cont.

- We can execute the function as follows:

```
> (binary-search '() 9)
NIL
> (binary-search '(7 (3 (1 () ())) (9 () ())) 1)
T
> (binary-search '(7 (3 (1 () ())) (9 () ())) 9)
T
> (binary-search '(7 (3 (1 () ())) (9 () ())) 7)
T
> (binary-search '(7 (3 (1 () ())) (9 () ())) 6)
NIL
```