CS316 1/29/2014.

Lisp ch2-ch6 pg→p111 ch9 is I/O not required

Craft. Appendix A.

practical. is better.

Functional Programming.

Programs in C++ and Java are usually imperative (and not funtional) in style.

Imperative programs specify a sequence of actions that we want the machine to perform. These actions frequently change the values of variables. (ex. x=y*7;) (e.g. by assignment) and often update components of data structures.

C++ and java are often regarded as object-oriented imperative languages.

Funtional programming is different. A functional program does not specify any sequence of actions. Its variables and data structures are immutable. Once a variable. it will keep that value for as long as the variable exists. Once a data structure has been created, its components will never change in value.

A functional program is a collection of functions. However the function in a functional program differ from ~~any~~ many C++/Java functions in the following ways:

① They do not change the ~~variable~~ values of variables and values of data structure components.

② They do not perform I/O.

Function in a functional program return a value without doing anything else.

- They have no side-effects.

To use a functional program, we write an expression that may call ~~many~~ any of the function defined in the program (as well an any available library function) The value of the expression will be output.

```
> (myfunc _ _ ) ↵
   the value of expression
->
```

```
>(+ 3 4)
7
>
```

```
> (load "myfile.lsp") ↵
   load functions defined in file from
```

① Read
② Evaluate
③ ~~Run~~ Print
④ ~~f~~ log

Java : g(h(2,3,y), d(9,w)) f(3,4)

Lisp : (g (h 2 3 y) (d 9 w)) (f 3 4).

Iteration is not used in functional program — iteration would not be useful because re-evaluating the same expression always gives the same value [since variables do not change in value]

Recursion is used a lot.

Conditional expressions (analogous to ? : expression in C++/Java) are used a lot.

Java: c ? e1 : e2

Lisp: (if c e1 e2)

Value is value of e1 if c ⇒ true [T (or anything other than NIL]
value of e2 if c ⇒ false [NIL in Lisp]

suppose value of x is 4.

```
x>4 ? x+9 : x-1
(if (> x 4)
    (+ x 9)
    (- x 1))
```
⇒ 3

To use Lisp on euclid or menus:

After logging in, enter: "CL" at the shell prompt to start the clisp interpreter.

clisp's prompt is:

[n] >

↑ increases by 1 with each iteration of the REPL.

At this prompt, enter any Lisp expression:

[1]> (+ 3 4) ↵

7

[2]> (setq x 4) ↵ [analogous to x=4 in java]

4

[3]> (if (> x 4)
(+ x 9)
(- x 1)) ↵

3 (x, 4) (* x 1)

[4]> (+ x (* x 1))

8 (x, 4) 20

[5]> (+ x (* x 5) 2)

26.

[6]> (sqrt x)

2

[7]> (sqrt x 2)

--- error message "too many arguments"

Break 1 [8]> :q ↵

[9]> (sqrt 2)

1.41 —

[10]> (exit) ← quits from clisp.

CS316 2/3/2016

Lisp: Relevant pages in Sethi 385–403 [see 10.1, 10.2, 10.3]

However, Sethi uses the Scheme dialect of Lisp where as we will use common Lisp.

High-level vs. Assembly Languages. [Further reading see 1.1. in Sethi]

High-Level programming languages (in contrast to assembly and machine languages) are languages that are designed to be indepent of the machines on which programs are to be executed. Programmer are more productive when using High-level languages, because they free programmer from having to constantly think about the inner working of the machine (e.g. registers and machine instructions). They allow programmer to think in terms of more abstract concepts such as variable, lists and function.

Five Advantages of using a High-level Language.

Consider the following high-level language statement (actually, a Java statement) and its equivalent in an assembly languages.

```
x[i+1] = x[i] + total * x[i-1] ;   (*)
```

Advantages of the high-level language statement:

① It is more concise and easier to write correctly. The assembly language equivalent of (*) would consist of many instruction and be easy to get wrong. ["more writeable"]

② It is easier to read and understand, and hence to debug and modify. This also makes it easier for programmers to train themselves by reading existing code. It makes it easier for adapt exsiting code into new projects. ["more readable"]

[note that readability and writeability are two different properties. Some language features may increase one but decrease the other!]

For example, the possibility of doing the same simple thing in a number of good ways makes a language "more writeable" (since a programmer only needs to be familar with one good way to do what she/he wants) but makes it more difficult to understand code written by others (since they may do things in ways that you are unfamilar with). Perl is a well known language that

is often regarded as having this property. Perl programmers call the property

TMTOWTDI "There's More than one way to do it."

$arr[-1], $arr[$#arr], $[@arr-1] are 3 good ways in Perl to get the last element of an array arr. [In Java, we would write arr[arr.lenght-1].

As another example, free formatting of source files (i.e. making tabs and new lines equivalent to spaces, and making arbitrarily long sequences of spaces equivalent to a simple space) makes it easier to write programs but may make programs less readable when they are badly formatted or fomatted in a way that you are not accustomed to.

Most moden languages are free-format languages. Python is well known exception — in python correct indention is required by the language. — indentation and matching deindentation play the same role in python an { and } do in java.

Requiring programmers to declare every variable before it is used and to delare the type of data that will be stored in each variable. (as in Java and C++, but not Lisp, Perl and Python). increase the maount of effort it takes to write a program (especially, a short program) but may make code easier to read and understand.

③ The machine can automatically detect many more kinds of error and inconsistency in the code, either before execution or during execution. Type mismatch, and array index out of range, and misspelt identifier are 3 examples of errors that would be detected by Java the first and third would be detected by the compiler, before the program is executed) ["more reliable"]

④ Most high-level languages come with good standard libraries of predefined function (and other good libraries may be available either commerically or for free)

⑤ High-level language code the uses only standard library function is portable — it can be used on a wide variety of systems with no change or only minor changes.

CS316 2/5/2014.

Lisp.

Advantages of Funtional programming over Imperative programming.

oldest high-level Language

Fortran [First version mid-1950s; Backus, & Coworkers]

2nd oldest

Lisp. [first version Late 1950s; McCarthy & coworkers]

Because variables and data structures are immutable:

① Code is easier to understand and reason about.

② Function are easier to test, because there is no possibility that a function will change the value of a nonlocal variable or a nonlocal data sturcture compent in an inappropriate way. Ease of testing is for the increased, if the codes doesn't use nonlocal variable at all. In that case each function can be tested just by calling it with different argument values and inspecting the values that are returned. Such testing is particulary effective for recursive function, because when a ~~recurise~~ recursive function return an incorrect result for one argument value it will ~~re~~ typically return incorrect results for many larger argument values.

③ Different subexpression can be evaluated in parallel, since evaluation of one subexpression will not ~~interface~~ interfere with evaluation of another. [However, most current implementations of Lisp do not take advantage of this — in fact it would be hard to do so since Lisp does not require programmers to program in a purely functional style.]

Lisp (programming in Lisp) [Further reading Ch2, 3 in WH]

We will be using Common Lisp [Sethi use Scheme, which you ~~can~~ will need to be able to read but will not be expected to write. Clojure is a more recently developed dialect that has attracted much interest in the Lisp community]

symbolic

S-expressions.

Lisp expressions are called S-expression

S-expressions are used ②

CS316

① As Lisp data [the value of any lisp variable (that has a value) is always an S-expression. More generally, the value of any Lisp expression that has a value is an S-expression]

② As Lisp code (Lisp function calls and, more generally, all Lisp expression that can be evaluated, are S-expressions).

Examples:

① 3 → a number (or number atom)

② CAT → a symbol (or symbol atom)

③ (The cat dog) → List of length 3

④ (+ 2 4) → List of length 3

⑤ (Defun FAC (N)
(IF (= N 0)
1
(* N (FAC (- N 1)))))

(1st elem, 2nd elem, 3rd elem, 4th elem)

List of length 4 [note that the 4th element of this list is itself a list of 4 elements]

The 5th example is a Lisp function definition, and is similar to the following Java function definition.

```
c ? e1 : e2        Java
(if c e1 e2)       Lisp
```

```
long fac(int n) {
    return n==0 ? 1 : n * fac(n-1);
}
```

Lisp function definitions have the form

(defun f (a b ... c)
expr)

(a b ... c): formal parameters

expr: expr whose value will be returned. [In normal = nm - functional programming there maybe more than one expr]

S-expression that are meant as Lisp code are called Lisp forms.

Every form is an S-expression. BUT Most S-expression are not possible forms.

(The cat dog) possible form

(5 cat dog) is not a possible form (can't use number as function form).

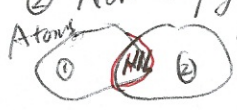
((A B) C D) (can't use a list as function form)

Lisp is a homoiconic language — code has the same syntactic form as data.

S-expression are of 2 kinds

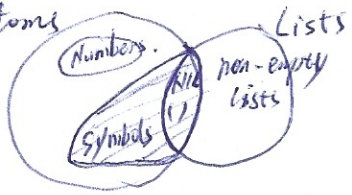
① Atoms < numbers, symbols, other kinds of atom

② Non-empty Lists [The empty list is an atom — it is the constant symbol NIL]



CS316 2/10/2014

S-expression



S-expression
- Non-empty Lists
- Atom
  - Numbers
  - Symbols
  - Other kind of atom (e.g. strings)

(load "sto (load "too (load "solution.lsp")

"solution.lsp" : string.

The constant symbol NIL [=()] represents the empty list.
NIL also represents false.

Lists

S-expression

A list has the form $(e_1, e_2, \cdots e_k)$ $(k \geq 0)$ k is the length of the list.

e.g. ((A) (B 7/4) (1 7 4)) is a list of length 3, each of whose elements is itself a list.

Numbers
- integers (arbitrarily large) } use for exact computation — no rounding error
- ratios (e.g. 17/5) }
- floating point numbers (e.g. 3.4, 4.1e17)
- Complex numbers (e.g. #C(4.1 2.7) = 4.1 + 2.7i)

Symbols

Symbols are used
① As variable names* and names of constants. *e.g. parameter name
e.g. (x, y23, DOG, *, >=, x+1, Z>=4).
x+1 xy1 x-1 x_-1 x+-<1 Apple-Tree (- Apple Tree).
score → grade.
② As names of functions, names of special forms and names of macros.
[A macros is a user-defined special form]
(DEFUN F (x) (+ x 3))
- DEFUN: name of special form
- F: name of a function
- (x): name of a variable
- 3: number
- Symbols

Special forms we have seen so far: DEFUN setf IF.
③ As data. (Any S-expression can be used as data in lisp.)
The value of a lisp variable can be any S-expression)

but always consider about upper case e.g. apple, list read as APPLE
\f \a will be lower case
|apple| will be apple too.

>(F 6) = 9
>(F (F (+ 3 4))) = 13

```
(defun g (x y z)
  (+ (sqrt (+ x y))
     (- y z)))
```

>(g 3 4) = 2

CS316 2/10/2014

Evaluation of Atoms

Symbols can be used as varibles.

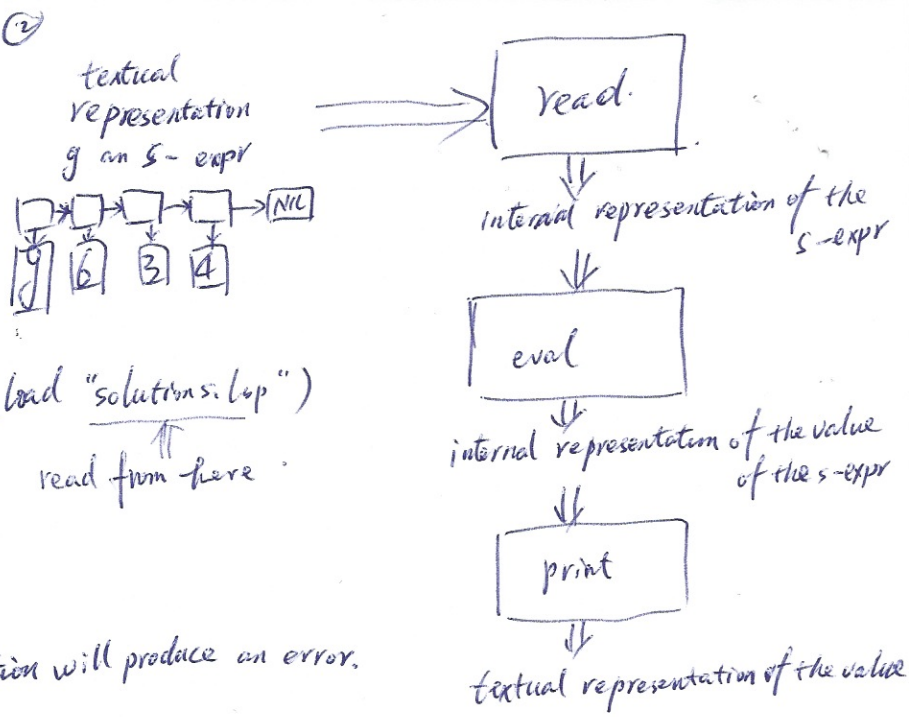
(A symbol evaluates to its current value as a ~~var~~ variable or constant)

```
>173      >17/3     >6/4      >"dog"
173       17/3      3/2.      "dog"
```

~~the value of~~ NIL is NIL.

```
>NIL
NIL
```

If a symbol has no value, then its valuation will produce an error.



Evaluation of Non-Empty Lists

Case 1. If the first element of the list is a symbol that is the name of function, then the value of this list in the result of calling that function. In this case, the values of the 2nd and subsequent elements of the list (if any) will be passes as actual arguements in calling the function.

Case 2. If the first element of the list is a symbol that is the name of a special form or a macro, then the way the first is evaluated depends on the special form or macro.

Case 3. If the first element of the list is a list of lambda-expression form.

eg. (lambda (x y) (+ x y z))

then the value of the list is the result of calling the function defined by the lambda expression.

>((lambda (x y) (+ x y 2)) 4 6) =12.

If none of case 1, 2, and 3 applies, evaluation of the list produces an error.

```
>(3 + 4)        >((F 3))
error           error
```

~~Important But~~ QUOTE special form: (Quote e) → any s-expr. evaluates to e [e is not evaluated]

(QUOTE e) can be written 'e,

```
>'((+ 3 4) (7+3))
((+ 3 4) (7+3))
```

Important Built-in Function

Car or first return the first element of a list

```
>(car '(FAC 7 9))      >(car '(A B (C)) D E)
FAC                    (A B (C))
```

Cdr or rest return the rest of the list after the car is remained

```
>(cdr '(FAC 7 9))      >(cdr '(3 4))      >cdr '(A B (C)) D E)
(7 9)                  (4)                (D E)
```

CS316 2/19/2014 ©

Last time:

car [= first] CDR [= Rest]

built-in functions.

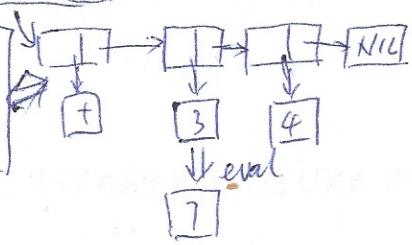
Special form QUOTE (QUOTE <expr>) ≡ '<expr>

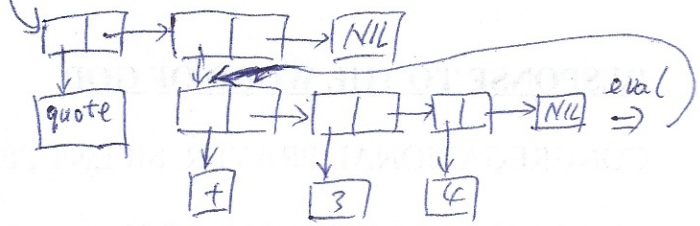
```
> (quote (+ 3 4))
(+ 3 4)
```

← This will not evaluate the expression →

```
> '(+ 3 4)
(+ 3 4)
```

```
> (+ 3 4)
7.
```



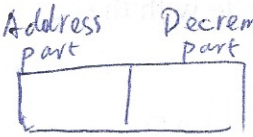



```
> (quote (A (+ 7 (B 9))))
(A (+ 7 (B 9)))
```

```
> '(A (+ 7 (B 9)))
(A (+ 7 (B 9)))
```

```
> (first '(+ D (7 9 (+) K)))
+
> (rest '(+ D (7 9 (+) K)))
(D (7 9 (+) K))
```

```
> (first (rest '(A B C)))
B
```



(second ℓ) ≡ (first (rest ℓ))

(third ℓ) ≡ (first (rest (rest ℓ)))

(fourth ℓ) ≡ (first (rest (rest (rest ℓ))))

(cadr ℓ) ≡ (car (cdr ℓ)) = (first (rest ℓ)) = (second ℓ)

(caddr ℓ) = (first (rest (rest ℓ))) = (third ℓ).

(cadddr ℓ) = (first (rest (rest (rest ℓ)))) = (fourth ℓ).

(caar ℓ) = (first (first ℓ))

(cadadr ℓ) = (first (rest (first (rest ℓ))))
= (second (second ℓ))

Any sequence of up to 4 a's and d's may be used to form a C...R function name.

CONS, LIST, APPEND

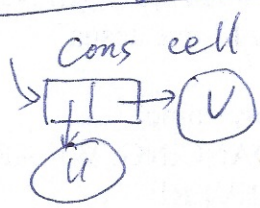
If L is a list of length n and x is any S-expr, then

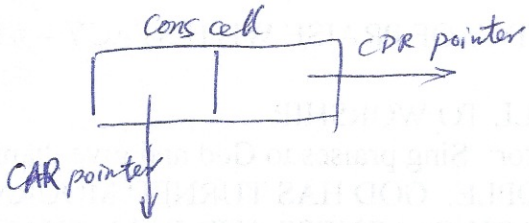
(cons x L) ⇒ a list of length n+1 whose CAR is the S-expr x and whose CDR is the list L.

```
>(CONS 7 '(A B C))
(7 A B C)
>(cons '(A B C) '(1 2))
((A B C) 1 2)
```

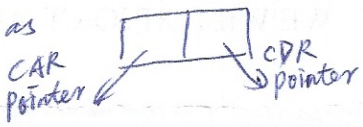
CONS always takes exactly 2 arguments

(cons u v)





often drawn as

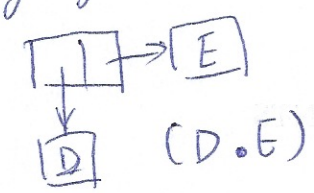


Other names: HEAD = CAR = FIRST
TAIL = CDR = REST.

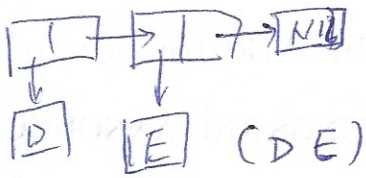
In this course, the 2nd argument of cons should always be a list

If the 2nd argument is not a list, then cons return a dotted list (actually a dotted pair)

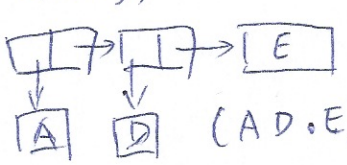
```
>(cons 'D 'E)
(D . E)
  ↑ this dot shows there is a bug in your code.
>(cons '(A B C) 7)
((A B C) . 7)
>(cons 'A '(D . E))
(A D . E)
```



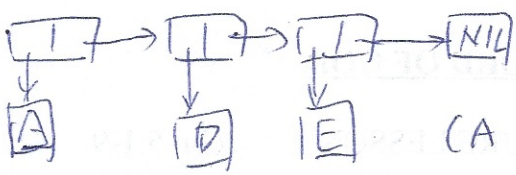
(D . E)



(D E)



(A D . E)



(A D E)

APPEND Append can take any number of arguments. In this course each argument of append should be a list. (otherwise either there will be an evaluation error or append will return a dotted list.)

(append L_1 L_2 ... L_k) returns a ~~lengt~~ list whose length is length(L_1) + length(L_2) + ... + length(L_k).

```
>(append '(1 2 D) '(K))
(1 2 D K)
>(append '(1 2 D) '(E F 7 K) '(P Q))
(1 2 D E F 7 K P Q).
> (cons '(A B) '(D E F))
((A B) D E F)
>(append '(A B) '(D E F))
(A B D E F).
```

```
>(cons 7 nil)
(7)
> (cons '(A B C) nil)
((A B C))
>(append '(A B C) nil)
(A B C)
```

```
>(cons nil '(P Q R S))
(NIL P Q R S).
```

List List can take any number of arguments. each argument can be any S-expr.

(List e_1 e_2 -- e_k) returns a list of ~~leg~~ length k whose elements are the values of the e's.

```
>(List '(1 2 D) '(E F 7) (car '(P Q))
          ⇓        ⇓          ⇓
       (1 2 D)   (E F 7)      P
(1 2 D) (E F 7) P)
```

(List X) = (cons X nil) = (append (List X) L)

```
>(List 7) ~~(List X)~~
(7)
```

if X ≠ a list then (car X) and (cdr X) give evaluation error

```
> (car 7)        > (cdr 'D)
  error            error message.
> (car nil)      >(cdr (NIL)
  NIL              NIL
```

illogical but sometimes convenient.

Some useful Built-in Functions.

```
>(last '(A B C D) 2)
(C D)
>(last '(A B C D)) = (last '(A B C D) 1)
(D)
>(butlast '(A B C D E) 3)
(A B)
>(butlast '(A B C D E))
(A B C D)
>(nthcdr 4 '(A B C D E F))
(E F)
```

~~>(nth L)~~ >(nth K L) = (car (nthcdr K L)) = K+1st element of L.

Ex >(nth 4 '(A B C D E F))
E

(length '(A B C D E)) ⇒ 5

(length '((A B) (C) D E) 7)) ⇒ 3.

(reverse '(A B C D E)) ⇒ (E D C B A).

(expt x y) ⇒ x^y.

(expt 2 0.5) ⇒ 1.414.

CS 316 2/20/2014. ①

~~And~~ Monday's class will be a Lisp class.

> (assoc 'John '((fred 2) (john k) (peter (u v)) (x 19)))

(John k)

> (round 7.3)

7;

0.3. ← second value — ignore this

> (round 7.7)

8;

-0.3

> (round 7.5)

8;

-0.5

[Rule: round to whichever of n and n+1 is even]

> abs (max -9 3.8 4 1.2)

4

> (min -9 3.8 4 1.2)

-9

> (abs 7)

7

> (abs -7)

7.

predicates

A predicate is a function that return a value that means "true" or "false".

False is represented by the symbol NIL, All other S-expression represent "true".

(A B C) is one way to represent "true".

e.g. > (member 'a '(p q a b c))

(A B C)

The constant symbol T is the commonest way to represent "true".

It is unusual to use T for any other purpose [since T is a constant, it cannot be used as a variable (not even as a formal parameter*)]

* (defun f (t) (* t 3)) is wrong

not good.

T and NIL evaluate to themselves, so you don't need to QOUTE them.

> 't

T

> t

T.

Important Built-in Predicates

⇒ means "returns" or "evaluates to"

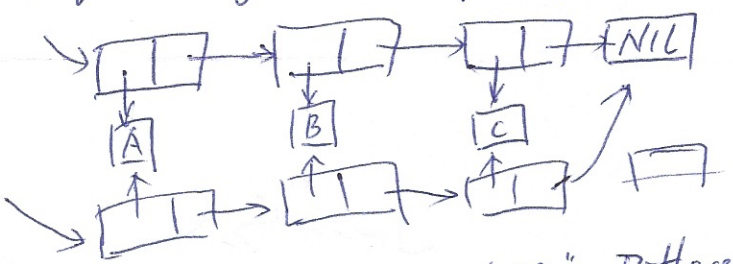
(atom x) ⇒ T if x is an atom
⇒ NIL if x ⇒ a nonempty list.

(equal x y) ⇒ T. if the values of x and y are the same
⇒ NIL. if the values of x and y are not the same.

(equal (+ 1 2) 3) ⇒ T
(equal (+ 1 2) 3.0) ⇒ NIL.
(equal (+ 1.0 2) 3.0) ⇒ T.

(eq '(A B C) '(A B C)) ⇒ NIL eq like == in java, check the pointer.

(eq x y) ⇒ T if x, y evaluate to the same identical object.



Symbols are "memory unique". Different occurrences of the same symbol are represented by the same object.

(eq 'a 'a) ⇒ T
(eq (second '(x a y)) (third '(b a a k))) ⇒ T

Numbers are not always "memory unique".

(eq (expt 2 50) (expt 2 50)) ⇒ NIL in clisp (may return T in other implementation)

(eql x y) ⇒ T. if (eq x y) or if x and y are equal numbers (or equal character)
= NIL for other values of x and y.

You can use (eq x y) if you know at least one of x and y is a symbol.
You can use (eql x y) if you know at least one of x and y is a number or a symbol.

Use equal if it is possible that both arguments are nonempty lists.

[EQUAL is slower than EQ and EQL]

(null x) ⇒ T if x ⇒ ( )
⇒ NIL if x ⇒ any other S-exp.

(not x) ⇒ T if x ⇒ NIL
⇒ F if x ⇒ True.

Since NIL = ( ), and all other S-expression represent true (not x) = (null x)

Use (not x) to test if x is flase or to negate a boolean value x

Use (null x) to test if x is an empty list.

symbolp
numberp
listp
consp
integerp
floatp
realp

} use to test whether the argument is a symbol, number etc.

(integerp 77) ⇒ T

(integerp 77.0) ⇒ NIL

A cons is a nonempty list.

(consp x) ⇒ T just if (listp x) ⇒ T and (null x) ⇒ NIL

(consp x) ≡ (not (atom x))

[NOTE (atom x) correct
(atomp x) incorrect
BAD ↗

predicates on Numbers

The following predicates produce an evaluation error if any argument is not a number.

(= x1 x2 — xk) ⇒ T if the x's have equal numerical values.

⇒ NIL if the x's have numerical values that are not all equal.

if the types of the x's include both rational (incl. integer) and floating pt., then floating pt. values are coerced to rational before the equality test is performeed.

(= 7 'DOG) produces an evaluation error

(= 1.5 3/2) ⇒ T
(= 1.0 1) ⇒ T
(equal 1.0 1) ⇒ NIL.

(= 1/5 0.2) ⇒ NIL because 0.2 can not be represented exactly as a binary floating pt. value. The representation of 0.2 will have some rounding error and when converted to rational it gives a value that is not exactly 1/5.

cs316 2/20/2014 ⑥

Note: /= V (not equal)
i = x
<> x

(/= x y) ≡ (not (= x y).

(zerop x) ≡ (= x 0)

(zerop 'DOG) produces an evaluation error.

(> x1 x2 ... xk) ⇒T if the values of the x's are in strictly decreasing order
=NIL if the values of the x's are not in strictly decreasing order.

similary (< x1 x2 ... xk)
(>= x1 x2 ... xk)
(<= x1 x2 ... xk).

(plusp x) ≡ (> x 0)
(minusp x) ≡ (< x 0)

for all real number x exactly one of
(minusp x)
(zerop x)
(plusp x)
is true.

(evenp n) ⇒T if n ⇒ an even integer.
⇒NIL if n ⇒ an odd integer.
if n ⇒ a value that isn't an integer then we get an evaluation error.

ch4 p59 (up to) Do problems 1-6 on Asn 3.

endp ← test empty

(endp x) ⇒ T if x ⇒ ( )

= NIL if x ⇒ a non-empty list

if x ⇒ an S-expr that is not a list (i.e. an atom other than NIL)

then endp produces an evaluation error.

If x needs to be a list for ~~the~~ subsequent code to work, then it is better to call (endp x) rather than (null x), the if x is not a list, endp will immediately report an error.

[similarly, it is better to call (= x y) than (eql x y) if x and y need to be member for the rest of the code to work.]

Member.

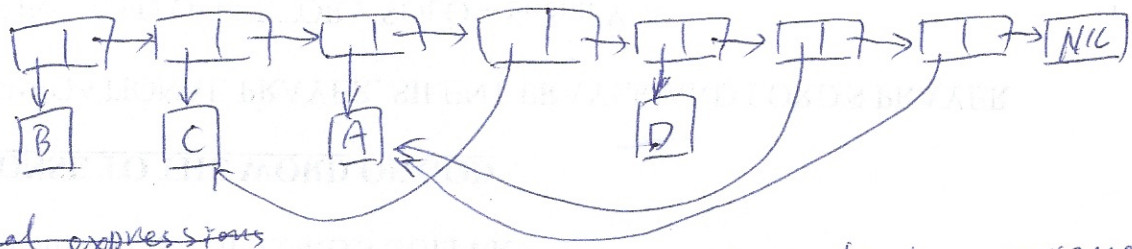
(member 'a '(b c a c d a a)) ⇒ (A C D A A)

(member x L) ⇒ a true* value if the atom x is an element of L.

(x: symbol or number; L: List)

⇒ NIL if x is not an member of L.

* The true value that is returned is not T, in fact, it is the part of L that begins with the first occurrence of x.



~~Conditional expressions~~

Member returns the part of L that begins with the first occurrence of x.

when it returns "true" because:

① It is no more work to return that list than ~~if~~ would be return T.

② The list contain information that maybe useful in some applications.

(e.g. MULTIPLE MEMBER in Asn 3).

CS316 2/24/2014 (2)

Conditional Expression

(if c e_1 e_2) is analogous to c ? e_1 : e_2 in C++/Java.

To evaluate this form, Lisp first evaluates c. Then if the value of c is true (i.e. not NIL), Lisp evaluates e_1 and return its value as the value of the IF. if the value c is NIL then Lisp evaluates e_2 and return its value as the value of the IF.

Note: Only one of e_1 and e_2 will be evaluated!

```
> (+ (if (> 3 2)
         4
         7)
      9)
> 13
```

⇒ (3 > 2 ? 4 : 7) + 9 ⇒ 13

java analog

(if c e) ≡ (if c e nil)

(when c e) ≡ (if c e)

(unless c e) ≡ (if c nil e).

We now use IF to write a function KIND such that (KIND e) will return a list that tells you what kind of S-expression e is. as well as the value of e.

```
> (kind (+ 3 3))
(integer-0 0)
```

```
> (kind (+ 3 4))
(NUM-BUT-NOT-0 7)
```

```
(kind '(+ 3 4))
NON-Empty List (+ 3 4)
```

```
> (kind (car '(a b c)))
(NON-NUMERIC-ATOM. A)
```

```
define kind (e)
  (if (= e 0)
      (integer-0 0)
      (if (equal e 0)
          (integer-0 0)
          (if (memberp e)
              (list 'num-but-not-0 e)
              (if (consp e)
                  (list 'non-empty-list e)
                  (list 'non numeric-atom e))))))
```

missed use member.

(~~mumble~~ (number x L)

use eql to test whether x is equal to any element of L.

(~~equal '(1 2) Y₂ (1 2) 10~~)

(member ~~equal~~ '(1 2) '(a (1 2) b)) ⇒ NIL.

(member x L :test #'equal)

~~Use~~ uses EQUAT instead of EQL.

(member '(1 2) '(a (1 2) b) :Test #'equal) ⇒ ((1 2) B).

COND

(cond (c₁ e₁)
(c₂ e₂)
⋮
(cₖ eₖ))

guard, expression, clauses

≡

(if c₁
e₁
(if c₂
e₂
(if ...
(if cₖ
eₖ) ...)))

if every c is false, then value of the COND is ~~NIL~~.

However ~~it usually~~ it is extremely common for c_k to be the symbol T. In that case the value ~~of~~ of e_k of $c_1, c_2, \ldots c_k$ are all false.

Now we write the above KIND function using ~~cond~~ COND instead of IF:

```
(defun kind (e)
  cond ((eql e 0) '(integer_0 0))
       ((memberp e) '(list 'NUMBER-BUT-> not-0 e))
       ((consp e (list 'non-empty first e))
       (t (list 'non-number-atom e))))
```

AND and OR

(and e₁ e₂ ... eₖ) is analogous to e₁ && e₂ && ... && eₖ in C++/Java except that when the value of the AND is true it's value is the value of e_k (which is true but need not be T)

CS316 2/24/2014 ④

```
>(and (member 'a '(b a c))
     (member 'b '(b a c))).
```

⇒ BAC

(OR e_1 e_2 ... e_k) is analogous to $e_1 || e_2 || \cdots || e_k$ in C++/Java except that when the value of the OR is true its value is the value of the first true e (which need not be T).

```
(or (member 'a '(b a c))
    (member 'b '(b a c)))
```

⇒ (A C)

Like && and || in java/c++, Lisp's AND and OR are special forms that are evaluated using short-circuit evaluation

When evaluating (AND e_1 e_2 ... e_k) Lisp evaluates the e's in left to right order but stops as soon as a false e is found; then it return NIL without evaluating any subsequent e's. if every e is true then the value of e_k is returned.

When evaluation (OR e_1 e_2 ... e_k) Lisp evaluates the e's in left to right order, but stops as soon as it find true e; then the value of that true e is returned without evaluating of any subsequene e's. if every e is false then NIL is returned.

Examples of short-circuit evaluation

```
(defun is_atom_or_has_atomic_car (e)
       (or (atom e) (atom (car e))))
```

> (is_atom_or_has_atomic_car '(A B C)) } this call does not depend on short-circuit evaluation.
T

> (is_atom_or_has_atomic_car '((P) Q R)) } this call does not depend on short-circuit evaluation to work.
NIL ((P) Q R) or (P)

> (is_atom_or_has_atomic_car 'A) } This works only because of short-circuit evaluation!
T

if or were an ordinary function. then evaluation of the 2nd argument [(atom (car e))] would produce an error when (car 'A) is evaluated.

Now consider

```
(defun BAD_is_a_atom_or_has_atomic_car (e)
       (or (atom (car e)) (atom e)))
```

> (BAD_is_a_atom_or_has_atomic_car 'A)
produces an evaluation error when the first argument [(atom (car e))] is evaluated.

Let us now define an ordinary function that is similar to OR:

```
(defun strict-or (a b)
       (or a b))

(defun INCORRECT-is_atom_or_has_atomic_car (e)
       (strict-or (atom e) (atom (car e)))).
```

> (INCORRECT_is_atom_or_has_atomic_car 'A)
gives an evaluation error when the argument (atom (car e)) of strict-or is evaluated

Exercise: Consider the function

```
(defun is_list_with_nonatomic_car (e)
       (and (consp e) (consp (car e))))
```

listp. NIL → list return T.

Give an example of a call of this function that works only because of short-circuit evaluation. Thus the call would not work if we replaced and with strict-and where strict-and is defined by

```
(defun strict-and (a b)
  (and a b))
```

Exercise to test your understanding of consp and car

Would the above definition of is-list-with-non-atomic-car work if we replaced (consp e) with (listp e)? if not, give a counterexample. If it does work, explain why.

LET and LET* (pp 44-6 in WH)

```
(defun f (e e1 w)
  (let ((a (round (sqrt e)))
        (b (* e 2))
        (c (+ e e1)))
    (/ (+ a b c)
       w)))
```

Local variables whose scope is the body of the LET

Let body.

meaning: (/ (+ a b c) w)
where a = (round (sqrt e))
b = (* e 2)
c = (+ e e1)
a, b, c only work in Let scope

```
> (f 3 2 6)
```

e: 3, e1: 2, e1: 6
a = 2
b = 6
c = 5

13/6

bad

```
(defun f (a b w)
  (let ((a (round (sqrt a)))
        (b (* a 2))
        (c (+ a b)))
    (/ (+ a b c)
       w)))
```

This is another way to write the same function. *much less readable!

```
(let ((x1 expr1)
      (x2 expr2)
      ⋮
      (xk exprk))
  body expr)
```

evaluated as follows:

① Evaluate $expr_1$ $expr_2$ ··· $expr_k$ in the environment that exists outside the LET.

② Evaluate body-expr. in an environment in which each local variable x_i is bound to the value of the corresponding expression $expr_i$ [Binding of other variables are the same as they were outside the LET]

The above LET is roughly equivalent to

```
(defun fund(x1 x2 ... xk)
    body-expr).

(func expr1 expr2 ... exprk).
```

```
(defun f (a b w)
  (defun func (a b c)
    (/ (+ a b c)
       w))
  (func (round (sqrt a))
        (* a 2)
        (+ a b)))
```

LET* is similar to LET except that each $expr_i$ is evaluated in an environment that includes binding of $x_1, \ldots x_{i-1}$ to the value of $expr_1 \ldots expr_{i-1}$

```
(defun f (a b w)
  (let* ((a (round (sqrt a)))
         (b (* a 2))
         (c (+ a b)))
    (/ (+ a b c)
       w)))
```

```
(let* ((x1 e1)
       (x2 e2)
       ( ⋮ )
       (xk ek))
  body-expr)
```

≡

```
(let ((x1 e1))
  (let ((x2 e2))
    ⋮
    (let ((xk ek))
      body-expr) ...)).
```

CS316 3/3/2014.

Try to do the 7 problems in sec 1. of Asn 4 before Wednesdays class.

Common uses of Let and Let*

① To give meaningful names to certain expression, as in

(Let ((birth-year (eighth L)))
  ... birth-year ...)

} much clearer than

... | ... (eighth L) ... |

② To use the value of a computationally expensive expression two or more times

(Let ((x expensive-expr))
  | x x x |)

② is often used to avoid making the same recursive call two or more times

$f(n) = f(n-1) + \sqrt{f(n-1)}$ Number of recursive calls grows exponentially with depth of recursion (and hence with argument size).

Better

Let* $x = f(n-1)$

$f(n) = x + \sqrt{x}$.

(Let ((x (f (- n 1))))
  (+ x (sqrt x))).

number of recursive calls = depth of recursion = argument size.

Example to help you with problem A-G and 1-7.

Write a function FACTORIAL such that (FACTORIAL n) => $n \times (n-1) \times \cdots \times 2 \times 1$.

[note: $0! = 1$]

step 1 Assume FACTORIAL has already been written correctly. Then we can write a function My-FACTORIAL which returns the same value as FACTORIAL where its argument > 0. in the following way.

(defun my-factorial (n)
  (let ((x (factorial (- n 1))))
    (* n x)))

step 2A. Now we can improve my-factrical to make it work correctly even when the argument is 0; (defun better-myfactrical (n)
  (if (zerop n)
    1
    (let ((x (factorical (- n 1))))
      (* n x)))).

Step 2B.

Now we can modify the definition of Better-my-factorial to give a definition of FACTORIAL itself.

To do this, it is enough to change the name of the function to Factorial: There is no need to change the definition (except for stylistic reasons).

Final definition

```
(defun factorial (n)
    (if (zerop n)
        1
        (let ((x (factorial (- n 1))))
            (* n x)))).
```

Note if you can do one of A-G but still have difficulty with the corresponding problem in 1-7, then you have understood the method! (Or maybe your solution to problem A-G was actually wrong).

Step 2C

We see that X is never used more than once, so we can eliminate the LET.

⇒

```
(defun factorial (n)
    (if (zerop n)
        1
        (* n (factorial (- n 1)))))
```

Assembly Language Programing vs High Level Language (contd.)

Previously, we considered the advantages of using a high-level language.

- greater readability
- greater "writeability"
- more errors can be automatically detected.
- more portable.
- availability of good libraries of useful functions.

The main disadvantages of using a high-level language is loss of efficiency: High-level language code will usually be slower and/or use more memory than equivalent Assembly language code that has been carefully written by a programmer who is skilled in the use of that assembly language.

However, in recent year it has become quite hard/time-consuming even for skilled assembly language programmers to write code that is more efficient than the code generated by a good complier. An important reason in that modern CPUs will ~~simulater~~ simultaneously dispatch two or more machine instructions to different internal functional units (for parallel execution) when they detect that this is possible.

Thus
```
copy  A0 TO A1   [A0 = A1]
ADD   A1 TO A2   [A2 += A1]
ADD   A2 TO A3   [A3 += A2]
```

~~Many make language bad~~ may take longer to execute an a cpu than the equivalent instruction.

- copy A0 to A1
- ADD A2 TO A3

} because the first 2 instructions can be executed in parallel as can the last two.

- ADD A1 to A2
- ADD A0 to A3

it is hard (and error-prove) work for a human programmer to take full advantage of ~~them~~ this.

Ex: Write a function LIST_LENGTH such that if L is any List then (LIST-LENGTH L) ⇒ the no. of elements in L [in fact there is a built-in function LENGTH that does this.]

Step 1. Assume LIST-LENGTH has already been written. Complete the following function My-List-LENGTH so that if L is any nonempty list then (MY-LIST-LENGTH L) ⇒ the same value as (LIST-LENGTH L) (i.e. the no. of elements in L).

```
(defun my-list-length (L)
    (let ((x (LIST-LENGTH (cdr L))))
        (+ x 1)))
```

[Note (+ x 1) does not work of L=NIL. L=NIL x=0. We would want the value of the let to be = 0.].

Step 2A Add a case so the function works even if L=NIL. call the function Better-my-list-length.

```
(defun better-my-list-length (L)
    (if (endp L)
        0
        (let ((x (List-length (cdr L))))
            (+ x 1))))
```

Step 2B the above depends on the assuptin that List-length has already been written correctly. However we can just rename Better-my-list-length to List length.

```
(defun List-Length (L)
    (if (endp L)
        0
        (let ((x (list-length (cdr L))))
            (+ x 1))))
```

null and endp ? ↑ 양자성 List.

Since x is never used more than once, we can eliminate the LET.

```
(defun List-length (L)
    (if (endp L)
        0
        (+ (list-length (cdr L)) 1)))
```

General Pattern

A recursive function of the argument which is assumed to be a list or a non-negative integer can often be written as follows:

e is a list →

```
(defun f (e)
  (if (endp e)
      ⊕
      (let ((x (f (cdr e))))
        ⊗)))
```

e is a non-negative int.

```
(defun f (e)
  (if (zerop e)
      ⊕
      (let ((x (f (- e 1))))
        ⊗)))
```

⊕ is the base-case expression, the value of this should be (f 1) or (f 0)

⊗ This is the general case expression. This expression must be an expression that computes (f e) from X and (possibly) e.

Notes:

1. The ⊗ expressions may have more than one case — it maybe, e.g. an if or a COND expression.
2. If there is no case which X is used more than once, then it is good style to eliminate the LET (as we did in the FACTORIAL and List-length examples).
3. If the LET is not eliminated and there are any cases in which X does not need to be used, then move such cases outside the LET to avoid making an unnecessary recursive call in the cases.

How can we write the ⊗ Expression?

If we can't see an appropriate ⊗ then try some examples — look at various possible argument values e and in each case find a ⊗-expression that would give the correct value in that case

For each ⊗ expression, think of sufficient conditions (which may depend on e and X) for that ⊗-expression.. would give the right result.

When you have a set of ⊗-expressions whose suffient conditions cover all possibilities, then you are ready to write the final ⊗-expr.

Ex Write a function

(evens L) such that if L is a list of integers then (evens L)

⇒ a list of all the even elements of L. (that can be obtained by omitting all the odd elements). Ex: (evens '(1 2 3 4)) ⇒ (2 4) (evens '()) ⇒ NIL

```
(defun evens (L)
  (if (endp L)
      NIL
      (let ((X (evens (cdr L))))
        (if (evenp (car L)) (cons (car L) X) X))))
```

Since X is never used more than once, we can eliminate the LET! — exercise.

The ⊛-expr may require a helping function.

for function of 2 or more arguments, decide which argument to make smaller in the recursive call. (In simple problems the other arguments can be left unchanged in the recursive call.)

CS316 3/10/2014. ①

Exam1 March 31 (Monday).

Ex: (of Lisp function with 2 args): write a function List-append such that (List-append L1 L2) returns the same list as (append L1 L2). for all lists L1 and L2.

(List-append '(1 2 3) '(4 5 6 7)) ⇒ (1 2 3 4 5 6 7).

```
(defun list-append (L1 L2)
    (if (endp L1)
        L2.
        (let ((X (list-append (cdr L1) L2)))
          ⊛ (cons (car L1) X))))
```

Guess that L1 is the argument we should make smaller in the recursive call

expr that computes list-append L1 L2. from X and possibly L1 L2

exercise eliminate the LET

Note: The ⊛ expression may have more than one case, and may involve calls of helping functions — e.g. SET-REMOVE can be used in the ⊛-expr for singletons. Since X is only used once,

Exercise: Verify that if you had guessed that L2 should be made smaller in the recursive call, then there would be no good way to write the ⊛ expression.

Assembly vs high-level languages contd

Nowadays, assembly is not used much. some situation where is still used (sometimes) are:

a) To program microcontrollers — inexpensive single-chip computer that have very little memory. These are used e.g. to control microwave ovens or a car's antilock braking system.

b) To write short code fragments that need to be as fast as possible — e.g. in an inner loop in a game program.

c) To write code that needs to interact with the operating system, the cpu or other hardware in ways that are not adequately supported by library function that are available for you to call from your high-level language.

For b) and c), we can either link assembly language by a high-level language compiler, or use a high-level language compiler that supports inline assembly.

CS316 3/6/2014 (2)

programming paradigms

A programming paradigms is a style of programming that is based on a particular way of thinking about what a "program" is.

4 long-established paradigms.

① Imperative programming
1a) Object-oriented programming | imperative

② Functional programming

③ Logic programming

1. Imperative programming.

Programs are regarded as specification of sequens of actions that the machine is carry out. The action may (and frequently will) chang values of variables and update values stored in data structives.

This has always been the most widly used program paradigm. Assembly Language programming is imperative.

Important languages that were designed solely or mainly for use in imperative programing machine.

Fortran: [- first high-level language
- developed in the mid-1950s by Backus and his team at IBM.
- designed for scientific computing (i.e. doing mathematical calculations for engineers and scientists.)
- dominated scientific computing from the early 1960's through the 1980s]

Algol 60. [designed for scientific computing in the late 50s by leading computing scientists (e.g. perlis, Samelson). introduced many important ideas (e.g. control structures, compulsory variable declarations) but never was as widely used as it was admired]

Cobol [designed for commercial data processing
- e.g. management payrolls, account, inventory, employee records.

For many years this was by far the most widely used language.
(mid 60s – early 90's).

Basic [designed for teaching programming to non-scientists in the mid 1960s, widly used for programming microcomputer in the 1970s and 1980s].

pascal [designed to teach computer science students to write (well structured imperative) programs.
– very widely used for this from early 1980s to mid 1990s]

Imperative languages that are widely used today include C and perl.

16) Object-oriented imperative programming.

This means imperative programs that make extensive use of objects

[imperative programming that does not use objects is sometimes referred to as procedural programming. The above-mentioned languages were designed for procedural programming.
in procedural programming function/procedures that work with passive data structures.
Important concepts of object-oriented programming are: ① polymorphic calls of methods ② encapsulation – hiding components behind a public interface ③ inheritance.

So far, the recursive function we have been writing have had the form.

```
(defun f (P1 P2 ... Pn)
  (if (base-case P1)
      <expr that computes (f P1 ... Pn) from X and possibly P1 ... Pn>
      (let ((X (f (smaller P1) P2 ... Pn)))
        ...)))
```

where

base-case = zerop
(smaller P_1) = (- P_1 1)

or

base-case = endp
(smaller P_1) = (cdr P_1)

[The above assumes P_1 is the parameter we choose to make smaller in the recursive call. It is possible that we should make one of the other's smaller instead.]

More sophisticated Recursive

Sometimes it is appropiate to make an argument smaller in a different way then by subtracting 1 or taking its CDR.

Other common ways to make an argument e smaller are:

(cddr e) = (rest (rest e))

(floor e 2) $= \lfloor \frac{e}{2} \rfloor$ for any non-negative integer e.

[another way to write SPLIT-LIST]

[another way to write SPLIT-NUMS]

a^n downgrade $(a^n \bmod m)$.

(cdr e1) where e1 is a suitable transformed version of e.

Ex Let's write SPLIT-LIST using cddr to make the argument smaller.

Recall (split-list '(A X 7 K 9 3)) ⇒ ((A 7 9) (X K 3)).

```
(defun split-list (L)
  (if (endp L)
      '(() ())
      (let ((X (split-list (cddr L))))
        (if (endp (cdr L))
            (list L ())
            (list (cons (car L) (car X))
                  (cons (cadr L) (cadr X)))))))
```

To write the ⊛-expr, let's try some examples

L = (A X 7 K 9 3)

X = ((7 9) (K 3))

Want the ⊛ expr to ⇒ ((A 7 9) (X K 3))

clearly, this case (list (cons (car L) (car X))
(cons (cadr L) (cadr X)))

is a good ⊛-expr. We see that this ⊛-expr actually works whenever L has 2 or more elements.

But, if L has length 1, what happens?

L = (A)
X = (() ())

Here the above ⊛ expr gives ((A) (NIL)) which is wrong — the expr needs to give ((A) ())

In this case, we can just use the ⊛-expr (list L ())

(endp (cdr L)) tests whether L has ≤ 1 element

X is used twice in the second case, so we should NOT eliminate the LET.

But X is not used in the 1st case, so we should move that case outside the LET.

Final version:

```
(defun split-list (L)
  ~~(if (endp L)~~ } not actually needed we see!
      ~~(list () ())~~
    (if (endp (cdr L))
        (list L ())
      (let ((X (split-list (cddr L))))
        (list (cons (car L) (car X))
              (cons (cadr L) (cadr X)))))))
```

[Exercise: Explain why not!]

Example of (floor e 2)

Write a function (power a n) which returns (expt a n) whenever n is a positive integer and a is a number.

slow way $a^n = a * a^{n-1}$ (slow if n is very large).

Fast way $a^n = \left(a^{\lfloor \frac{n}{2} \rfloor}\right)^2$ if n is even. for $n \geq 2$

$= \left(a^{\lfloor \frac{n}{2} \rfloor}\right)^2 a$ if n is odd

```
(defun power (a n)
  (if (= n 1)
      a
      (let ((x (power a (floor n 2))))
        (if (evenp n)
            (* x x)
            (* x x a)))))
```

Another idea:

change more than one argument in the recursive call.

$a^n = (a^2)^{\frac{n}{2}}$ if n is even.

$= (a^2)^{\lfloor \frac{n}{2} \rfloor} a$ if n is odd.

```
(defun power_b (a n)
  (if (= n 1)
      a
      (if (evenp n)
          (power_b (* a a) (/ n 2))
          (* a (power_b (* a a) (floor n 2))))))
```

Note Modular exponentiation

(computes $m^n \bmod k$) [m, n, k integers]

can be done in a similar way. modular exponentiation with large exponents is useful in cryptography.]