Introduction to Cognitive Robotics

Module 8: An Introduction to Functional Programming with Lisp

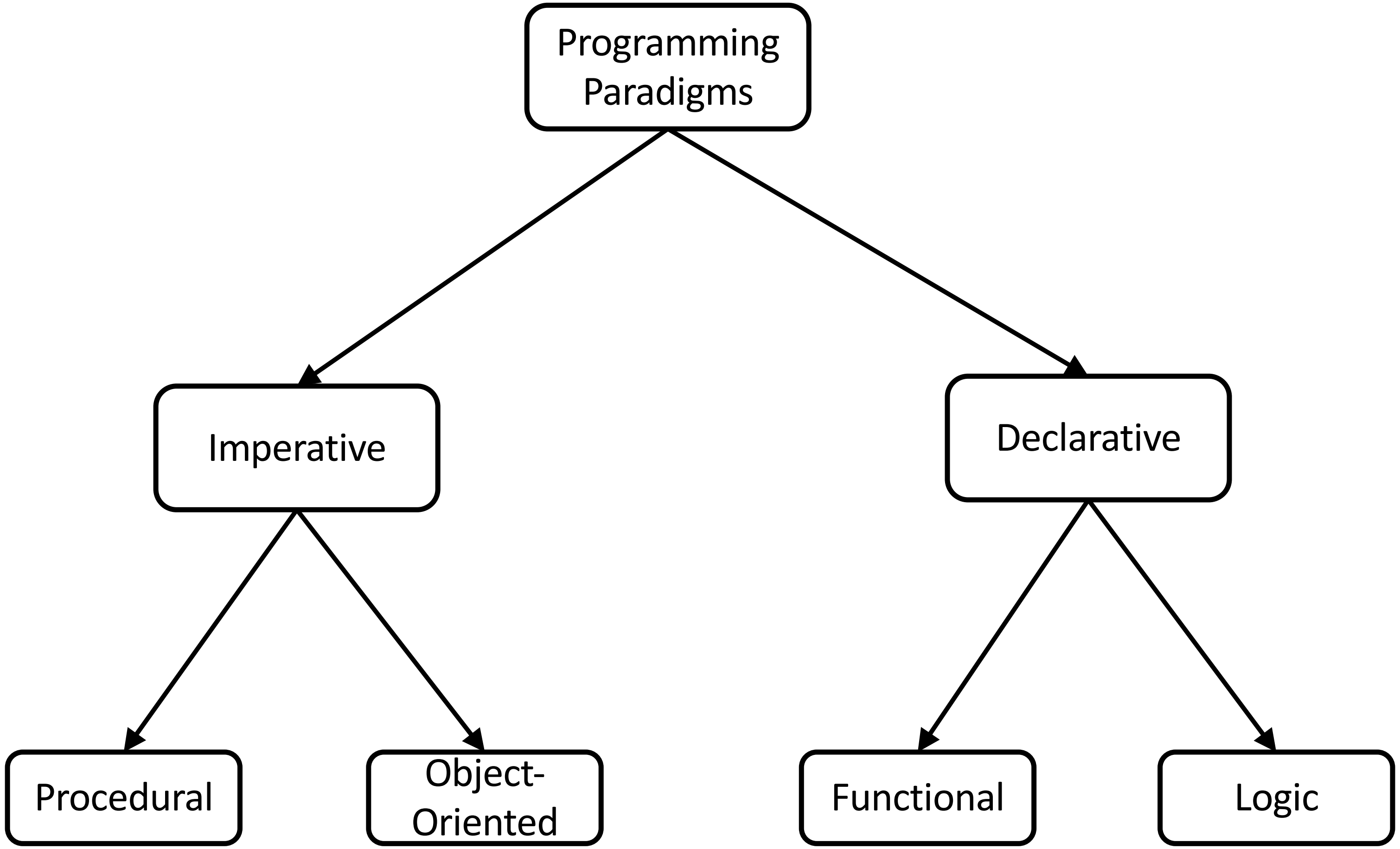
Lecture 1: Common Lisp - REPL, lists, structures, equality, conditionals, CONS, CAR, CDR, dotted and assoc-list

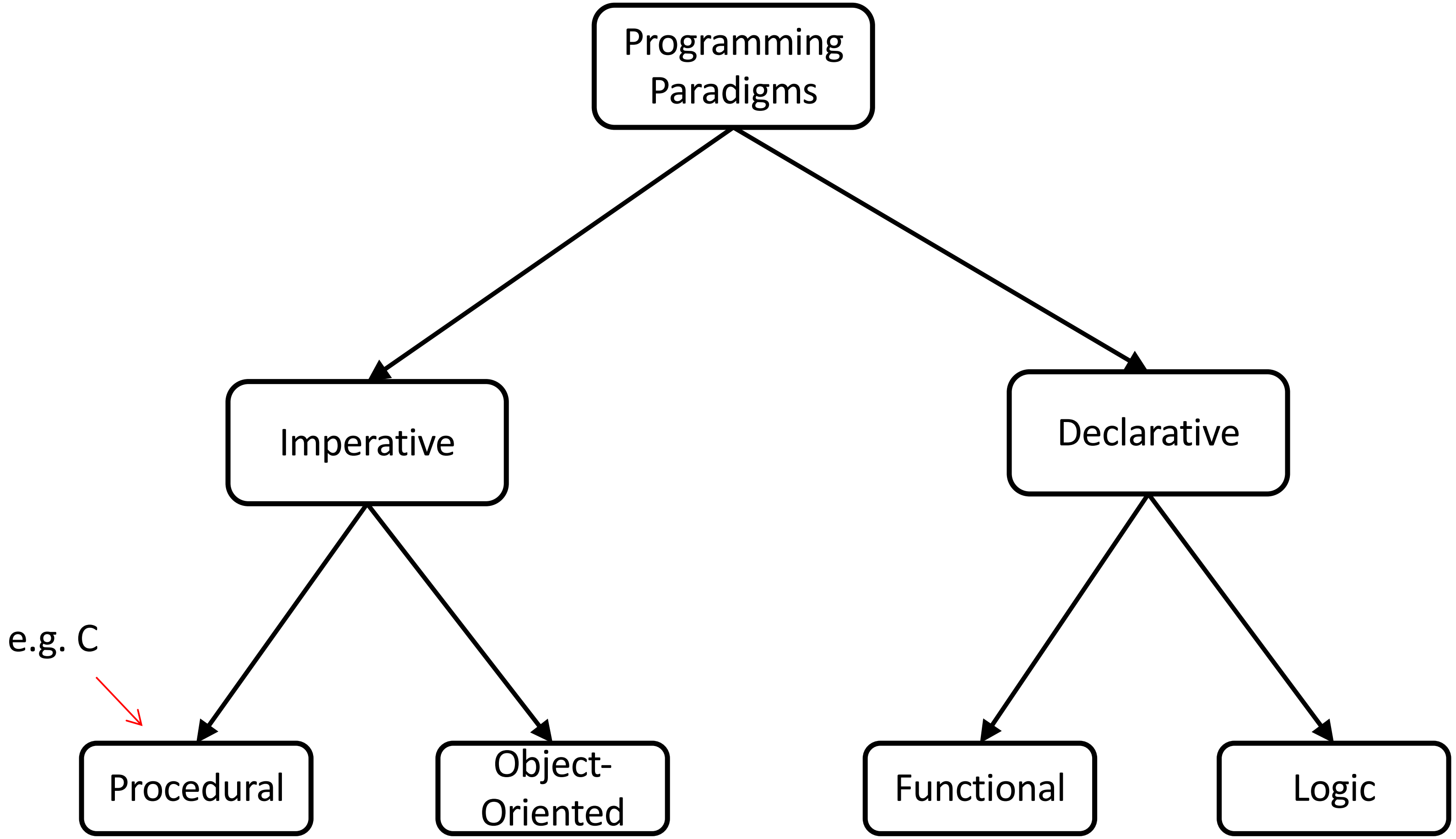
David Vernon
Carnegie Mellon University Africa

www.vernon.eu

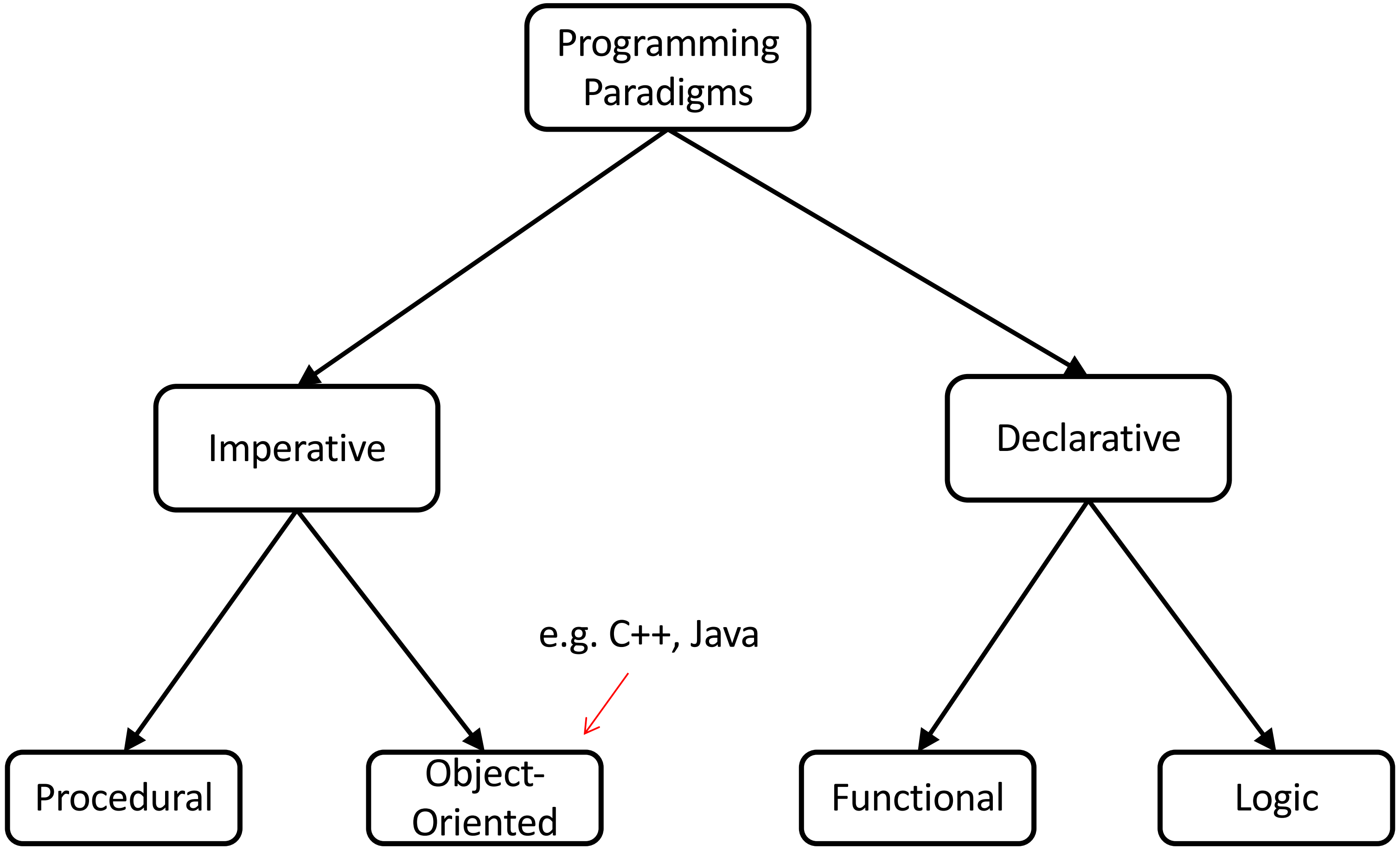
Aside:
Programming Paradigms

Note: This is an oversimplification



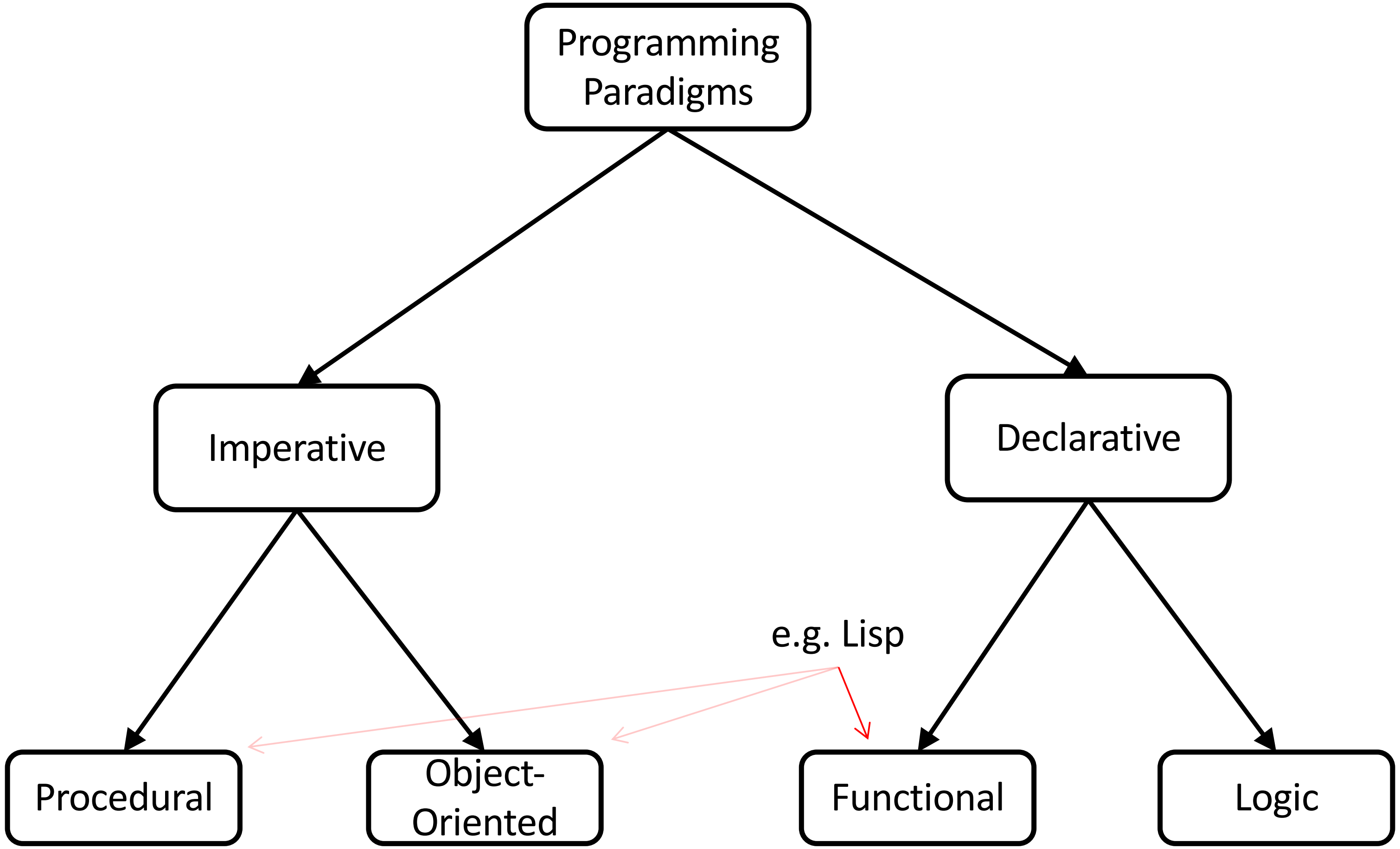


First **do this** and
next **do that**
Focus on how



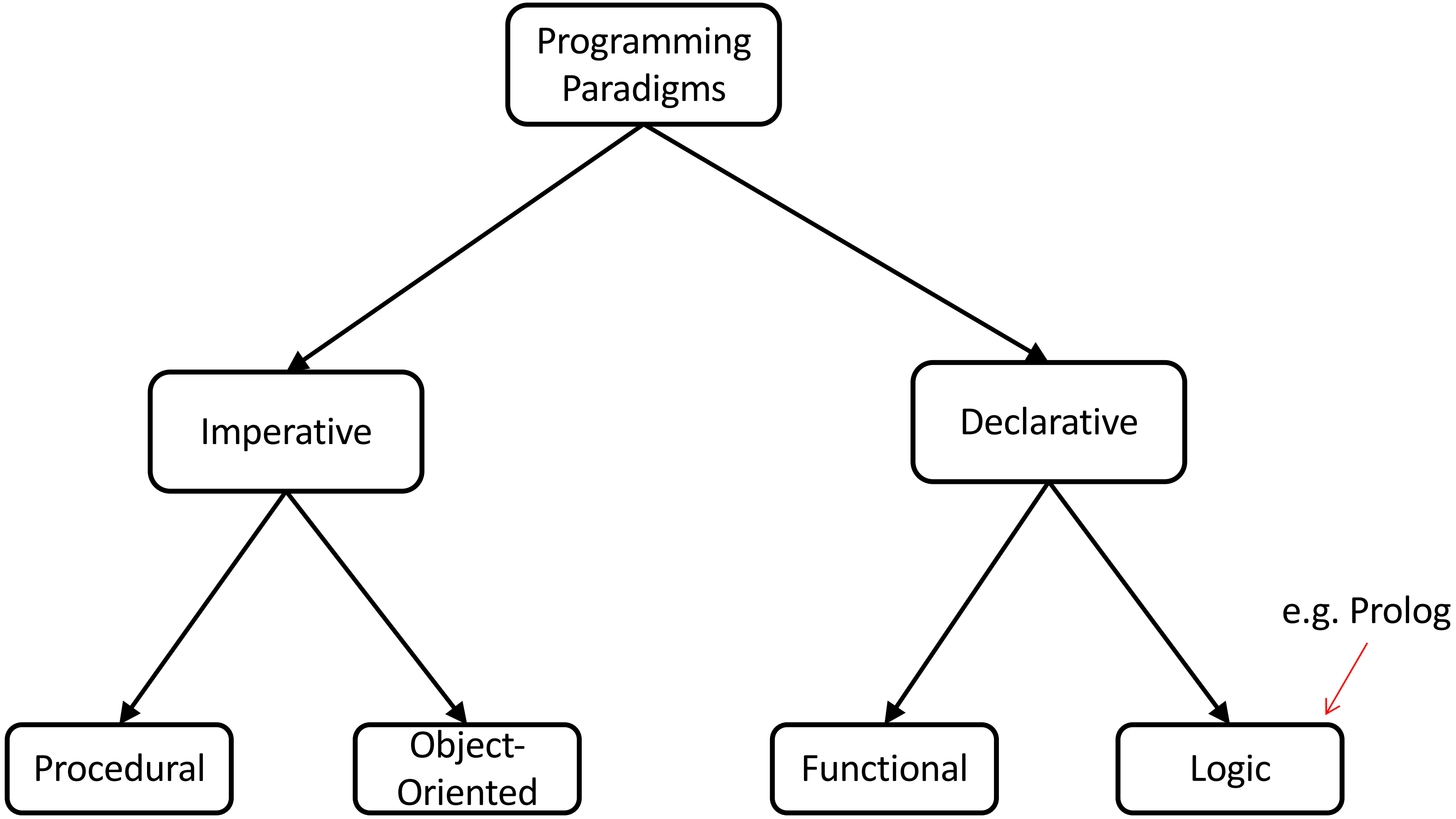
Send messages between objects
to accomplish some task

Focus on how

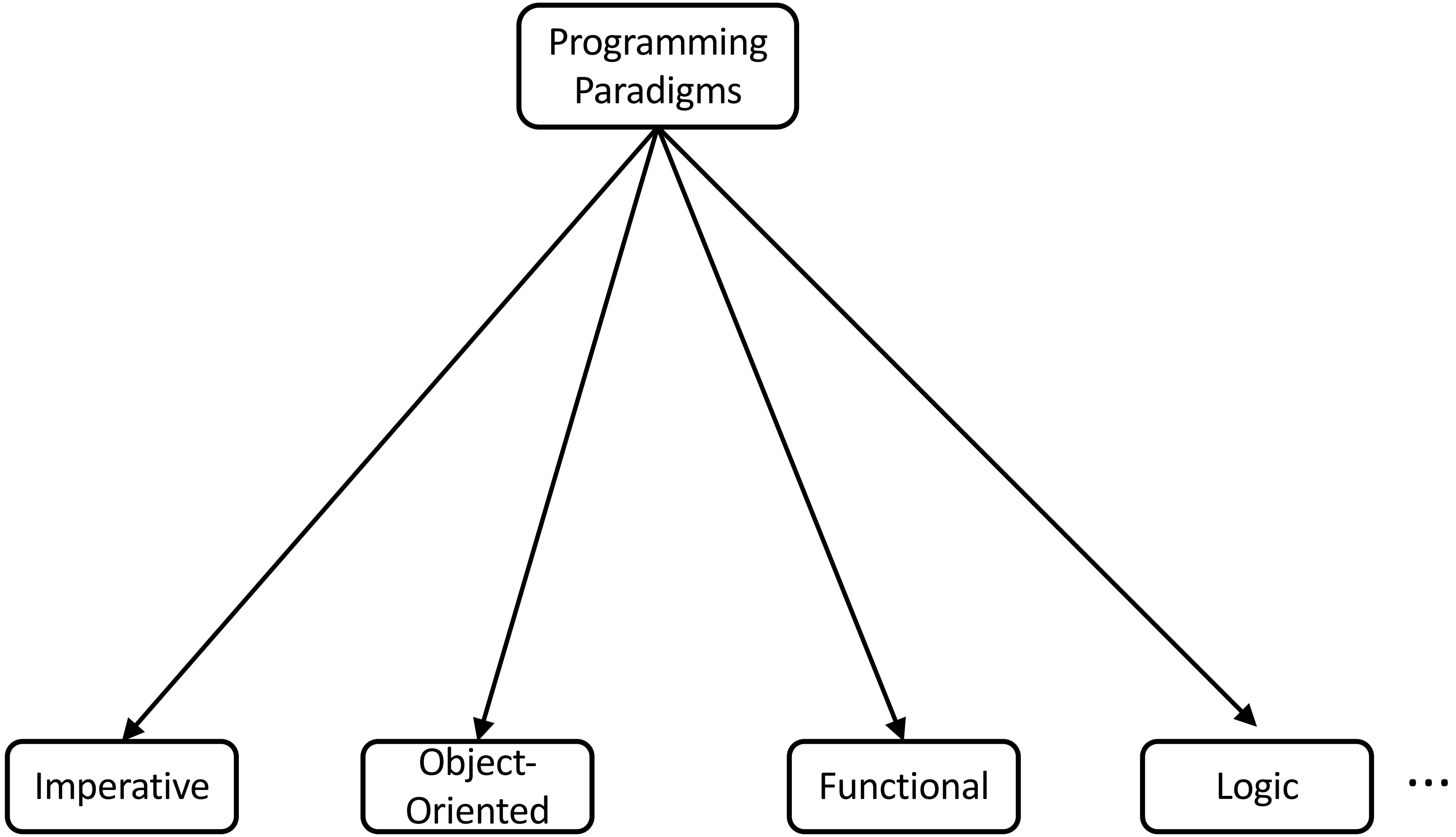


Evaluate an expression and use the resulting value for something

Focus on what



Answer a question using logical deduction based on facts and rules
Focus on what



http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigms.html

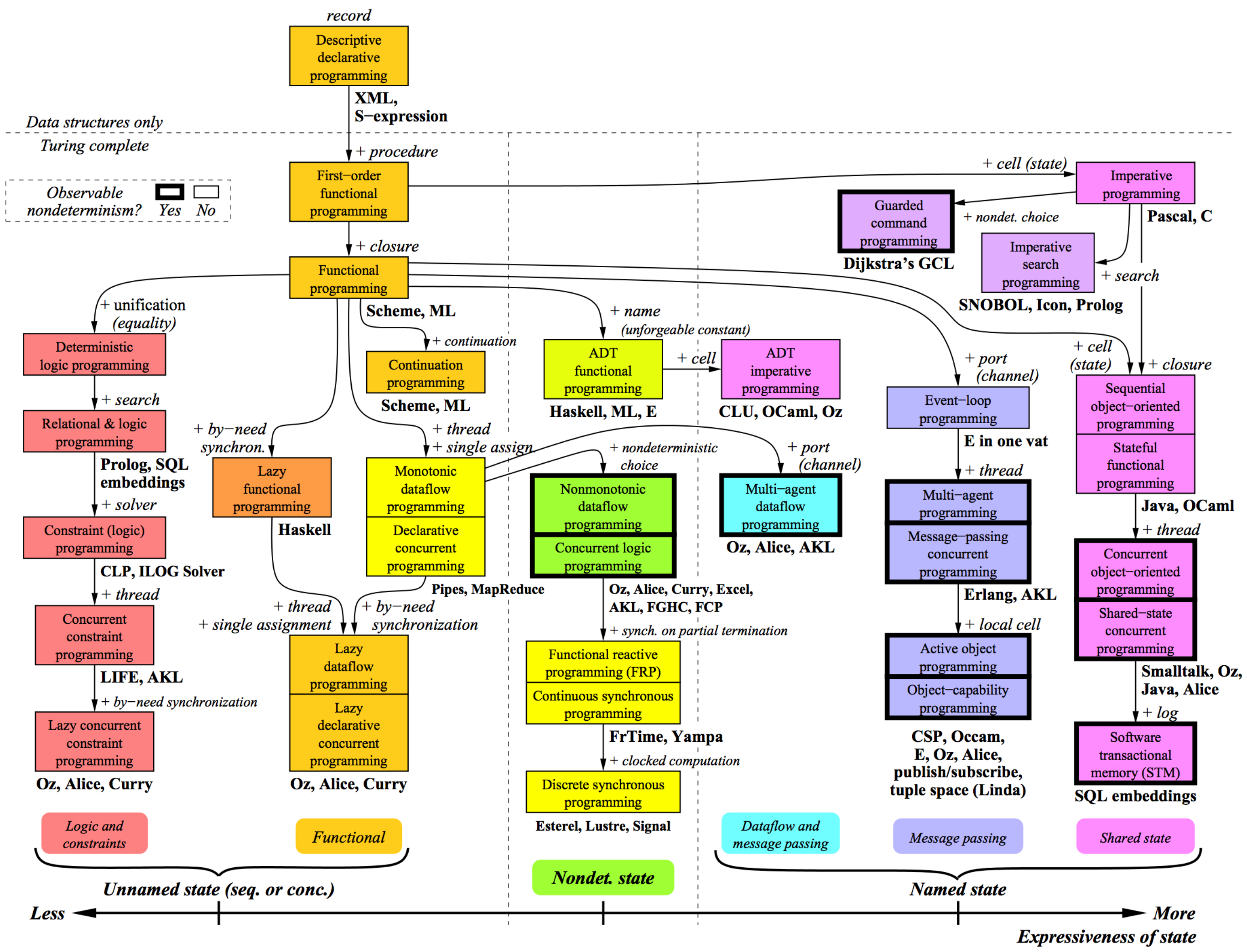


Figure 2. Taxonomy of programming paradigms

Credit: Peter van Roy https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

Abstract

Programming language “paradigms” are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them? This paper argues that we should abandon this method of teaching languages, offers an alternative, reconciles an important split in programming language education, and describes a textbook that explores these matters.

Comment

The book discussed in this paper is available here.

Paper

PDF

Credit: Shriram Krishnamurthi http://cs.brown.edu/~sk/Publications/Papers/Published/sk-teach-pl-post-linnaean/

Programming Paradigms:

Ways of thinking or looking at a problem

(perhaps not so useful as a way of classifying languages)

Essentials of Common Lisp 1

Lisp

- Powerful high-level language, especially good for symbolic and functional programming

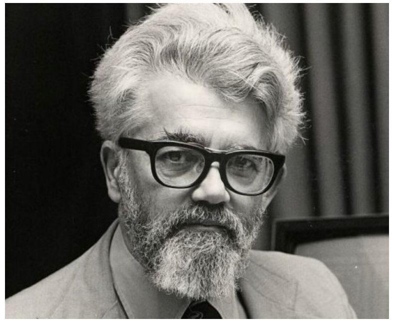
- "Discovered" by John McCarthy in 1958
  - Interpreter written by Steve Russell, one of his students

  - Many dialects over the years … Scheme, Clojure, Emacs Lisp, and Common Lisp, among others

- Lisp is not outdated …
  the tendency is for other languages to develop Lisp-like features (e.g. Python)

Aside …

The term Artificial Intelligence (AI)
was coined by John McCarthy
(Dartmouth Workshop,1956)



- The science and engineering of making intelligent machines

- Every aspect of learning or any other feature of intelligence can be so precisely described that a machine can be made to simulate it

Upsides of Lisp

- Functional programming paradigm: different way of thinking
  - Transcends (& subsumes) imperative and object-oriented paradigms
- Extensible:
  - Suitable for writing domain-specific programming languages
- Facilitates bottom-up programming
  - Extend the language upwards towards the application
  - Makes the application program easier to write
- Interactive development: REPL (Read-Eval-Print Loop)

Downsides of Lisp

- Apparently esoteric appearance and syntax
  - Prefix notation
  - Everything is bracketed (for a reason)
- In this case, first impressions are deceiving
  - The appearance and syntax are very helpful
  - … once you get used to them (it is worth persevering!)

Take-home message

- Lisp opens up a new way of programming:

  as well as writing your program in the language,
  you can extend the language to suit your program.

- Why? Because the Lisp language can be written in Lisp

- For more on the reason Lisp is used in CRAM, read
  http://cram-system.org/doc/package/why_lisp

Sources

The following notes are derived mainly from

P. Graham. *ANSI Common Lisp*, Prentice-Hall, 1996.

- Other sources are credited on the relevant slides
- You are encouraged to read Paul Graham's book
- Other reading material is listed on the last slide
- You are strongly advised to read Chapter 2 which is available online:

  `http://ep.yimg.com/ty/cdn/paulgraham/acl2.txt`

Goal

We focus mainly on those elements that are necessary to

- Read and understand CRAM code
- Write some simple CRAM code
  - We'll do some simple robot programming exercises later
- You'll need to study Lisp in more depth to be a CRAM developer

Toplevel

- Toplevel is the Lisp compiler's interactive front-end
  - It is also referred to as the REPL (Read-Eval-Print Loop)
- You type Lisp expressions into the toplevel, and the system displays their values

Operator

```
CL-USER> (+ 2 3)
5
CL-USER> (+ 2 3 4)
9
CL-USER> (/ (- 7 1) (- 4 2))
3
CL-USER>
```

The List form uses Prefix notation

Arguments

Nested expressions

REPL

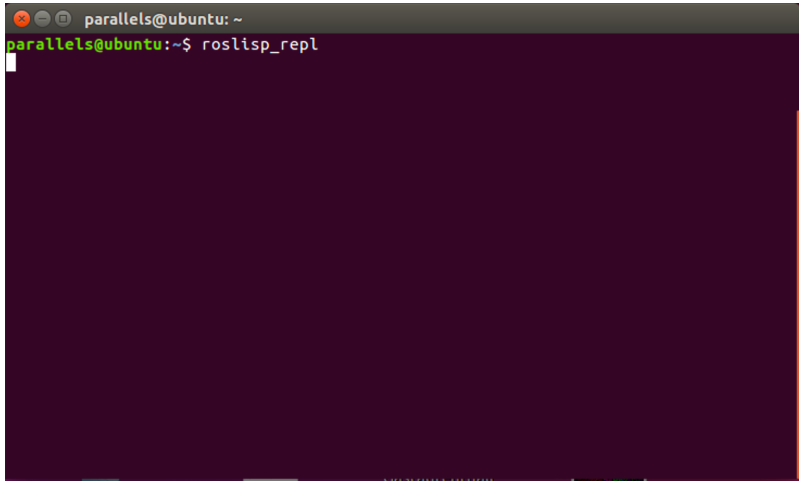
(Read-Eval-Print Loop)

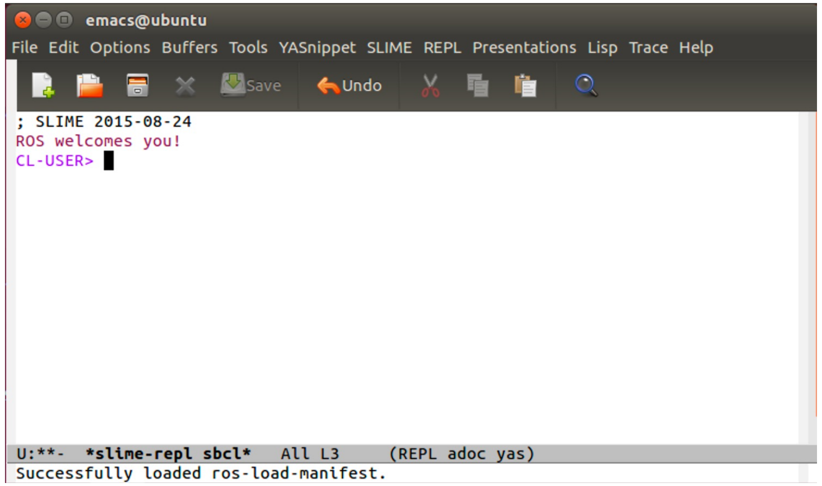
As you go through this set of slides, you might like to try out the examples.

Use `roslisp_repl` to launch the Lisp compiler's interactive front-end: REPL

```
$ roslisp_repl
```

Command entered in terminal





Evaluation

- `+` is a function
- `(+ 2 3)` is a function call
- Order of evaluation
  1. Arguments are evaluated, from left to right
  2. The values of the arguments are passed to the function named by the operator which returns the value of the expression
- If an argument is a function call, it is evaluated using the same rules

Exception to the evaluation rule: the quote function

- The rule for the quote function is “do nothing”

- It takes a single argument and return it verbatim

```
CL-USER> (quote (+ 3 5))
(+ 3 5)
```

close quote mark (apostrophe)

- More common shorthand notation: the single quote

```
CL-USER> '(+ 3 5)
(+ 3 5)
```

- A way of protecting expressions from being evaluated

Selectable exception to the evaluation rule: the backquote function

- Used by itself, a backquote is equivalent to a normal quote

open quote mark

```
CL-USER> `(a b c)
(A B C)
```

- However, within the backquoted expression you can use use , (comma) and ,@ (comma-at) to turn evaluation back on

Selectable exception to the evaluation rule: the backquote function

If you prefix a comma to something within a backquoted expression, it will be evaluated

```
CL-USER> (setf a 1 b 2)
2

CL-USER> `(a is ,a and b is ,b)
(A IS 1 AND B IS 2)
```

```
; SLIME 2015-08-24
ROS welcomes you!
CL-USER> (setf a 1 b 2)
; in: SETF A
;     (SETF A 1)
; ==>
;   (SETQ A 1)
;
; caught WARNING:
;   undefined variable: A
;
; compilation unit finished
;   Undefined variable:
;     A
;   caught 1 WARNING condition

;     (SETF B 2)
; ==>
;   (SETQ B 2)
;
; caught WARNING:
;   undefined variable: B
;
; compilation unit finished
;   Undefined variable:
;     B
;   caught 1 WARNING condition
2
CL-USER> a
1
CL-USER> b
2
```

Note: you will get a warning about undefined variables when you run some of the examples. This is because the variable has not been defined before assigning a value to it. In the next lecture, we will see how to define variables with the let and let* operator (for local variables) and the defparameter operator (for global variables).

Selectable exception to the evaluation rule: the backquote function

Comma-at is like comma, but splices its argument (which should be a list)

```
CL-USER> (setf lst '(a b c))
(A B C)

CL-USER> `(lst is ,lst)
(LST IS (ABC))

CL-USER> `(its elements are ,@lst)
(ITS ELEMENTS ARE A B C )
```

Data

Lisp has all the usual data types and more:

- Integers: 123
- Strings: “Common Lisp”
- Symbols: words (typically variable names), automatically converted to uppercase

```
CL-USER> 'Hello
HELLO
```

- Lists: ...
  (Historically, Lisp was written LISP which derives from "LISt Processor")

Lists

- Zero or more elements (of any type) enclosed in parentheses

```
CL-USER> '(my 2 "Daughters")
(MY 2 "Daughters")
CL-USER> '(the list (a b c) has 3 elements)
(THE LIST (A B C) HAS 3 ELEMENTS)
```

- You can build lists by calling `list`

- Since `list` is a function, its arguments are evaluated

```
CL-USER> (list 'my (+ 2 1) "Daughters")
(MY 3 "Daughters")
```

Lists

- Lisp programs are expressed as lists

- Lisp program generate lists, therefore Lisp programs can generate Lisp code

- Expressions and lists

  – If a list is quoted, evaluation returns the list itself

  – If a list is not quoted, the list is treated as code and evaluation returns its value

```
CL-USER> (list '(+ 2 1) (+ 2 1))
((+ 2 1) 3)
```

Lists

Empty list

```
CL-USER> ()
NIL
CL-USER> nil
NIL
```

List Operations

- The function `cons` builds a list

- If its second argument is a list, it returns a new list with the first argument added to the front of the list

```
CL-USER> (cons 'a '(b c d))
(A B C D)
```

- More on the function `cons` later

List Operations

- The function `car` returns the first element of a list

```
CL-USER> (car '(a b c))
A
```

- The function `cdr` returns the list after the first element of a list

```
CL-USER> (cdr '(a b c))
(B C)
```

- You can use combinations of `car` and `cdr` to reach any element of a list

Specialized Data Structures

- The list is the most versatile data structure in Common Lisp
- There other data structures:
  - Arrays (including vectors and strings)
  - Structures
  - Hash tables
- We focus here on structures

Structures

- Use `defstruct` to define a structure

```
CL-USER> (defstruct point
           x
           y)
```

- This defines a `point` to be a structure with two fields, `x y`
- It also implicitly defines the following functions

`make-point`	returns a new `point`
`point-p`	returns true if the argument is a `point`
`copy-point`	makes a copy of a `point`
`point-x`	returns the value of the `x` field of `point`
`point-y`	returns the value of the `y` field of `point`

Structures

Macro

... and stored in this (generalized) variable

```
CL-USER> (setf p (make-point :x 0 :y 0))
#S(POINT X 0 Y 0)
CL-USER>  (point-x p)
0
CL-USER> (setf (point-y p) 2)
#S(POINT X 0 Y 2)
CL-USER> (point-p p)
T
```

This form is evaluated ...

Truth

- The symbol `t` is the default representation for truth.
- The function `listp` returns true if its argument is a list:

```
CL-USER> (listp '(a b c))
T
```

- A function whose return value is intended to be interpreted as truth or falsity is called a predicate.
  - `t` ... true
  - `nil` (empty list) ... false

Truth

Common Lisp predicates often have names that end with `p`

```
CL-USER> (listp 27)
NIL
```

Why?

Conditional `if`

- Usually takes three arguments:
  - a test expression
  - a then expression
  - and an else expression
- The test expression is evaluated
  - If it returns true, the then expression is evaluated and its value is returned
  - If the test expression returns false, the else expression is evaluated and its value is returned

Conditional `if`

```
CL-USER> (if (listp '(a b c))
             (+ 1 2)
             (+ 5 6))
3
CL-USER> (if (listp 27)
             (+ 1 2)
             (+ 5 6))
11
CL-USER> (if (listp 27) ;no else
             (+ 2 3))
NIL
```

Conditional `if`

- Use `progn` if you want multiple expressions in either the then part or the else part

- `progn` takes any number of expressions, evaluates them in order, and returns the value of the last expression

```
CL-USER> (if (oddp that)
             (progn
               (format t "Hmm, that's odd.")
               (+ that 1)))
```

Conditional `when`

- `when` takes an expression and a body of code
- The body will be evaluated if the test expression returns true

```
CL-USER> (when (oddp that)
           (format t "Hmm, that's odd.")
           (+ that 1))
```

Conditional `unless`

- `unless` takes an expressions and a body of code.

- The body will be evaluated if the test expression returns false

```
CL-USER> (unless (evenp that)
           (format t "Hmm, that's odd.")
           (+ that 1))
```

Logical operators `and` and `or`

- Both take any number of arguments, but only evaluate as many as they need to in order to decide what to return

- If all its arguments are true (that is, not nil), `and` returns the value of the last one:

```
CL-USER> (and t (+ 1 2))
3
```

If one of the arguments is false, none of the remaining arguments are evaluated

- `or` … evaluation of the arguments stops if one of them is true

Equality predicates

`eq`, `eql`, `equal`, and `equalp`

The predicate `eq` returns true only if its arguments are identical

- Two objects are `eq` if they share the same memory
- Characters or numbers are not associated with any particular memory location so `eq` does not apply to them

Equality predicates

`eq`, `eql`, `equal`, and `equalp`

The predicate `eql` returns true for its arguments

- If they are `eq`, or
- If they are numbers of the same type with the same value, or
- If they are character objects that represent the same character
- Default predicate for testing equality

```
CL-USER> (eql (cons 'a nil) (cons 'a nil))
NIL
CL-USER> (setf x (cons 'a nil))
(A)
CL-USER> (eql x x)
T
```

Equality predicates
eq, eql, equal, and equalp

The predicate equal returns true if its arguments have the same content (i.e. would print the same)

- The contents of the arguments just have to be identical

```
CL-USER> (setf x (cons 'a nil))
(A)
CL-USER> (equal x (cons 'a nil))
T
```

Equality predicates
`eq`, `eql`, `equal`, and `equalp`

The predicate `equalp` returns true for the arguments if

- they are `equal`;
- if they are characters and satisfy char-equal, which ignores alphabetic case and certain other attributes of characters
- if they are numbers and have the same numerical value, even if they are of different types; or
- if they have components that are all `equalp`.

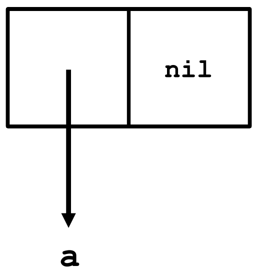
Equality predicates
`eq`, `eql`, `equal`, and `equalp`

- To fully understand the difference between these predicates, we need to know how lists are represented in Lisp
- We have already met the `cons`, `car`, and `cdr` functions
- We said `cons` builds a list as follows:
  - If its second argument is a list,
  - it returns a new list with the first argument added to the front of the list

Equality predicates
`eq`, `eql`, `equal`, and `equalp`

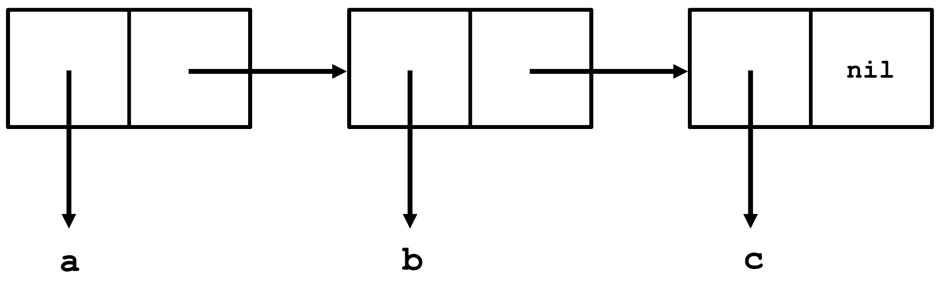
- What `cons` really does is combine two objects into a two-part object called a cons
- Conceptually, a cons is a pair of pointers, the first to the car and the second to the cdr
- The two halves of a cons can point to any kind of object, including a cons
  - One half of the cons points to the first element in the list
  - The other half point to the rest of the list (which is either a cons or `nil`)

Cons



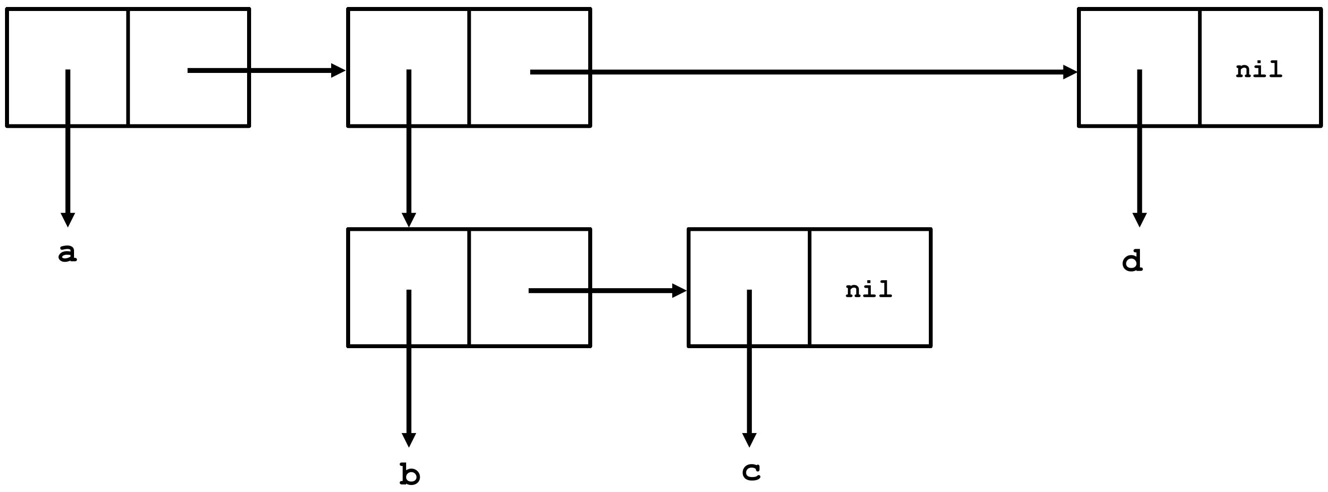

```
CL-USER> (setf x (cons 'a nil))
(A)
CL-USR> (car x)
A
CL-USER> (cdr x)
NIL
```

List



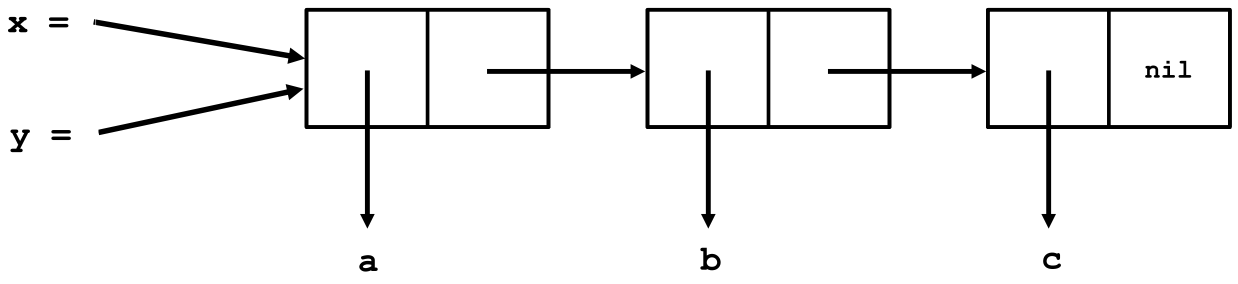

```
CL-USER> (setf y (list 'a 'b 'c))
(A B C)
CL-USER> (cdr y)
(B C)
```

List



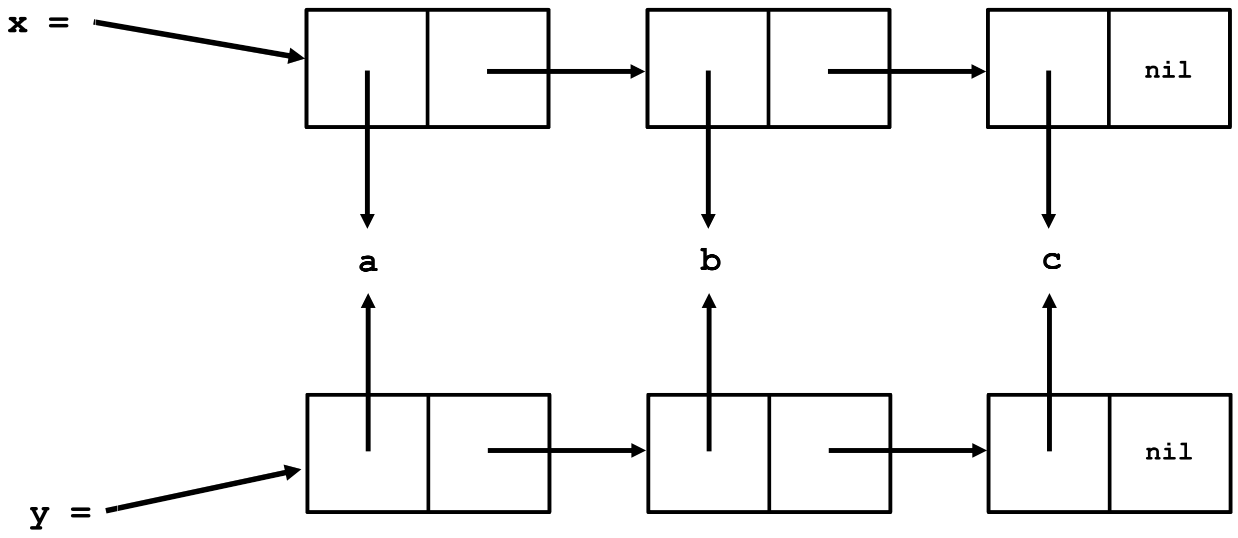

```
CL-USER> (setf z (list 'a (list 'b 'c) 'd))
(A (B C) D)
CL-USER> (car (cdr z))
(B C)
```

List



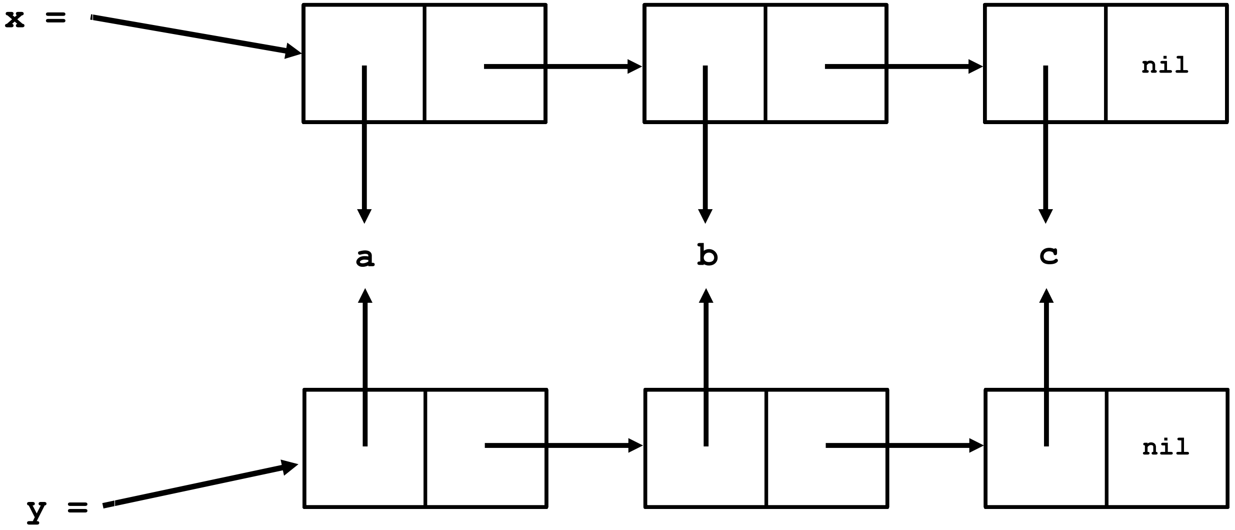

```
CL-USER> (setf x '(a b c))
(A B C)
CL-USER> (setf y x)
(A B C)
CL-USER> (eql x y)
T
```

List




```
CL-USER>  (setf x '(a b c)
                y (copy-list x))
(A B C)
CL-USER> (eql x y)
NIL
CL-USER> (equal x y)
T
```

List




```
CL-USER>  (defun our-copy-list (lst)
            (if (atom lst)
              lst
              (cons (car lst) (our-copy-list (cdr lst)))))
```

Equality predicates

`eq`, `eql`, `equal`, and `equalp`

(eq *object1 object2***)** Function

Returns true iff *object1* and *object2* are identical.

(eql *object1 object2***)** Function

Returns true iff *object1* and *object2* are **eq**, or the same character, or numbers that would look the same when printed.

(equal *object1 object2***)** Function

Returns true iff *object1* and *object2* are **eql**; or are conses whose cars and cdrs are **equal**; or are strings or bit-vectors of the same length (observing fill pointers) whose elements are **eql**; or are pathnames whose components are equivalent. May not terminate for circular arguments.

(equalp *object1 object2***)** Function

Returns true iff *object1* and *object2* are **equal**, **char-equal**, or **=**; or are conses whose cars and cdrs are **equalp**; or are arrays with the same dimensions whose active elements are **equalp**; or are structures of the same type whose elements are **equalp**; or are hash tables with the same test function and number of entries whose keys (as determined by the test function) are all associated with **equalp** values. Reasonable to assume that it may not terminate for circular arguments.

Credit: P. Graham. ANSI Common Lisp, Prentice-Hall, 1996

Equality predicates
`eq`, `eql`, `equal`, and `equalp`

```
CL-USER> (> 2 1.5d0)
T
CL-USER> (<= 3.0d0 3)
T
CL-USER> (eq 1 1)
T
CL-USER> (eq 'bla 'bla)
T
CL-USER> (eq "bla" "bla")
NIL
CL-USER> (eq '(1 2 3) '(1 2 3))
NIL
CL-USER> (eql '(1 2 3) '(1 2 3))
NIL
CL-USER> (eql 1.0 1)
NIL
```

```
CL-USER> (equal '(1 2 3) '(1 2 3))
T
CL-USER> (equal "bla" "bla")
T
CL-USER> (equal "bla" "Bla")
NIL
CL-USER> (equalp "bla" "Bla")
T
CL-USER> (equal #(1 2 3) #(1 2 3))
NIL
CL-USER> (equalp #(1 2 3) #(1 2 3))
T
CL-USER> (= 2.4 2.4d0)
NIL
CL-USER> (string= "hello" "hello")
T
```

Credit: Gayane Kazhoyan, Institute of Artificial Intelligence, University of Bremen

Equality predicates

`eq`, `eql`, `equal`, and `equalp`

x	y	`eq`	`eql`	`equal`	`equalp`
'a	'a	T	T	T	T
0	0	?	T	T	T
'(a)	'(a)	nil	nil	T	T
"ab"	"ab"	nil	nil	T	T
"Ab"	"aB"	nil	nil	nil	T
0	0.0	nil	nil	nil	T
0	1	nil	nil	nil	nil

Credit: Gayane Kazhoyan, Institute of Artificial Intelligence, University of Bremen

Dotted Lists

- The lists we have seen so far, i.e., the ones that are built with `list`, are called proper lists
  - A proper list is either `nil` or a cons whose `cdr` is a proper list

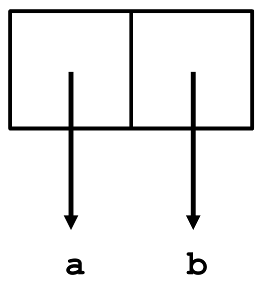
This means the final `cdr` is `nil`

- You can use conses for other things, however
  - Whenever you need a structure with two fields you can use a cons
  - You can use `car` to refer to the first field & `cdr` to refer to the second

Dotted Lists

Because this cons is not a proper list, it is displayed in dot notation

```
CL-USER> (setf pair (cons 'a 'b))
(A . B)
```

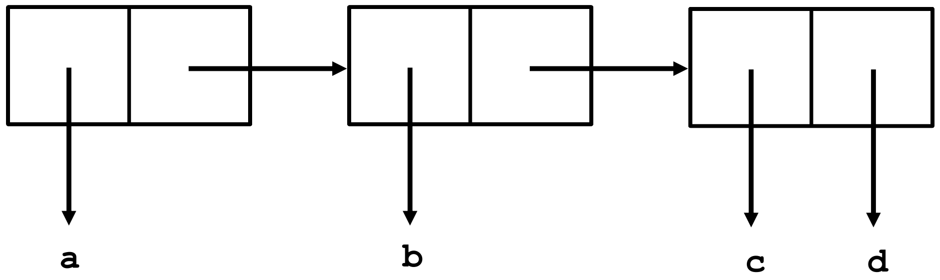


Dotted Lists

A cons that isn't a proper list is called a dotted list

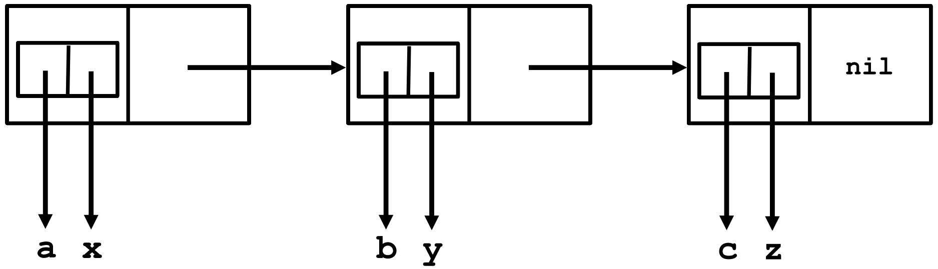
Not a very good name because conses that aren't proper lists are usually not meant to represent lists at all

```
CL-USER> (cons 'a (cons 'b (cons 'c 'd)))
(A B C . D)
```



Assoc-lists

- Conses can also be used to represent mappings
- A list of conses is called an assoc-list or alist



Assoc-lists

Common Lisp has a built-in function, `assoc`, for retrieving the pair associated with a given key

```
CL-USER> (setf trans '((+ . "add") (- . "subtract")))
((+ . "add") (- . "subtract"))
CL-USER> (assoc '+ trans)
(+ . "add")
CL-USER> (assoc '* trans)
NIL
```

Assoc-lists

You can use optional keyword arguments with `assoc`

`:test some-arg` is used to specify the test for equality

`:key some-arg` is used to specify a function to be applied to each element before comparison

Assoc-lists

Assoc-lists

```
CL-USER> (setf names (cons (cons "Alice" "Jones”)
                           (cons (cons "Bill" "Smith")
                                 (cons (cons "Cathy" "Smith")
                                       nil))))
(("Alice" . "Jones") ("Bill" . "Smith") ("Cathy" . "Smith"))
CL-USER> '(("Alice" . "Jones") ("Bill" . "Smith") ("Cathy" . "Smith"))
(("Alice" . "Jones") ("Bill" . "Smith") ("Cathy" . "Smith"))
CL-USER> (assoc "Alice" names)
NIL
CL-USER> (assoc "Alice" names :test #'string=)
("Alice" . "Jones")
CL-USER> (assoc ”alice" names :test #'string=)
NIL
CL-USER> (assoc ”alice" names :test #'string-equal)
("Alice" . "Jones")
CL-USER> (rassoc "Smith" names :test #'string=)
("Bill" . "Smith")
```

Note the sharp quote for the function
see notes on functions below

Recommended Reading

P. Graham. *ANSI Common Lisp*, Prentice-Hall, 1996, Chapter 2.

`http://ep.yimg.com/ty/cdn/paulgraham/acl2.txt`

The Lisp pages on Paul Graham's website:
`http://paulgraham.com/lisp.html`
especially the following:

What Made Lisp Different:

`http://paulgraham.com/diff.html`

Revenge of the Nerds (Essentially, the story of Lisp)

`http://paulgraham.com/icad.html`