CSC 530 Lecture Notes Week 2

Discussion of Assignment 1

Topics from the Lisp Primer

Topics from Part 1 of the Readings

I. Reading this week -- papers 1-4 on functional programming.

II. Discussion of Assignment 1

A. What are you doing here?

B. The *read-xeval-print-loop*

Assignment 1, cont’d

```
(defun read-xeval-print-loop ()
  (prog (alist result)
    (setq alist ’(nil))
    loop
        (princ "X>")
        (setq result (xeval (read) alist))
        (princ (car result))
        (setq alist (cadr result))
        (terpri)(terpri)
        (go loop)
  )
)
```

C. The meat of the matter is the *`alist`*.

III. "Naive" alist layout

A. A list of *bindings*.

B. General form

```
( name value )
```

C. For Lisp, two categories:

Alist layout, cont’d

1. variable binding

```
( var-name data-value )
```

2. Function binding is a triple of the form

```
( function-name
        formal-parms
        function-body )
```

Alist layout, cont’d

D. Distinguish variable versus function bindings by lengths.

E. In Assignment 1, bindings created and modified in three ways:

1. Variable bindings by `(xsetq x v)`
2. Function bindings by `(xdefun f parms body)`
3. Function call bindings by (f a_1 ... a_n)

Assignment 1, cont'd

F. Addition, removal, and search done LIFO.

G. What is *naive* about the organization -- does not accurately represent scoping rules of Common Lisp.

H. The bottom line for Assignment 1

  1. same evaluation results as mine

  2. may differ in alist dump

On to the Lisp Primer

IV. Selected primer topics

A. Let’s have a look

B. Complete details in Lisp ref man

1. Overview

Compared to C, the major diffs are

- syntax
- interpretive environment
- lack of explicit type declarations

Overview, cont’d

Major similarities include:

- Program structure and scoping.
- Function invocation, conditionals
- Underlying similarity in data structures

2. An Introductory Session

```
% ~/classes/530/bin/gcl
GCL (GNU Common Lisp) ...

>(+ 2 2)
4

>(defun TwoPlusTwo () (+ 2 2))
TWOPLUSTWO

>(defun TwoPlusXPlusY (x y) (+ 2 x y))
TWOPLUSXPLUSY

>(TwoPlusXPlusY 10 20)
32

>(load "avg.l")
Loading avg.l
Finished loading avg.l
T
```

```
>(avg ’(1 2 3 4 5))
3

>(avg ’(a b c))

Error: C is not of type NUMBER.
Fast links are on: ...
Error signalled by +.
Broken at +.  Type :H for Help.
>>:q

Top level.
>(help)

GCL (GNU Common Lisp)


>(bye)
Bye.
```

3. Lexical and Syntactic Structure

- Very simple -- atoms and lists.
- Atoms are:
  - identifier
  - integer or real
  - double-quoted string
  - constants t and nil
- A list is zero or more elements in matching parentheses

3.1. Expr and Function Call Syntax

Lisp	**C**
(+ a b)	a + b
(f 10 20)	f(10, 20)
(< (+ a b) (- c d))	(a + b) < (c - d)

- General format:

$$(\textit{function-name}\ \textit{arg}_1\ \ldots\ \textit{arg}_n)$$

Function Call Syntax, cont’d

- Function call evaluated as:

  1. *function-name* checked.
  2. Each arg_i evaluated.
  3. Each arg_i bound *call-by-value* discipline
  4. Body of function evaluated

- Quite similar to C

3.2. The Quote Function

- Consider

```
>(defun f (x) ... )

> (defun g (x) ...)
```

- Given these, consider

```
(f (g 10))
```

- Alternatively, consider

```
(f ’(g 10))
```

3.3. No `main` Function Necessary

- Simply `defun` and `load`

- Any defined function can be called.

4. Arithmetic, Logical, Conditional Expressions

`(+` *numbers*`)`

`(1+` *number*`)`

`(-` *numbers*`)`

`(1-` *number*`)`

`(*` *numbers*`)`

`(/` *numbers*`)`

See ref man for others.

4.1. Type Predicates

`(atom` *expr*`)`

`(listp` *expr*`)`

`(null` *expr*`)`

`(numberp` *expr*`)`

`(stringp` *expr*`)`

`(functionp` *expr*`)`

See ref man for others.

4.2. The `cond` Conditional

(cond (($\textit{test-expr}_1$) $\textit{expr}_1$ ... $\textit{expr}_j$)
. . .
(($\textit{test-expr}_n$) $\textit{expr}_1$ ... $\textit{expr}_k$)

4.3. Equality Functions

`=`	numeric
`string=`	string
`equal`	general expr
`eq`	same-object

5. Function Definitions

Lisp:

```
(defun f (x y)
    (plus x y)
)
```

C:

```
int f(int x,y) {
    return x + y
}
```

6. Lists and List Operations

- List is *the* basic data structure.

- Lisp does support others -- but who cares.

6.1. Three Basic List Ops

Operation	Meaning
car	first element
cdr	everything except first element
cons	construct a new list (essentially)

The tail recursion idiom

```
(defun PrintListElems (l)
    (cond ( (not (null l))
            (print (car l))
                (PrintListElems(cdr l))
          )
    )
)
```

Comparable to in C:

```
void PrintArrayElems(int a[], int n) {
    for (i=0; i<n; i++)
        printf("%d0, a[i]);
```

6.2. cXr forms

- General form:

  `c`*X*`r`

  where the *X* can be replaced by two, three, or four *a*'s and/or *d*'s

- E.g., `(cadr L)`

6.3. Other Useful List Ops

`(append` *lists*`)`

`(list` *elements*`)`

`(member` *element list*`)`

`(length` *list*`)`

`(reverse` *list*`)`

`(nth` *n list*`)`

`(nthcdr` *n list*`)`

`(assoc` *key alist*`)`

`(sort` *list*`)`

6.4. Dot Notation

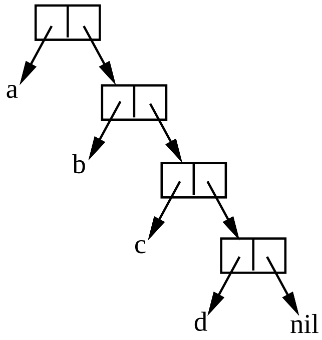


Figure 1: Internal representation of the list`(a b c d)`.

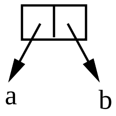


Figure 2: Internal representatino of`(cons 'a 'b)`.

7. Building C-Like Data Structures

7.1. Arrays

- Trivially represented as lists

- Implementation of array indexing

```
(defun my-nth (n l)
    (cond ( (< n 0) nil )
          ( (eq n 0) (car l) )
          ( t (my-nth (- n 1)
                      (cdr l)) )
    )
)
```

7.2. Structs

- General format:

$$((\textit{field-name}_1\ \textit{value}_1) \ldots$$
$$(\textit{field-name}_1\ \textit{value}_1))$$

- Functions to access and modify

```
(defun getfield (field-name struct)
  (cond
    ( (eq struct nil)  nil )
    ( (eq field-name (caar struct))
      (car struct) )
    ( t (getfield field-name
               (cdr struct)) )
  )
)
```

Structs, contd’

```
(defun setfield
        (field-name value struct)

  (cond ( (null struct) nil )

        ( (eq field-name
              (caar struct) )
          (cons (cons (caar struct)
                      (list value))
                (cdr struct)) )

        ( t (cons (car struct)
                  (setfield field-name
                    value (cdr struct))) )
  )
)
```

7.3. Linked Lists and Trees

7.3.1. Linked Lists

- Just plain lists in Lisp

- At underlying dot-notation level, Lisp lists are implemented using pointers

7.3.2. N-Ary Trees

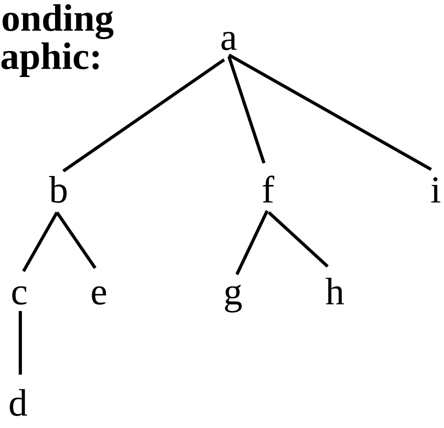
- General form

  (root $subtree_1$ ... $subtree_n$ *)*

- For example

Lisp: `(a (b (c d) e) (f g h) i)`

Corresponding Graphic:



8. A Multi-Function Example

- Merge sort

- Illustrates typical Lisp style

9. Basic Input and Output

`(read [`*stream*`])`

`(print` *expr* `[`*stream*`])`

`(prin1` *expr* `[`*stream*`])`

`(princ` *expr* `[`*stream*`])`

`(terpri [`*stream*`])`

`(open` *UNIX-filename*`)`

10. Programs as Data

- Lists and function calls are syntactically identical

- Programs and data can be manipulated interchangeably

- Any expr can be treated equally well as program or data

10.1. Eval

- Callable `eval` same as in read-eval-print loop

- Any legal Lisp expression can be executed

10.2. Apply

- "Junior" function slightly less powerful than `eval`

- E.g.,

```
(apply '+ '(2 2))
```

produces 4.

11. Scoping with Let

Lisp: **C:**

```
(let                {
    ( i                 int i;
      (j 10)            int j = 10;
      (k 20) )          int k = 20;

    expr1               stmt1
       ...                 ...
    exprn               stmtn
)                   }
```

- Also `let*`

12. Imperative Features

- Features thus far comprise the *functional* subset.

- Imperative features of Lisp make it more "C-like".

12.1. Assignment Statements

- `setq` is Lisp assignment
- E.g.,

```
(setq x (+ 2 2))
```

- More general form is `setf`
- E.g.,

```
(setf (cadr x) 10)
```

12.2. Scope and Binding

- No explicit var decls in Lisp.
- Site of binding defines scope.
  - *o* Top-level binding means global
  - *o* Local binding (i.e., inside defun) means local
  - *o* Function parms are local
    - In Lisp *free* means not bound in the current scope.

12.3. Prog

$$\begin{gathered}(\mathrm{prog}\ ((\mathit{var}_1\ \mathit{val}_1))\ldots\\ (\mathit{var}_n\ \mathit{val}_n)\\ \mathit{expr}_1\ \ldots\ \mathit{expr}_k)\end{gathered}$$

Lisp:

```
(prog
    ((i 10)
     (j 20.5)
     (k "xyz"))

  (setq i (+ i 1))
  (setq j (1+ j))
  (print (+ i j))
)
```

C:

```
{
    int i = 10;
    float j = 20.5;
    char* k = "xyz"

    i = i + 1;
    j += 1;
    printf("%d0, i + j);
}
```

Prog, cont’d

- `return` returns from `prog`

- Don’t confuse Lisp’s `return` with C’s

- `go` is a standard goto, e.g.

```
(defun read-eval-print-loop ()
    (prog ()
        loop
            (princ ">")
            (print (eval (read)))
            (terpri)
            (go loop)
```

)

)

12.4. Iterative Control

- General form:

  (do ((var_1 val_1 rep_1) ...
  (var_n val_n rep_n)) *exit-clause*
  $expr_1$ ... $expr_k$)

- *exit-clause* form:

  ([*test* [$test\text{-}expr_1$ ... $test\text{-}expr_m$]])

- Similar to C for loop, except test is

an *until*.

12.5. Destructive List Operations

`(rplaca` *cons-cell expr*`)`

`(rplacd` *cons-cell expr*`)`

`(nconc` *lists*`)`

`(setf` *cons-cell expr*`)`

Destructive, cont’d

```
>(setq x-safe ’(a b c))
(A B C)

>(setq y-safe x-safe)
(A B C)

>(setq x-safe
    (cons ’x (cdr x-safe)))
(X B C)

>y-safe
(A B C)
```

Destructive, cont’d

```
>(setq x-unsafe ’(a b c))
(A B C)

>(setq y-unsafe x-unsafe)
(A B C)

>(rplaca x-unsafe ’x)
(X B C)

>y-unsafe
(X B C)
```

Destructive, cont’d

```
>(setf (cadr y-unsafe) ’y)
Y

>x-unsafe
(X Y C)

>y-unsafe
(X Y C)
```

12.6. Call-by-Reference

- Destructive list ops make it possible

- E.g.,

```
(defun dsetfield
        (field-name value struct)
    (setf
      (cdr (assoc field-name struct))
      (list value))
)
```

12.7. Pointers Fully Revealed

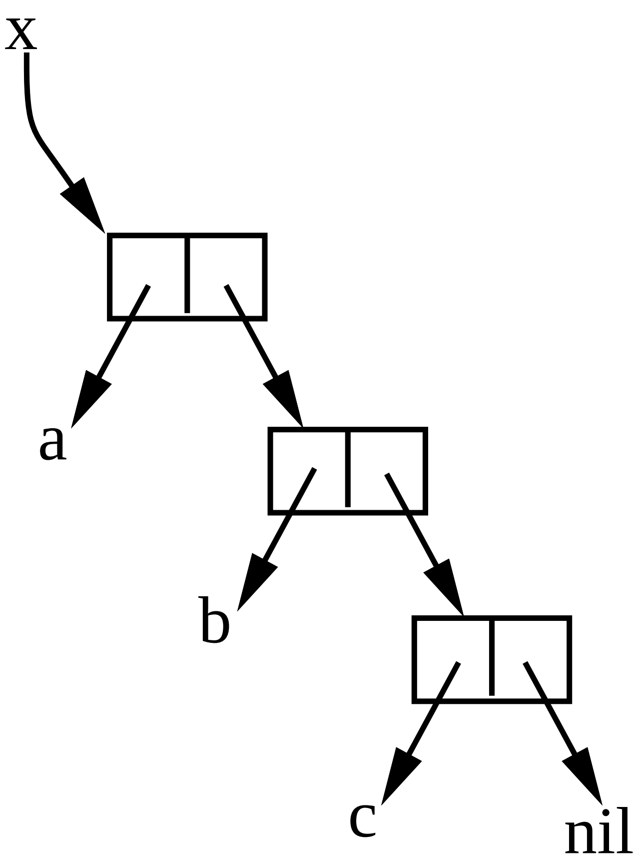
- Consider:

```
>(setq x ’(a b c))
(A B C)

>(defun f (i j k)
    (setf (cdddr k) (cdr j))
    (setf (cadr j) i)
    (setf (cdr j) k) --primer typo--
    (setq z k)
    nil
)
F

>(f x (cdr x) ’(a b c))
NIL
```

Pointers Revealed, cont’d

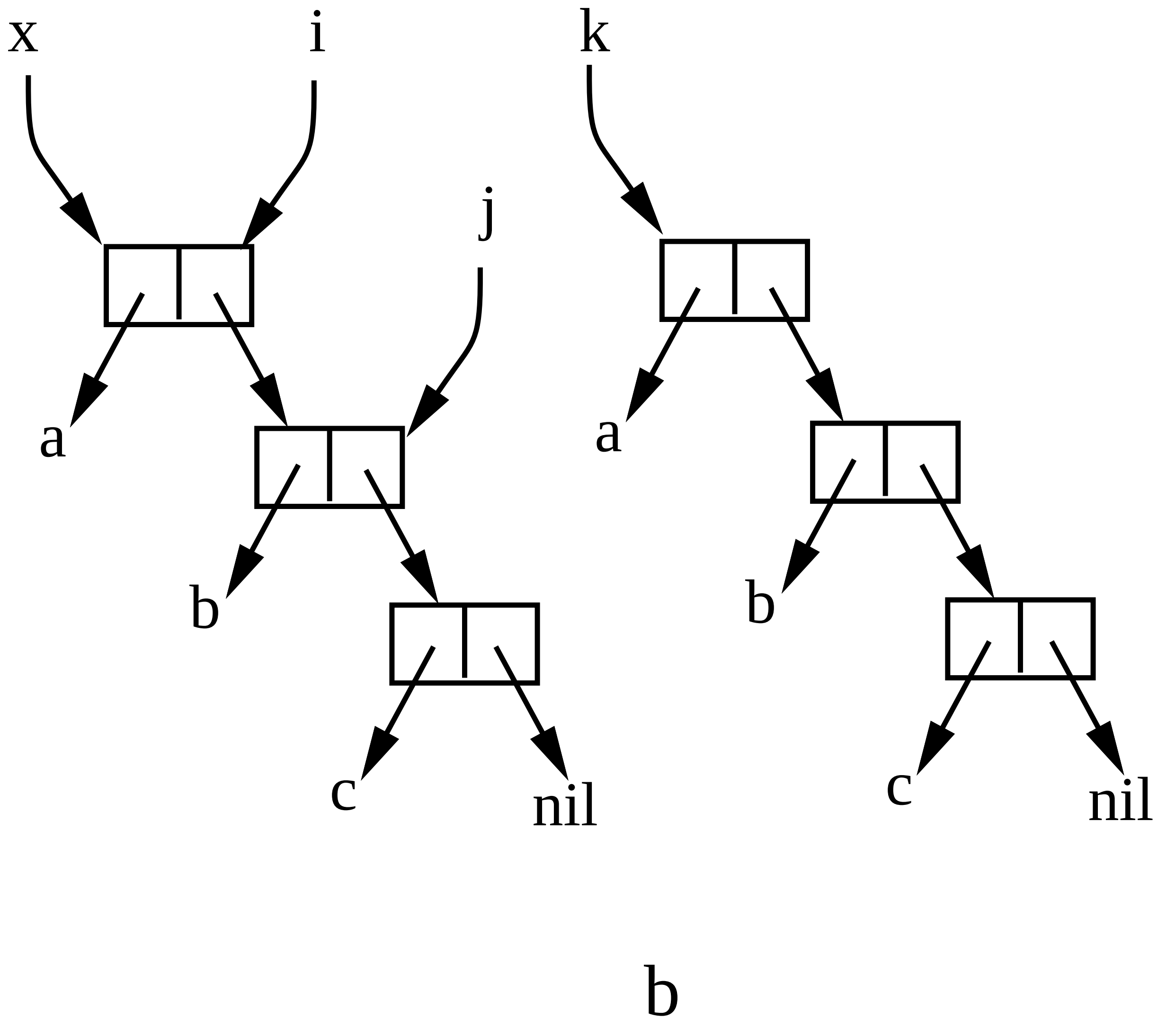
- Three snapshots:
  a. after assignment to x, before f called
  b. after f called, parameters bound, before body executed
  c. after execution of f

Pointers Revealed, cont’d

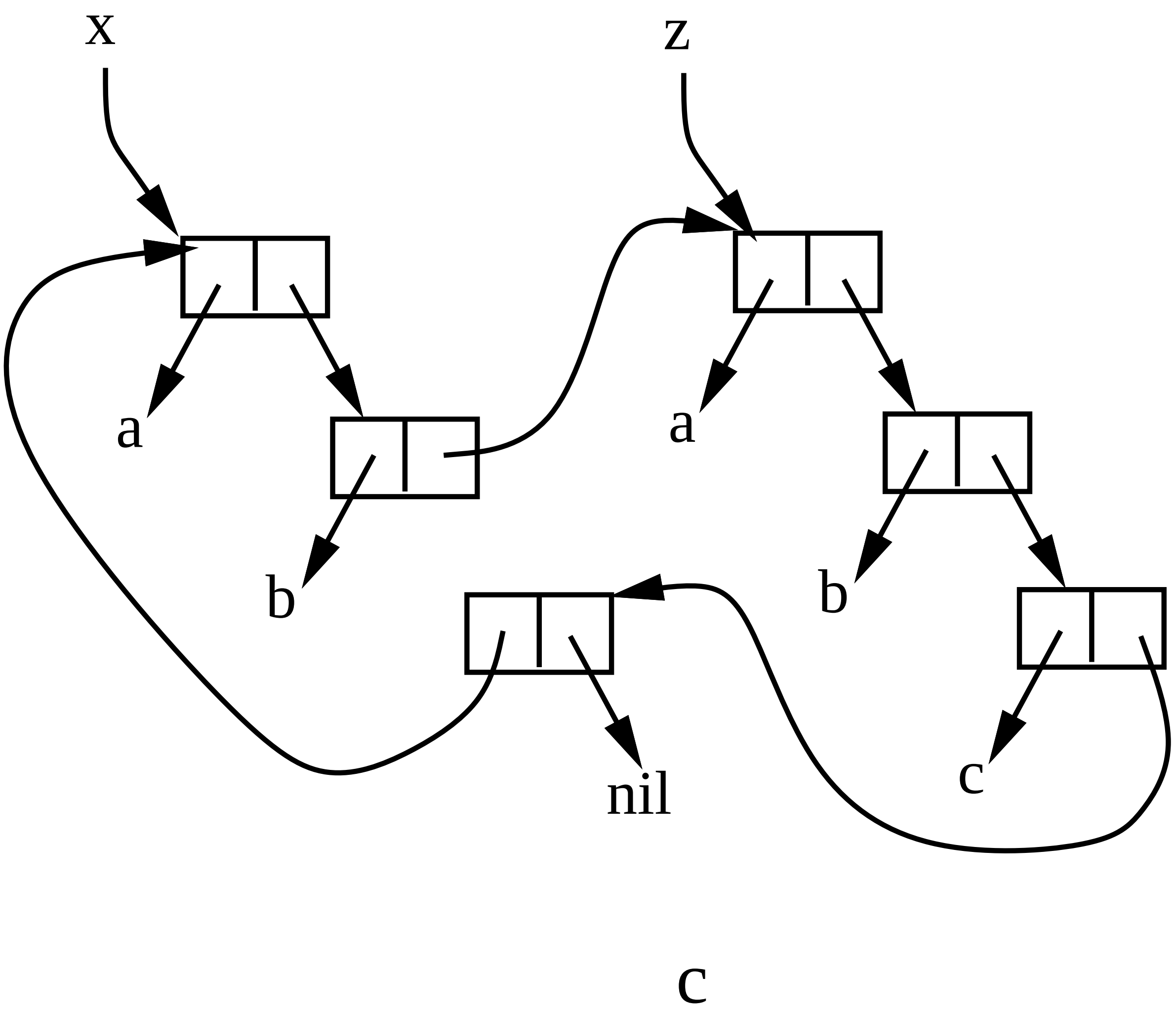


a

Pointers Revealed, cont’d



Pointers Revealed, cont’d



Pointers Revealed, cont’d

- An important point -- non-destructive ops are more efficient than they might appear

- E.g.,

  ```
  (cons huge-list1 huge-list2)
  ```

  takes the same time as any cons

- It just copies pointers

V. Types of languages Lisp can be

A. *Pure Applicative*

B. *Single-Assignment Applicative*

C. *Large-Grain Applicative*

D. *Imperative*

E. *Nasty Imperative*

More on
Functional Programming

VI. "Compelling" motivations

A. Referential transparency, aka no side effects

B. Verifiability

C. Concurrency

D. Other techniques for efficient evaluation, including *lazy evaluation* and *memoization*.

VII. Referential transparency

A. An expression always has the same value

B. I.e., "side effect free"

C. Important implications:

   1. Any expr need only be evaluated once in a given context

   2. Non-nested exprs can be evaluated in parallel

Referential transparency, cont'd

D. Any data modification operator violates referential transparency

  1. Side-effects lead to different values in the same context

  2. E.g.,

Referential transparency, cont’d

```
>(setq z 0)
0

>(defun expr (x)
    (setq z (+ z x)))
expr

>(defun f (x y) (+ x y z))
f

>(f (expr 1) (expr 1))
5

>(f (expr 1) (expr 1))
```

11

VIII. Benefits of side-effect-free programming ...

IX. **Program verifiability**

A. Outline:

1. Provide a spec of input, P
2. Provide a spec of output, Q
3. Prove P{program}Q

B. P a function of *all possible inputs*, Q of *all possible outputs*.

C. Also, language must be formally

defined.

X. Concurrency models

A. Consider

```
>(defun f (x y z) ... )

>(f (big1 ...) (big2 ...)
         (big3 ...))
```

B. big_i are costly computations.

C. A basic form of concurrency is parallel eval of function args

D. Another model is *dataflow*.

XI. **Dataflow evaluation**

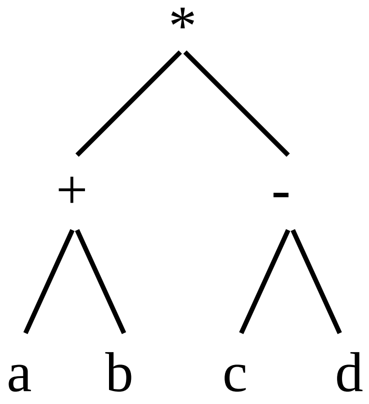
A. Expr eval as tree traversal

B. Sequential, depth-first

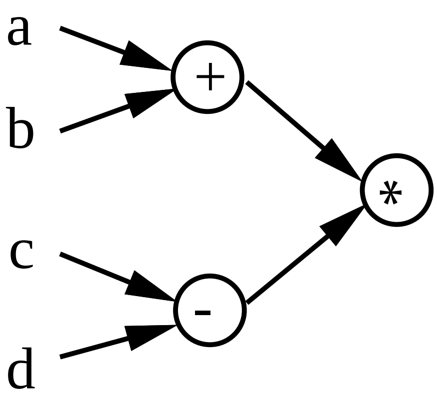
C. In dataflow model:

1. One operator per processor
2. Processor awaits inputs
3. Proceeds independently
4. Outputs results

Dataflow, cont’d



a. Sequential tree-based model.



b. Concurrent dataflow model.

XII. Lazy evaluation

A. Some terminology

1. Synonymous: lazy, strict, demand-driven.

2. Synonymous: eager, non-strict, data-driven.

3. More in future lectures.

Lazy eval, cont’d

B. Normal in most languages is "eager"

  1. Recall fundamental rules for Lisp function eval:

     a. Eval args

     b. Eval function body

  2. I.e., eval *all* args, even if not necessary.

Lazy eval, cont’d

C. Basic idea of lazy eval:

  1. Dont eval args before body

  2. Rather, wait until arg is used

D. Consider

Lazy eval, cont'd

```
>(defun stupid-but-lazy (x y)
    (cond ( (= 1 1) x )
          ( t y )
    )
)

>(stupid-but-lazy 1
    (some-hugely-lengthy-\
        computation))
```

Lazy eval, cont'd

E. Not intelligent, but advantage is clear

F. Can we be lazy in an imperative language?

   1. In general, no.

   2. We cannot guarantee no side effects.

   3. E.g.,

Lazy eval, cont’d

```
>(defun way-stupid-but-lazy (x y)
    (cond ( (= 1 1) z ) ;z free
          ( t y )
    )
)

>(way-stupid-but-lazy 1
                    (setq z 1))
```

XIII. How lazy do we get?

A. When must we perform arg eval?

B. What language primitives are lazy?

C. Consider rules for a lazy Lisp:

How Lazy, cont’d

1. `cond`, `cons`, `car`, and `cdr` are lazy.
2. User-defined functions are lazy.
3. `print` and arithmetic/logical ops are eager.
4. Stop being lazy when eager function "demands" a value, or when we eval a literal.

How Lazy, cont’d

D. Consider:

```
L>(defun (lazy+ (x y) (+ x y)))
lazy+

L>(lazy+ 2 (lazy+ 2 (lazy+ 2 2)))
8
```

E. Trace ...

How Lazy, cont’d

F. Important to understand order of eval.

1. With eager eval, an inside-out order.

2. With lazy eval *(notes typo)*, order is outside-in.

XIV. Lazy eval of infinite functions

A. Can cope effectively

1. Consider

```
>(defun not-so-stupid-but-lazy (x y)
    (cond ( (= 1 1) x )
          ( t y )
    )
)

>(defun infinite-computation ()
    (prog ()
        loop (go loop)
    )
)

>(not-so-stupid-but-lazy 1
    (infinite-computation))
1
```

Lazy infinite, cont’d

B. Potentially infinite *generator functions*

1. Consider

```
>(defun all-ints ()
    (cons 0 (1+ (all-ints))))
all-ints

>(nth 2 (all-ints))
2
```

2. In this example:

Generator functions, cont’d

a. What scheme for lazy eval could work?

b. How exactly does finite execution proceed?

c. What does GCL do with this example?

d. How would *you* implement this?

XV. **Lazy dataflow**

A. A natural idea.

B. A node begins with just enough inputs.

C. A radical approach is fully *demand-driven* eval.

1. All nodes start immediately.
2. A node demands from input line only when needed.

XVI. Memoization

A. Referential transparency implies expr eval only once.

B. An eval strategy:

1. First time, compute the function.
2. Store result for given args in a table -- the *memo*.
3. On subsequent evals, look up args, return memo if found.

Memoization, cont’d

C. Memoization in imperative languages?

  1. Answer same as for lazy eval
  2. Viz., must guarantee side-effect-free behavior.

D. We’ll consider in an upcoming assignment.

XVII. Memoization in dataflow

A. A number of interesting approaches

B. One is to allow dataflow lines to *remember*.

XVIII. To think about

A. Do lazy evaluation and memoization make sense together?

1. If so, how?

2. If not, why not?

B. More to come.

XIX. Concluding thoughts

A. Concepts extremely influential.

B. E.g., C compilers implement memoization and lazy eval.

C. So-called "modern" practices based on functional concepts:

Concluding thoughts, cont’d

1. Lessening use of global vars
2. Defining vars and args constant where possible
3. Formally specifying behavior

D. Ongoing research continues to pioneer new concepts.