Basic Lisp Operations

Function invocation

- It is an S-expression – just another list!

  (function arg1 arg2 ... argN)

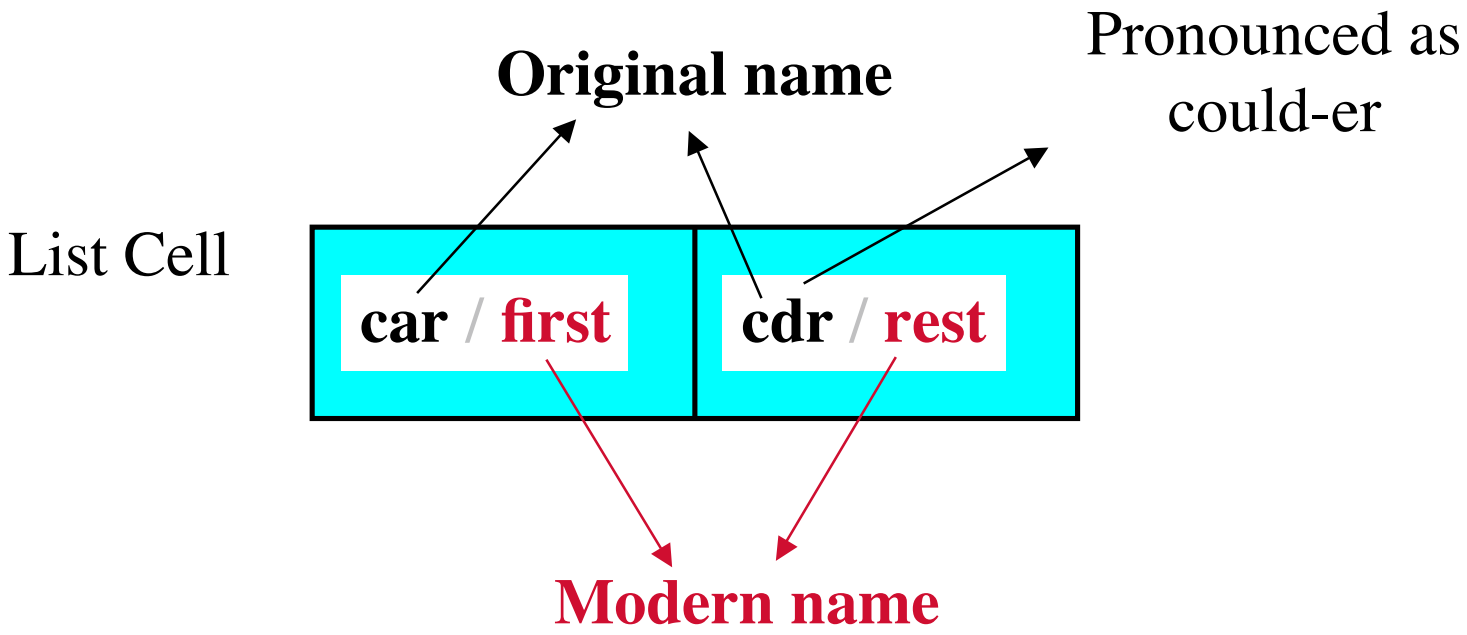
- First list item is the function – **prefix notation**
- The other list items are the arguments to the function.
- Arguments can themselves be lists
  - **(+ 1 2 3 (+ 4 5 6) 7 8 9) ==> 45**
  - **Outer + has 7 arguments, inner + has 3 arguments**
  - **Arguments are evaluated before the function**

Basic Functions

Can build Lisp out of these functions

- List access & creation
  - **car or first – access first in list**
  - **cdr or rest – access all but first**
  - **cons – construct a list cell**
- Other
  - **quote or ' – take literally, do not interpret**
  - **atom – true if argument is an atom**
  - **eq – true if arguments are same object**
  - **cond – conditional**
    generalized "if ... then ... else ...."

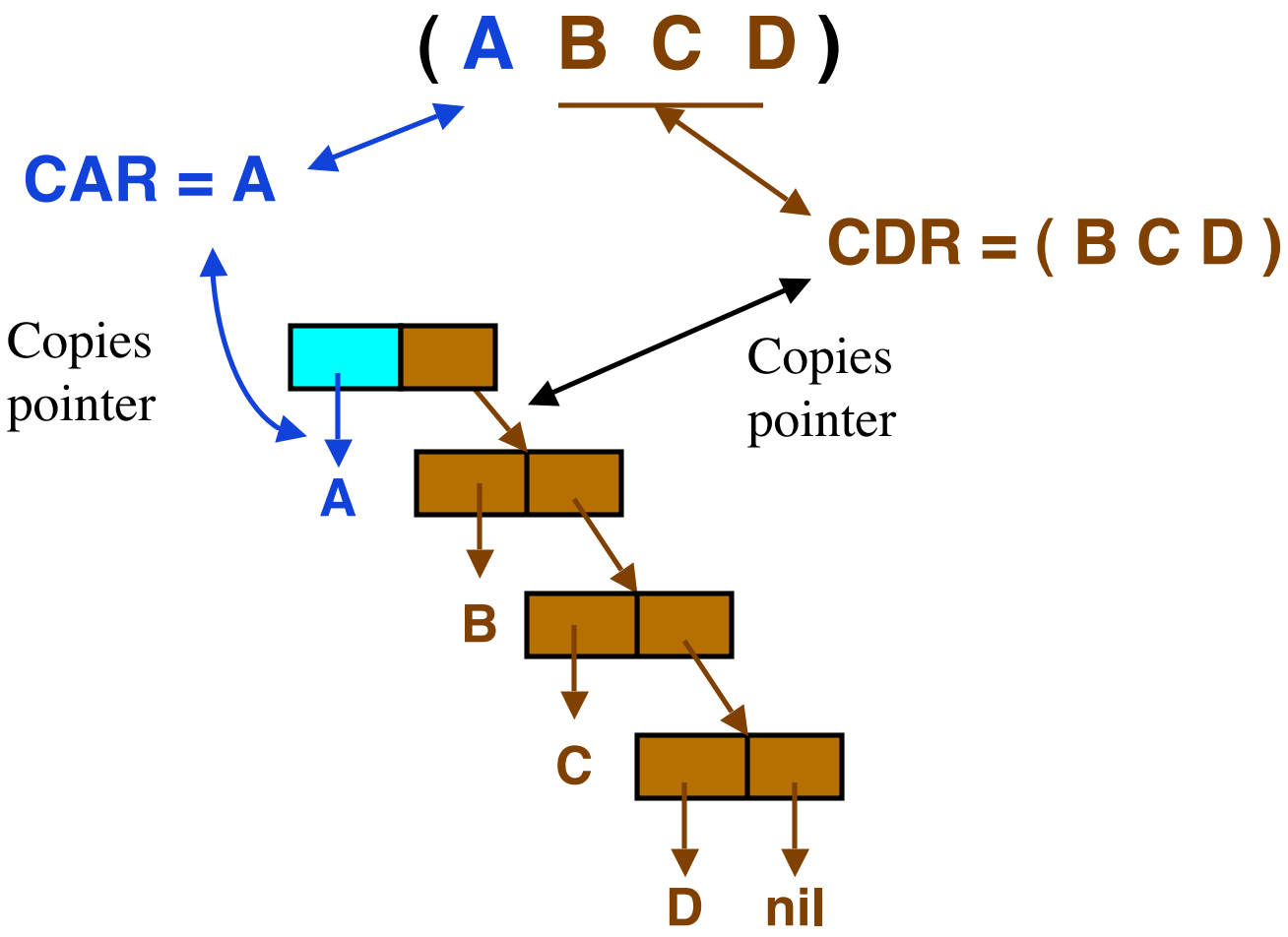
List access functions



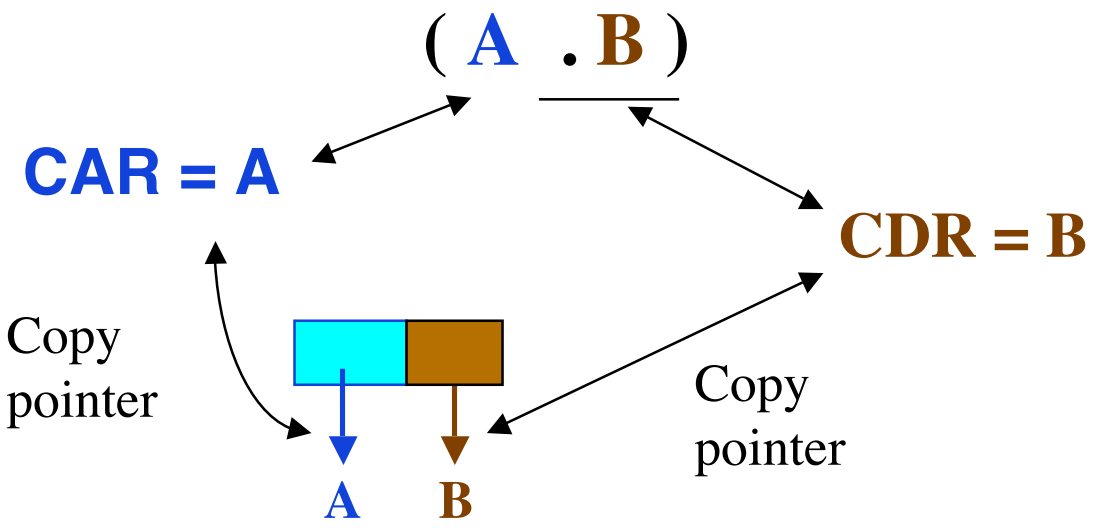
(car '(a b c)) ≡ (first '(a b c)) ≡ a

(cdr '(a b c)) ≡(rest '(a b c)) ≡(b c)

CAR and CDR – Structural View 1



CAR and CDR – Structural View 2



(car '(a b c)) – why the quote?

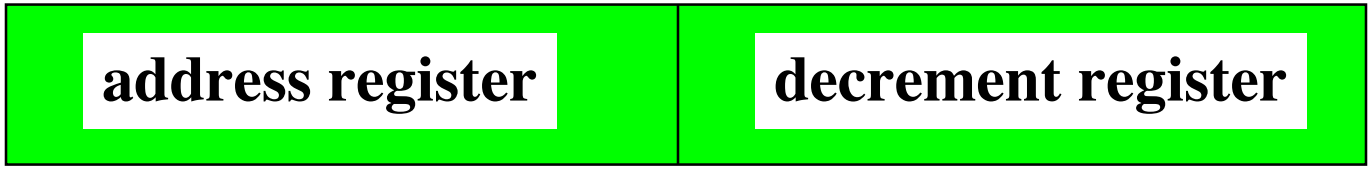
- Recall that arguments are evaluated before the function
- If we wrote – **(car (a b c))**
  - **argument (a b c) would be evaluated before the car**
  - **a would be a function call**
  - **but we literally want the list (a b c) not the result of evaluating a on the arguments b and c.**
- **'(...)** is syntactic sugar for **(quote ...)** where Lisp treats the function **quote** as a special function whose arguments are not evaluated first

(car '(a b c)) ≡ (car (quote (a b c)))

Why the names CAR and CDR?

- Original Lisp developed for an IBM 704 computer which had 18 bit registers
- Pairs of registers could be handled as a single 36 bit 'word'

one word = one lisp cell



CAR ≡ Contents Address Register

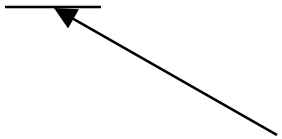
CDR ≡ Contents Decrement Register

Short hand for nested car's and cdr's

Accessing deeper into Lisp structures occurs so frequently that additonal functions are introduced into Lisp.

For example

(cdddar ...) ≡ (cdr (cdr (cdr (car ...))))

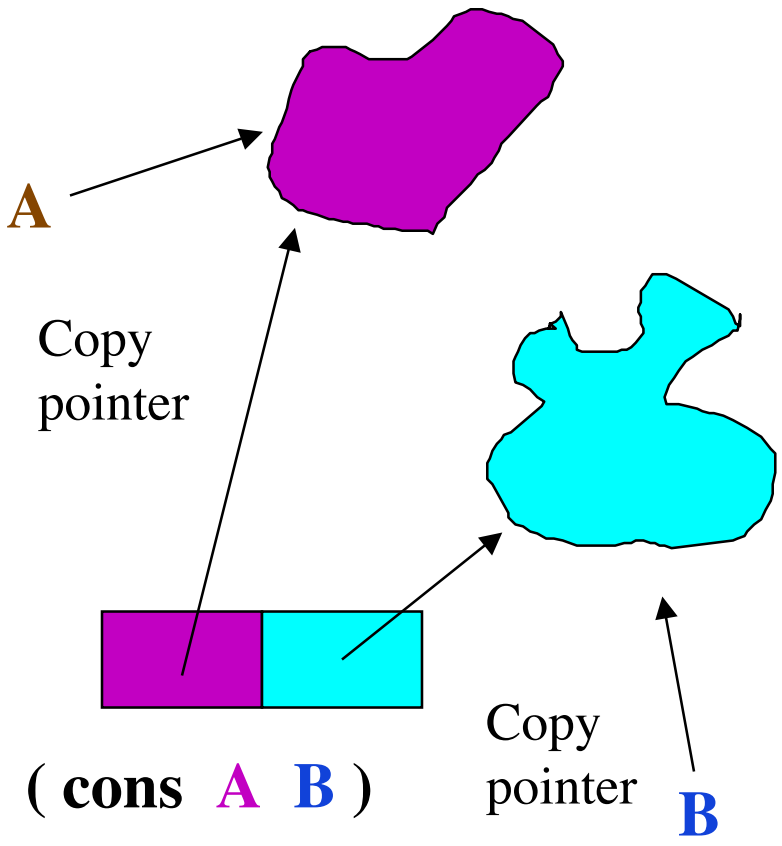


Interpret from right to left

Length depends upon the implementation.

Creating a New Lisp Cell – cons

- Only one constructor function – **cons**
- Copies pointers to the arguments
- Laws
  - » **(car (cons A B)) = A**
  - » **(cdr (cons A B)) = B**



Destructive List Construction

- Cons is expensive as it creates a new cell
  - **memory allocation is invoked**
    - **But it is non destructive – no side effects**

Following is dangerous – do not use in the course!

- For efficiency Common Lisp provides a set of destructive operations – they change lists
  - **(replca cell newValue) & (replcd cell newValue)**
    - **Replace the car and cdr fields of cell with ponters to newValue**
  - **(nconc x y)**
    - **Replace the cdr field of the last component of x with a pointer to y**

SETQ – Define a symbol value

- **(setq x value)**
  - **If the symbol x does not exist it is created**
  - **Symbol x is given the value value**
- In this course **USE ONLY AT THE GLOBAL LEVEL** to create symbols required to test your programs
- Example
  - **(setq x '(1+ 4)) sets the value of x to the list (1+ 4)**
  - **Note the x is not quoted but the second argument is if you do not want to evaluate it.**

Compare SET and SETQ

- **(setq x 'y)**
  - **x has the value y**
- **(set x 'z)**
  - **x still has the value y**
  - **but a new symbol y is created with the value z**
  - **why?**
- See the notes on symbols

DEFUN – define a function

- (**defun** **functionName** (**argumentList**)

  List of S-expressions to evaluate when the function is invoked – usually only one S-expression

)

  - **Example**

  (defun add (a b) (+ a b))

- Value of the function is the value of the last S-expression that is executed

- Functions in Lisp are typically small
  - » **rarely more than 1/2 a page in length**

eq

- (eq x y)
  - Shallow equality
  - True if x and y are the same internal Lisp object
    - It is possible, in some implementations, that a number or character may be represented by different objects
      - As a consequence, (eq 3 3) may be either true or false

eql

- (eql x y)
  - Shallow equality
  - Either eq or the same number or character
    - Note that 3.0 and 3 are different numbers.
    - Implementation may have eql either true or false

equal

- (equal x y)
  - » **Deep equality**
  - » **True if x and y have equivalent values**

- **eq** is stricter than **eql**, which is stricter than **equal**