CS 35L – Software Construction
Spring 2023 – Professor Paul Eggert

LECTURE 1: INTRO

- Standalone Apps → Runs directly on computer (cs33, machine code)
  - Internet
- Applications atop an operating system → CS31
  - Emacs (eww) → Code that looks at itself (Reflection/Introspection)
  - Operating Systems → Linux
- Web applications →assumes operating system and browser (35L)
- PWAs (progressive web applications)
  - Client-server approach

Big programs →cs130 swe → Application frameworks, Set of libraries/packages + Development philosophy (tools)
Collaborative Development → Git → User (repositories), Internals

Topics

- File Systems (Data on persistent storage)
- Scripting ("Quick and dirty programming" BUT maintainable albeit slower code)
  - Shell, Python, JavaScript, Lisp ☹
- Building and Distribution
  - Program distribution to users
- Version Control (Configuration Management) (Developers vs Users vs Other Devs)
  - Code written by devs is not the same as code being run
- Low Level Debugging (for OS + CS33)
  - GDB
  - C
- Client Server Code
  - Program that operates on two computers at the same time

CS 130

- Programming
- Data Design
- Integration (gluing together software components)
- Configuration (LAUSD Payrolls)
- Testing
- Forensics (Programmer went wrong when not debugging)

Operating System

- Kernel (Applications cannot see this)
- Application Processes (Processes are running applications)

User, Network, Other Files
Application
Other Libraries (i.e. graphics, JSON reading)
C Library
__________ APPLICATION SEES THIS BOUNDARY______99% OF CODE OPERATES ABOVE__________________________________
System Calls (Look like function calls , open or read a file)
Kernel (Prevents program from accessing certain instructions which can breach security)
Instructions
Hardware

Hardware → Software

Applications (Isolation from each other, one app cannot fully access another's memory)
Applications interact via the kernel
i.e. One app writes to file, one reads from file, apps are hooked by file system
Kernel + Machine

SHELL
- Command Language (Scripting language)
- Designed to be run <u>interactively</u> (Program that you write as you think what you want the program to do, you can run programs while writing them)
    - Ctrl+D, ^D, Cmd+D -> End of File (if at start of line)
    - Ctrl+C → Interrupt current process, process dies

Some Apps
- Shell
- Cat
- Ps (process status)
    - Lists current processes
    - Ps -e →lists all the processes running on the system
- Less
    - Interactively control a large amount of data to see small portions of it
    - Control over display of data
    - Text edit that wont let you edit / READ ONLY text editor
    - Less /etc/passwd

        - /eggert/ → searches file for eggert

Ps -e | less

| -> first ones output is second apps input
    e.g. ps output will input into less

Programs do heavy work, shell INTERACTS,
    e.g. Cat app doesn't copy data, the program doesi

cat /etc/os-release /etc/passwd
→Reads from file

\ → cmd continues onto next line
<u>LECTURE 2:</u>
C++ program
Input → program → output
Out = function of program on input

- Emacs the program
- Emacs the process
    - Program that's currenting running and doing something on the machine
    - Program + Internal Data
    - Has input and output
        - Input: Keyboard, Output: Screen (like any program)
        - Interact with windowing system
        - Input and output to files
        - Input/fetch Information to URL, interact or post to URL
        - Start a new process and control it (be notified of when process finishes or stop it from running)
        - Use process for input and output via pipes [interprocess communication data] (e.g. cat)

Latency kills cloud applications connections to windowing systems (speed of light)

Char is 8-bit, ASCII is 7bit set
- 0 is control
- Control char to del character
- C-c is 0000011

Internal data changes when processing data

Emacs internal data (and program as well)

Emacs (scripting language, decisions, control) (85% of source)
Lisp (Objects)

C (low level I/O, Lisp interpreter, CPU, GPU, intensive) (15%)

- Low level objects, Objects and OOP derived directly in C, not built in
- Lisp interpreter is slower than C code

Written and operates in low level code, decisions made by program made by other program (e.g. Emacs, JavaScript, Python)

Control and Meta MODIFY characters but are not characters themselves
Putchar('\r') [Carriage Return]
\r -> ^M
<Tab> → C-i

Emacs commands

- C-x C-c → Exit
- C-q → Input character/Control chars
- M-x shell <Ret>
    - Printf "<C-q C-c>\n" (looks like ^C\n) → prints control C, approximates what cntrl-C can look like
        - ^C display, C-c you typed this
    - C-q C-x
- M-c turns on top (leftmost) bit
- C-z →suspend
- C-x C-f → visit file
- C-a → Start of line
- C-e → End of line
- M-< → Start of buffer
- M-> → End of Buffer
- C-h → help
- C-x C-b → Lists "interesting" buffers
- 2-buffer display
- C-x 1 → Close all other displays/buffers
- C-x 2 → Split the display

- C-x b → Switch to buffer
- C-x C-q → disable read only mode
- C-x h → Refers to the entire buffer (cmd-a, highlights the entire buffer/window)
    - Point (the cursor), Mark ()
- C-SPACE or C-@ → set mark
- M-w → makes a copy of region between marks into temporary area
- C-y → Copy from temporary into current buffer (Yanking)
- C-x d → List the current ≤y
- C-g → Interrupt, END ANY PROCESS
- *scratch* → Address to access Temp buffer
- Emacs commands are generally not case sensitive, C-SP and C-@ are the same (they both type the null byte)
- Fg resumes emacs

Emacs writes and reads to and from file. Emacs takes input from keyboard. Buffer object (like a string, byte sequence, represents the contents of a file, somewhat large), String (sequence of bytes, somewhat small)

Buffers are oriented towards editing

- Insertion is cheap O(1) [String insertion is O(N)]
-

Strings cannot be modified (easily)

Modes (Emacs is a modeful editor, responds to any command but commands may be different based on mode)

- Fundamental
    - F opens directory in directories menu
- Read-Only
- Write-Only
- Lisp Interactive
- Directory Edit
    - R (capital r) renames file

(defun foo (x) * x x)) C-j calls command
foo
(foo 29)
>>841

LECTURE 3:

Emacs IDE + shell → get work done (operations)
files →persistence
Emacs IDE → write programs (development)

Emacs IDE, Shell, Files → Little language to run other programs (configuration language)

ASL: application-specific languages, design language around common tasks within applications

Simple language

- Regular expressions, used in grep etc
    - Grep is standard text searching tool, looks for patterns in text files
    - $ is shell, stdio is pattern every file that ends in .c
    - $ grep stdio *.c
    - $ grep '^#include *<stdio.h>$' *.c
    - Regular expression, ordinary characters, x ø @, match themselves

    - E* → match 0 or more occurrences of E
    - ^E → E at the start of the line
    - E$ → E at line end
    - \C → C is a special character, matches just C, undefined if defined character is passed
    - $ grep '^#include *<stdio.h>$.*/*' *.c
    - * is not a regular experession in *.c
    -
    - [abcx] → matches any single character inside of the square brackets (a,b,c,x 1 character)
    - . → matches any single character
    - IDENTIFIERS / REGULAR EXPRESSIONS / REGEXES
    - [ABC…XYZabc…xyz_][ABC…XYZabc…xyz_0123456789]*
        - [A-Za-z_][A-Za-z0-9_]*
        - _ starts identifiers
        - [-~]
        - [^ABC] matches any char not capital ABC
        - [ABC^] matches
        - *, \ is regular character in regex, [A*] matches As and *s
        - [ ][ ], ending bracket must be put first if used to match
        - [^][(){}]
        - [^-az], minus must be put first if used to match
        - [[:alpha:]] → :stuff: is a character class, matches Chinese/any character
        - [[:digit:]] → matches Arabic, English, Hindi, any digits
            - Same as <ctype.h>
        - Emacs supports "ascii" (non-ascii characters)
            - [[:ascii:]]
        - Grep does not support ascii, so you must used
            - [null-del]
        - $grep '[^ABC]' foo.c
- Thousands of different characters → Unicode
- 16 bits → C unsigned short
    - Change all text formats from ASCII → Unicode
    - 16 bits is not big enough, compatability issues when some duplicate chars between Chinese and Japanese
- 32 bits
    - Takes a lot of memory, can hold all chars (so far)

UTF-8

- Upwards compatible with ASCII

- Space efficient
- Unambiguous when you drop in the middle of a byte string, what and where the next byte is (character boundaries unambiguous)

Some bytes take more storage than others, byte string → char string
Encode char string to byte string, decode byte string to char string
3 kinds of bytes

- [0xxxxxxx] ASCII character
- [10yyyyyy] 2nd or later byte in multibyte character, y is payload, 6 bits
- [110wwwww] <u>number of 1s</u> tells how many bytes are in multibyte char, count of bytes in char, w is leading bits of payload (max is 4)
- Unicode character U+03AF
  11 1010 1111 → 3 A F
  110 1110 10101111

Emacs text encoding

"Hello, world!"

M-x shell
>>od -c foo.txt
>> od -t x1 foo.txt → prints in hexadecimal
right and left double quotes are stored in UTF-8
Electric quote mode

C-u C-x = → gives information on character highlighted by cursor

Errors with UTF

- Invalid bytes give truncating errors
- Encoding error
  - 110 byte followed by byte starting with 0
  - 110 byte followed by 10 byte followed by 10 byte
- Overlong encoding: two representations of same character can cause bugs, shorter representation of character must be used to be valid
  - 110 00000 10 111111 → U+003F
  - 00 111111, two representations of same character, one byte and two byte scan.

Regular expressions / Regexes

- Grep 'a\(bcd\)*e'
  - * applies to subexpression bcd
- Grep 'a\(bed\)\{5,10\}e'

  - Matches digits between 5-10

Extended regular expressions

- Has | or operator
  - Grep -E ‘ab|cd’
  - Grep -E ‘a(bc|de)*f’
- Has ?
  - Grep -E ‘a(bcd)?f’
- + : one or more occurences of preceding expression
  - Grep -E ‘a(bcd)+f’

Basic Regular Expression

Perl

- Grep -p

Java, Javascript, Python has regular expressions, syntaxed differently

Other regex tools

- Sed → stream editor, reads input, outputs output, reads each line
  - Sed ‘s/ab*c/de/’ → substitute de string for instances of ab*c regex matches
  - Sed ‘s/\[/(/; s/]/)/’
- Awk → sed++, Perl expands on awk,

Shell scripting

- Command language for running other commands
  - Find where words are, invoke commands (from /usr/bin/)
  - Grep

<u>LECTURE 4:</u>

Shell, File system, Emacs → Setup language (common shell opp, grep, regular expression, UTF-8) → ordinary shell command

- Shell commands a b c de fgh
  - Command a, arguments b, c, de, fgh

Tokens of shell (what if I wanted a space in my arguments)

- A b\ c d → cmd a with b\ c d\\ as arguments
  - Argv[1] = “b c”
  - Argv[2] = “d\”
  - Special Case: Backslash escapes
- Quoting in shell

        - ‘Single quotes’: a ‘b c’ ‘d\efg’ ‘I can put in any characters except for single quotation marks’
            - ‘hi
              there’ ‘single quotations can store newlines also’
            - A b’c d’e = a ‘bc de’
            - \x = x, newline is special case
            - Writing single quotation marks
                - Echo ‘3 o’ \’ ‘clock’
                - ‘\’’ is a single apostrophe in single quoted string
        - Double quotes
            - A “b$x c d”
                - $ means interpolate x’s value
                - \ is escape character in double quote string
                    - Echo “you paint \$3”
                    - “\\” → \
                    - Echo “you paid \$$n”
                        - \$ outputs “$”, $n interpolates n
                    - “$ “ → Prof think it’s a dollar sign and a space

Not every “command” is a command in shell (some are reserved words)

- Exit → built-into shell, exits shell
    - You cannot create an exit command
- If, then, else, fi
    - If test -f foo → is foo a file?
    - Then
        - Echo “foo is present!”
        - If test -r foo → is r a readable file, if yes, run then, if not, run else
        - Then cat foo
        - Else echo “foo not readable!”
        - Fi
    - Else
        - Echo “foo missing!”
        - Fi
- $? → exit status of most recent command → 0 is good, 1 is bad
    - Consult $? After failure
- Cat: 0 = catted everything, 1 = file not readable, I/O error
- Grep: 0 = match found, 1 = no match found, 2 = serious error (file not readable)

```
If grep eggert /etc/passwd
Then
        Echo eggert known
```

```
Else
        If test $? -eq 1
                Echo eggert is missing
        Else (-eq == 0, grep failed for more serious problem)
                Echo system error
        fi
fi
```

M-x shell <RET>.
You can use semicolons to replace newlines in shell

Test $? → output correct status

```
>>Weird a b c d
>>s=$? → assignment to shell variable
>> test $s -eq 26
```

Different technique
Case, in, esac

```
>>weird a bc
>> case $? In
        >> 0) echo ok ;;
        >> 1) echo missing ;;
        >> 2) echo trouble ;;
        >> *) echo the star is a default case ;; → globbing patterns
>> esac
```

```
File = "________"
Case $file in
        *.c) echo c source ;;
        *.h) echo c header ;;
        *.[a,o]) echo machine code ;;
        Xy|zw) echo 2-letter com ;;
Esac
```

Single quotations by default, double for use of $
GLOBBING STATEMENTS ARE DIFFERENT THAN REGEXES

Globbing patterns
- Foo-*.c

If
Then
Fi
(else is considered true)

Elif acts as else if, doesn't need multiple fis

```
{ cat a; cat b; }
{
  cat a;
  cat b;
}
```

! → negates a command
If ! grep eggert /etc/passwd

Common extension

  !! = text of previous command
  !* = reuse same command as before except for command name (arguments of previous command)
  !$ = last argument

While ____ do ____ done
Break → exit loop
Continue → next iteration of loop
: → true, false
Until (confusing)

```
While test -f foo
Do rm -i foo
done
```

<u>RESERVE WORDS ARE KEYWORDS</u>
TOKENS

- &&, ||
    - Cat a && grep e b
        - If cat and grep work, return grep, if not, return cat

- Grep a f || cat f
    - If grep works, return grep, if not, return cat
- Grep a f || {
  cat f
  echo ach
  }

```
A = b
(
        a=c
        Echo $a
)
Echo $a → b
```

For ___ in __ do ___ done

```
For I in a b 27 c
Do {

}
```

I is called for a, b, 27, c

```
For I in *.c
Do
{}
Done
```

Variables
Lij='tester' → argument after equal sing must be one word
<u>DO NOT ADD SPACES, lij = ' → cmd lij with args =, '</u>
<u>SHELL VARIABLES ARE ONLY STRINGS</u>

A=b c=27 d=fg calls all assignments separately
A=b cat foo
ONLKY FOR CAT ONLY, cast b to a just for cat

PATH=/usr/bin:/snap/bin sh
→ runs ubuntu
Sol
Echo $PATH

Export a=b

$! → Exit status of most recent command that have "waited for"
Process ID of background command
>sleep 100000000 & → run command in background in parallel
Kill

Wait process → waits for process to finish
Built-In variables (arguments to shell script)

- $* is everything but $0
- $@ acts like $* except when quoted
- "$@" → give me all of my arguments as strings

Take output of one command and treat it as shell command
Interesting > f
./f
Run interesting command, outputs stuff, run output

Cat $(Interesting)
Run interesting, take its output, type it into shell and execute shell

Cat $(Grep -v eggert *)→ print out the names of matching files that contain eggert within the file

Cat 'grep -v eggert *' (works but is not preferred)
<u>LECTURE 5: </u>Shell Scripting, Files and File names, Lisp Scripting

a. Effective Shell

How a shell command is expanded before executing it

Script → Transformed into other script in simpler language ("Expanded script:" Expansion occurs all the time (repeats over and over for each command)) → interpreter

Stages of Expansion in a shell command:

1) Delimit the command (find out where it starts and ends)

   a. Commands are in between semicolons [cmd | cmd2; <ENTER>]

2) Now the command is a sequence of words
   a. Tilde expansion: Word **<u>starts</u>** with a tilde
      i. ~ → Expands into home directory
      ii. ~/ → Currect directory
      iii. ~eggert → eggert's home directory
   b. Variable expansion
      i. $x → gets value of x
   c. Command substitution: $(command) → expands output of command and make it input of other command
      i. Ls $(echo ~) → ls /u/cs/fac/eggert
      ii. Cat $('This is terrible') → Attempts to cat this, is, and. Terrible
      iii. $ $(echo exit) → exits server
      iv. Pipes are delimited into commands, do not function normally
      v. Attempts to interpolate binary into command
   d. Arithmetic Expansion
      i. $((x + 5)) → x is interpreted as local shell variable
         1. X = 12 → $((x + 5)) = 17
      ii. $((cmd)) → space required for arithmetic expression
   e. Field splitting: Split words into smaller bits
      i. Word1 word2 word3 → woarda wordb wordc word
      ii. Files = 'a.c b.c'
      gcc $files → run commands with two arguments separated by spaces
      iii. What constitutes fields (what are the separator characters?)
         1. IFS: Interfield Separator, '\t \n', any one of these three characters are field separators
         oIFS=$IFS
         IFS=:
         ls
         2. "$IFS" → gets rid of field splitting, no expansion occurs in double quoted strings
   f. Pathname expansion (Globbing)
      i. * matches any sequence of characters (even empty sequence/nothing)
      ii. ? matches one char
      iii. [] and "" matches chars in a set
      iv. [!] negation (should be avoided)
      v. File names starting with . are not matched by * or ?, as they are considered file names
         1. Ls -a looks at all file names

vi. I/O Redirection: Some words in command are not treated as argument but rather as directions to shell as to what to do with output
   1. Cat foo bar > baz → Send output of cat output to baz instead of default
   2. Cat foo bar > bar → File named bar is truncated to zero, bar is standard output
   3. Cat bar < foo → Standard input comes from foo
   4. – represents standard input
      a. Cat – foo < bar → Read from standard input then foo, where standard input is bar
   5. Cat emacs-26.1 ~eggert/foo > ~eggert/foo
      a. Infinite loop
   6. Overlapping memory is undefined behavior
   7. Each redirection is done independently
      a. Cat <x <y
      b. >x >y → Create two empty files x, y
      c. >> foo → put output at end of foo rather than destroying foo and creating new foo
      d. Cat << EOF → "here document," ends at passed argument (EOF in this case)
      e. 0 is standard input, 1 is standard output, 2 is standard error
      f. Cat foo 1>bar 2>err
      g. 2 >&- close file
      h. 2 > &1 unifies standard error and output
      i. /dev/null is always empty
      j. & MEANS CLOSED

File names
File name ocmponenets
Pathnames

- Tree structured file system
  - Internal nodes of the tree are directories, leaf nodes are files (genereally regular files)
  - Symbollic links
  - Special file
- Leaves can be shared, two names for same file in same directory, multiple links can link to same node
- File names contain byte string that are not NULL
  - File name components: sequence of bit that contain no nulls, file names are built up of file name components, separated by /

- Ln f g → two different names for the same file
  mkdir f d/f
  ls -al . d
- Ls -I → print out internal id of each file
- Find . -inum [id]
  - Print out all name of file of id
- Ln → modifies directory to create new entry to directory

- Symbolic links/Hard links:
  - Rm f → remove directory entry
  - ln -s a b → creates a symbolic link a → b regardless of whether a exists or not
    - ls -l /bin
      - /bin has symbolic link to usr/bin (NO FRONT SLASH)

<u>LECTURE 6:</u>

Lisp + Emacs + IDE → Scripting
Python → Scripting

Shellk scripting/ Sh: command line interaction

```
Script1
        Script2 a b
Script2
        Cat $1
Bigscript
Script2() {
        Cat $1
}

Script2 a b
```

Bigscripts are preferred in Lisp, Python

- Integration, ease of viewing code
- Less jassle in invocation
- Easier packaging
- More efficient
- Less storage overhead
- Everything must be written in shell, breaking scripts into different files means some scripts can be written in different languages for effficiency's sake (e.g. C++)
- Modularization is better with the separate script approach
- **<u>DOES NOT HAVE |: CANNOT INTERCONNECT PROGRAMS LIKE SHELL USING |</u>**

Lisp: Old and Simple, “root language” 1950s, used as an AI language (albeit slow and theoretical originally but practical now)

- Lisp 1, 1.5 1950s
- Scheme 1970s
- Emacs Lisp 1970s

Emacs: Extension Language

- Doesn’t have a main function
- APP: does lots of stuff, app can be programmable, has its own language to be modifiable (user writes code in Elisp)
- Written in Elisp (Emacs lisp)
    - Chrome (Delta written in JavaScript)

Emacs: Object Oriented System

- Numbers, strings
- Buffers, windows, frames
- Process

How to glue these together? (The components)

Numbers and Strings (in Emacs)

- Fixed nums
    - Small integers (small enough to fit in a machine word 2^60smth)
    - 64 bit number,
- Big nums
    - Integers that are too large to fit in an integer
- Floats
    - Floating point numbers, decimals
- Objects are represented by a 64 bit object, which represents a pointer
    - Ex: 64 bit pointer points to C double on SEASnet server
    - Ex: 64 bit pointer points to first bytes of bignum with lots of bytes
- To differentiate between objects we can
    - Create a tag field: allocate some space at bottom bits in the stored data to define the type of data, I.e. double* pointer = smth (less commonly used items generally store this information in the item, sometimes used to define more information when combined with tag bits of pointer)
    - Integrate tag bits in the pointer itself to define what type of data an object is (Generally 3 bits long, can store 8 types of data)
    - Fixed nums: 64 bit number,
    - Value of fixed nums are stored in the pointer itself

C:
Struct lisp fool
EMACS_OBJ a = (EMACS_OBJ & foo) + 3
((long)& foo) % 8 == 0

Emacs:
Gives interaction buffer, open interaction buffer
Runs source code directly in scratch buffer
C-x b <RET>
C-x e → Runs code in any buffer
EX:
0 Position code after cursor and execute C-j to compute
>>0
+ 1 10000000000
>> 10000000001

(= 3 5) → Checks if 3 and 5 are equal via numeric equality (1 == 1.0)
(eq 12 12) → Checks if 64-bit values are same bit pattern (aka same object fundamentally) (1 != 1.0)
EX:
(eq 12 23)
>> nil
(eq 12 12)
>> t
(eq 1 1.0)
>> nil
(eq 10000000000000 100000000000)
>> nil (arguments are bignums and therefore require pointers to be processed, thus we are comparing pointers, which are different as they store different addresses)
(eq “abc” “abc”)
>> nil (pointer difference)
(setq s “abc”)
(eq s s)
>> t

(equal “abc” “abc”) → Less efficient, checks if CONTENTS of pointers are equal
>> t

Floating point quantities
+0.0
-0.0
Same value, different usages
1/-0.0 = - infinity
1/0.0 = infinity

(setq minf (/ 1 -0.0))

(= +0.0 -0.0)
>> t
(equal +0.0 -0.0)
>> t
(format “%s” +0.0)
>> “0.0”
(format “%s” -0.0)
>> “-0.0”
(setq nan (/ 0.0 0.0))
>> -0.0e+NaN (not a number)
(= 0.5 nan)
>> nil
(= nan nan)
>> nil
Nan NEVER equals anything
(eq nan nan)
>> t (only one nan exists, so same address for all nans)

Atom (symbol)

- Single “data point”
- Represented with 64 bits, has tag at bottom (rightmost) 3 bits
- Equal to itself and nothing else
- EX:
    - t nil
      (eq nil 0)
      >> nil
    - (eq nil nil)
      >> t
- You cannot have two atoms with the same name (they have the same address, implemented using a hash table)

List (Lisp is short for list processing)
via cons (pair)

- Compound data structure, two contiguous pieces of data
- Build Pair via cons command
  - (cons 1 3)
    >> (1 . 3)
- Nil is empty list, marks end of list
  - (cons 1 nil)
    >> (1)
    (cons 3 (cons -2 (cons 4 nil)))
    >> (3, -2, 4)
- Car → give first item in pair
- Cdr → give second item in pair
  - (setq li (cons 3 (cons -2 (cons 4 nil))))
    >> (3, -2, 4)
    (car li)
    >> 3
    (cdr li)
    >>(-2 4)
- List builds entire list
  - (list 3 -4 29)
    >> (3, -4 29)

CHARS

- 'a' = ?a
- Chars are integers

Other types

- Hash tables (keys can be anything)
- Char table (specialized for key that are chars)
- Markers (pointer into a buffer, moves with buffer as buffer is modified)
  - Integer displacement does not change when buffer changes
  - EX: "hello ^world" → "hello, ^world"
- Frames, Windows
- Terminal
- Process
- Function

FUNCTIONS

- Lambda functions are nameless
- EX:
  (lambda (x) (+ x 1)) → Expression that yields a function, adds 1 to argument x [+ x 1 is

executed during function call]
((lambda (x) (+ x 1)) 27)
>> 28

SPECIAL FORMS (emacs, looks like a function call excespt first parameter is a keyword)

- EX
  - (if (= a b) (cons a b) (cdr b)) → runs cons if true, cdr if nil [like else]
- You can put as many "else" expressions that you want, the true condition can only have one expression
  - (if (= a b) (cons a b) (message "ouch!") (cdr b))
- EX : if but you can pass multiple expressions if argument is true
  - (when (= a b) (message "x") (cons a b))
- EX: executes multiple expression unless argument is true
  - (unless (= a b) (message "x") (cons a b))
- EX:
  - (cond
    (= a b) a)
    ((< a b) b)
    (t (+ a b)))
- EX: binary: if exp is true, return nil, if nil, return t
  - (not E …)
- EX
  - (and exp1 exp2 exp3) → runs exp3 if nil?
  - (or exp1 exp2 expn) → return first expression that returns nil
  - (while)

Variables

- Local variable assignment uses double paranthesis
- (lambda (x)
  (setq x (+ 1 x))
  (* x x))
- EX: Multiple variable assignment
  - (let ((x (+ y z))
    (w (+ 2 3)))
    (* x x))

Emacs programs are represented as lists

- (3 4 5)
  >> Debugger entered—Lisp error: (invalid-function 3)
  >>…
  - Exit debugging: C-]
- Quoting: single apostrophe defines any subsequent expression as data and not as function

- EX:
  (let ((x) ‘(3 4 5)))
  (car x))
  >> 3
- (let ((x) ‘(list 3 4 5))) → list is stored as DATA due to ‘
  (car x))
  >> list
- (let ((x ‘(let ((y 3)) (* y y))))
  (car (cdr x)))
  >> (y 3)
- (let ((x ‘(let ((‘y 3)) (* y y))))
  (car (cdr x)))
  >> ((quote y) 3)
- (setq a 27)
  >> 27
  (list a 39)
  >> (27 39)
  ‘(a 39)
  >> (a 39)

<u>LECTURE 7:</u> Python, client servers and JavaScript

Lisp is an extension language, however Python is “in charge”

line = ‘GOOG, 100, 106.10’
types = [‘str’, ‘int’, ‘float’]
fields = [t(v) for t, v in zip(types, line.split(‘,’))]
>> [‘GOOG’, ‘100’, ‘106.10]
>> [(str, ‘GOOG’), (int, ‘100’), (float, ‘106.10’)]
>>fields = [‘GOOG’, 100, 106.10]

Motivation/history

- Computer Science Education
  - 1980s: BASIC in high school (became more complex by 1980)
    - Introduced in 1962 as teaching language (Fortran was too complicated)
    - Pain to teach (e.g. learn to unlearn)
    - Had to implement and program sorting methods and hash tables
  - ABC (made by CWI Amsterdam → “Let’s replace BASIC!”)
    - Indenting required
    - Comes with IDE
    - Standard sorting function (a.sort())
    - D[‘x’] = 27 (Hashing)

- Little language: Write things in language that makes most sense, glue it together using the shell

  sh sed

  awk grep
    - Knowledge of multiple languages required to properly use “little language” approach
- Perl: Scripting language
    - Unification of sh, sed, awk, grep
    - Wall (‘linguist by trade’) → “There are multiple ways to approach problems”
        - EX: These codes are the same
          ```
          if (a==b) c=d;
          c=d if (a==b);
          ```
- Python: Negative reaction of Perl
    - “There’s a good way to do it, and Python has it”
    - Indentation required
        - Ex:
          ```
          if a ==b:
             c = d;
             e = f;
          if (a==b) c=d, e=f
          ```
    - Strings
        - Double quotes and single quotes are allowed
        - ‘’’Multiline strings

          are allowed’’’
    - Object Oriented: In Python, every entity is an object
        - Every Pythonic object has:
            - Identity (roughly its address, identity cannot be used in pointers (used like address in C++))
                - Points to data definitions of entity
                - Immutable, cannot be changed
                - Id() → Identity of object as integer, cannot be used practically
            - Type
                - Immutable, cannot be changed
                - Type() → Type Object (that describes the type)
                    - Type(type(o)) → Type Type
            - Value
                - Mutable based on object’s type
    - Python built-in types

- None → literally None (kinda like nullptr)
- Numbers: int, float, complex, boolean
    - 1 + 2j is complex
    - Booleans: 0s and 1s
    - Like elisp, ints have no overflow issues
- Sequences: lists, strings, tuples, buffers (strings that can be changed)
    - Indexed by integers
- Mappings: dictionaries,
    - Indexed by other values
- Callables: functions, methods, classes,
- Internals: code, tracebacks

Sequences

- Like an array, sequenced by indexes (numbers) from 0 to N-1
- S = [0, 1, 2]
    - 0 <= I < len(s)
    - -len(s) <= I < len(s)
- Operations:
    - S[i]
    - S[i:j]
        - Takes subsequence of sequence, I up to but not including j
        - has length j-i
        - I defaults to 0, j defaults to len(s)
    - S[i:]
    - Car: s[0]
    - Min(s)
    - Max(s)
    - List(s)
- Mutable sequences
    - S[i]=v
    - S[i:j] = t
        - Can grow or shrink list
    - Del s[i]
    - Del s[i:j]
        - Different from s[i:j] = []

List Operations

- L is list
    - Lists are not linked lists internally
    - Lists store header data, pointer to list elements, value, alloc tells size of list

        - If len = alloc, python allocates list daouble the size of current one, points header pointer to new list, inplement like normal
- L.append(v)
    - Stores a copy of v
    - O(1) amortized (in the long run its O(1))
- Insertion
    - Cost: n * (1 + ½ + ¼ + 1/8 + …) → 2*n
- Extension: l.extend(s)
- L.count(v)
- L.index(v)
- L.insert(I,v)
    - Inserts v at index i
    - Typically O(len(s) - i)
    - Worse Case: O(len(s))
- L.pop(i)
    - Remove and return i-th element
- L.pop() = l.pop(-1)
- L.remove(v) = l.pop(l.index(v))
- L.reverse()
- L.sort()
    - Sorts inplace
    - Assumes < operation

Strings

- Immutable, cannot be changed internally
- S.join(t)
    - S is string, t is sequence of strings
    - ';'.join(['ab', 'bc', 'bcd'])
      >> 'ab;bc;bcd'
- S.split(sep)
    - 'ab;bc;bcd'.split(';')
- S.replace(old, new)
    - 'ab;bc;bcd'.replace(';', '/')
      'ab/bc/bcd'

Python dictionaries

- D[k]
    - D = {'abc' : 1, 'xyz' : 27}
      D['xyz'] = 27
- Keys **MUST** be immutable

- KeyError → Key is absent from dictionary
- D.get(k)
- D.get(k, v)
    - If key is absent, return v
- D.has_key(k)
- Del d[k]
- Len(d)
- D.clear()
- D.keys()
- D.values()
- D.update(e) → Assigns e's keys and values to d's keys and values
    - For k in e:
        d[k] = e[k]
- D.popitem()
    - Returns a key value pair

Functions
- *c trailing arguments passed as tuple
- **d can pass any number of arguments, stored in dictionary
- Def f(a, b, *c, **d)
    - Len(c) → c = (21, 13, "abc")

- Def f(b = 2, a = 3, de = 19, gh = 27)

Classes

```
Class c(a, b)
  Def m(self, x)

O = c()
o.m(19)
```

<u>LECTURE 8:</u> Python Classes, Client Servers

Python

Classes: Can have parent and child classes
- Closest implementation of method is called.
- In c++, if same method exists between two objects and is inherited, crash
- In python, it trails through ancestor tree, order of parents matter

class c(a, b):

```
  def m(self, x):
  def __init___(sdf, x, y)
  def __str__(self):
  def __cmp__(self, other):
    # -1 for <
    # 0 for =
    # +1 for >
    # ? → Used for NaN arguments
  Def __hash__(self):
  Def __lt__(self.others) -> bool # this is less than function
  Def __nonzero__(self):
  Def __add__(self, other):
```

Code is dynamic, can be changed during runtime
Let c be class,
c.__dict__ maps class names to values (is a dictionary, maps names associated with class with its values)
c.__dict__
>> { …, ‘m’, … }
c.__dict__[‘n’] = …
o = c(10, 19)
print(o) → calls __str__ of object o
if (o): → calls __nonzero__ of object
o + j → calls __add__ of objects

Client-side servers

- "faster lectures"
- Lots of underlying technology
- Practical problems in using JS
- Node.js, React: Javascript + POSIX + OS + shell (used for configuration)
- Configuration
  - Quoting issues

Client Server system

[Client] ←--------------→ [Server]

Network (Cloud)

- Not the only model in distributed computing
  - Multiple clients can connect to same server via network/cloud
- Server is in charge

Alternate approaches

- Multiple clients talk to multiple servers (to spread the load) via the network
- Peer to peer
  - Different machines on the network are not client or server, each machine maintains some aspect of the server, no specific responsibility for sole computer, responsibilities spread across multiple computers.
- Primary (generates questions) connects to node computers (who search for answers for questions)

Performance Issues

- Still concern over Big O notation
- Issues because application is distributed (e,.g. information shared between Client and Server, not stored locally)
- Categories
  - Throughput: How much total work is the system doing over time, how many queries/second, more server-side
  - Latency: More client-side, how much time does it take between client asking questions and response returns, what is the delay between request and response?
    - Throughput and Latency are often competing goals
      - Group processing to improve throughput

Sample strategies

- Throughput
  - Batch requests: Wait for multiple similar requests to save RAM on processing, out of order execution

- Latency
    - Client-side caching: Don’t send query to server, reuse answer to local question, client remembers recent answers and uses it to answer same or related questions without contacting server
        - Correctness Issues:
            - Cache validation: Check cache cheaply
            - Stale caches: Local caches may be invalid (e.g. Current temperature/time)
            - Aggressive caching: Prefetching too much information from server, server and client may get slower

The Internet (and some other alternatives)

- Before internet, the telephone system (land lines)
    - Circuit switching: Creating switch paths, only one person can access and use switch path at a time
        - Downsides
            - Incredibly inefficient (requires wires across country, most wires are unused)
            - One switch glitch destroys entire system
        - Upsides
            - While you’re conversing, only one person can have the wires reserved
            - No glitches in conversation
        - WW → Brooklyn, contact central office at Westwood, CO contacts switch in DTLA, switch contacts switch in Phoenix, AZ, so on, so on, reaches switch in Manhattan, contacts central office at Brooklyn
    - Packet switching: Paul Baran 1961
        - Break communication into small packets
        - Route of packet is sent dynamically
        - Upsides
            - Robustness in unreliable network (if one node shuts down, the system still functions fine)
            - More efficient, less idle time. More throughput (we want to send as much data through system as possible, we want to use as many wires as possible)
        - Downside
            - Packets can arrive out of order
        - WW sends Packet 1 to Central Office, Central Office → Reno → Chicago → Manhattan → Brooklyn

Packet switching basics

- Header (meta info), payload (data for app)

3 problems with sending packets

- Don’t receive packets in same order as sent (Out of order)
- Packets can be lost (if RAM is overloaded or out of order)
- Packets can be duplicated (network is misconfigured)

Protocols: Can be complicated,

- To manage complexity, protocols often use layers
    - Top layer: application layer: WWW apps, request webpages, file transfer protocol
    - Transport layer: Focuses on maintaining integrity of data stream, fixes packet-sending problems
    - Internet layer: Deals with routers instruction such that it can arrive at proper destination, Internet protocol (IP)
    - Bottom layer: Link Layer: Protocol that one node on the network uses to talk to an adjacent node, Point-to-Point, hardware oriented
        - E.g. phoenix contacts Tucson

IP: Multiple versions

- IPv4 (1983)
    - Packet header, what it contains
        - Length
        - Protocol number
        - Destination address (32 bit int)
            - 192.54.239.17
        - Source
        - Checksum (16 bit)
        - Time to live field (TTL): Each hop subtracts 1, if 0, delete packet, prevents infinite loop of data movement
- IPv6 (1998)
    - Larger address space (64 bit I think?)
    - Can pretend to be IPv4 (as not all servers are IPv4)

Transport layer protocols

- UDP: User Datagram Protocol (David Reed, MIT)
    - Thin Layer on top of IP
    - App’s responsibility to deal with packet loss/duplication, UDP is not reliable
- TCP: Transmission Control Protocol (Cince Cert, UCLA, Bob Kahn, Princeton)

    - Built atop IP (sometimes called TCP/IP), provides illusion of reliable, ordered, error-chode? stream
    - Defines a data stream in packet
    - Retransmission
    - Reassembly
    - Flow control (Accounts for if sender sends too much data that could overload system)

WWW Apps

- HTTP: HyperText Transfer Protocol
    - T Berners-Lee: Invents world-wide-web to make physics paper more accessible
- RTP: RealTime Protocol
- SGML: Standard Generalized Markup Language
    - Standard interchange for publishers of books
    - <p> body of para </p>
- TCP+
    - HTTP: client-server protocol with requests and responses
      Client: GET .index.html HTTP/1.1 \r\n
        [\r\n
      Server: 200 OK
        Metadata about webpage
    - Content type, length, transfer encoding (UTF-8 or smth else)
      empty line
        webpage contents
    - Metadata about webpage
    - HTTP/1.1
        - Multiple request/responses

<u>**LECTURE 9:**</u> **World Wide Web**

www
html
http (bidirectional TCP)
js

Layering:
Top → HTTP/1/2 → datastreams, TCP → packets, IP → Bottom

NETWORK PROTOCOLS [CS 118]
I/O

- Simple request/response
    - CC → S

    - Get classes/CS35L HTTP/1.0
    - Server → Client
        - Header notation: Timestamp, Content-type, text/html, length
        - Body:

HTTP/2: Mostly about efficiency (2015)

- Header Compression
    - Less time to debug, however more and more header information processed
    - Client and Server must agree how headers are used
- Server push
    - Avoids Client-pull (in response to request), server can send response back whenever they want
- Pipelining
    - HTTP 1: Client setup → Client request → server response → Teardown
    - HTTP 2: No Teardown, Setup can send multiple requests without waiting for response, server decides what and when to response (returns responses out of order)
        - Requests and responses must be associated with labels to keep requests in order
        - Code isn't line by line anymore
- Multiplexing
    - Single TCP connection for multiple conversations/sessions
    - Network can exchange information with Server
        - Server can have multiple conversations with multiple domains from same network
    - Network can host different, similar domain names in same network
- ISSUE: Head of the Line Problem
    - Sent packets 1, 2, 3, 4 [video]
    - Received packet 1
        - This is jnot good for realtime videos

HTTP/3 (2022)

- Built atop QUIC (not TCP)
    - QUIC is TCP v2 with UDP
        - QUIC is more complex protocol
        - Generally gets packets in order
        - gives advantages of TCP (giving information in correct order)
- Even more multiplexing

HTML (derived from publishing format SGML)

- Declarative, not imperative

    - Imperative, CS 31, writing a program that tells the program how to do something, program unfolds. Set of commands to produce a document
        - Hard to port, each company had different standards
        - Print a paragraph by spacing out a line, let there be a temporary indent of 2m, start paragraph with text, end paragraph with sp 0.25
    - Declarative: Mark-up that expresses document semantics (meaning of document) without specifying printing details
        - Focus on document and not how it's formatted
            - <p> </p>
        - I have the start of a paragraph, here are its contents, here is the end of the paragraph
        - CON: Lost the ability to add details in content

Terminology of HTML (further derived from SGML)

- HTML Element: Uses tags, starts with <>, ends with </>
    - <p>This is a paragraph </p>
    - Elements can contain nothing (be opening and closing tags, referred to as void elements)
        - <br/>
- Elements can have attributes (name value pairs associated with elements given in header tag)
    - Attributes should be distinct
    - Attributes can be arbitrary and any string
        - <section id="intro"> </section>
- Raw text elements
    - Can contain only text
    - Normally elements can nest
- <a href="https://www.ece.ucla.edu/index.html#intro"> Click Here! </a>
        - Protocol: https
        - Domain: ucla.edu
        - Resource: index.html
        - Location in Resource: #intro
    - Links files, other webpages
    - <link> used for header
    - <a href="#intro"> Click Here! </a>
        - Intro id of current webpage

Webpages can be representation for a tree:

```
<html>
        <head>
            <meta>                    </meta>
        </head>

        <body>
            <h1>              </h1>
            <section id=”abc”>
                <p>                </p>
            </section>
            <br/>
        </body>
</html>
```

Html

- Head
  - Meta
- Body
  - H1
    - Text
  - Section id=”abc”
    - P
      - Text
  - Br
    - Void

Issues with HTML:

- What can go wrong in HTML (How to troubleshoot)
  - Improperly closed tags

    -
      ```
      <html>
        <body>
          <p> Hello!
        </body>
      </html> </p>
      ```
  - Missing closing tag

    -
      ```
      <html>
        <body>
          <p> Hello!
        </body>
      </html>
      ```
  - Be strict in what you generate, be generous in what you accept
    - HTML can autogenerate closing tags to make missing tags function

- How to upgrade? (Update HTML format, upgrade to the web)
    - Radio button choice (Choose 1)
    - Multiple choice (Choose n)

        -
          ```
          <menu type="multi">
            <choice>        </choice>
            <choice>        </choice>
          </menu>
          ```
    - Muddle through (Trust that webpage can create program by filling in the gaps)
    - Client Request specifies which HTML flavor it understands (servers must support many HTML flavors)
    - Server specifies HTML flavor in its response (clients must support multiple versions, or do "0"/ignore the problem and have client figure it out)
        - Which HTML version?
            - Document type definition (DTD) (from SGML)
                - Lists all elements and their allowed attributes, subelements, (void)(raw)
                - Kind of like grammar
            - DTD for HTML 2, 3, 4, 5….
                - Whether the closing tag is optional
                    - These standards were late
        - HTML5 (evolving standard)
            - Updated with Git

HTML got complicated

- XML: Bare bones for HTML syntax with no sematics
    - Simple way to send tree structure data
- <xyz x="abc">
  </xyz>
- XHTML: Strict HTML
    - Closing tags required

Document Object Model (DOM) [for any object oriented language, but commonly used for JavaScript]

- Object Oriented Model for Tree Data Structures
    - Nodes are elements with attributes
    - Leaves are text and other stuff
- Examining and manipulating
- Imperative aspects
- Cascading Style Sheets (CSS)
    - Partially Imperative, partially Declarative
        - Code is declarative, interpretation is imperative

    - Separate declarations into two categories
        - Content: Words in paragraph
        - Presentation: How the paragraph looks on the screen to user
            - Comes from user preferences
                - Font size
            - Comes from element involved
                - H1 has specific text size
            - Ancestor element

JavaScript

- Programming language designed for browsers
    - Embedded in web browser (e.g. Firefox)
- Can be hooked into HTML
    - <script src=”javascript.js”></script>
    - <script>alert(“Hello!”)</script>
- Errors are more prone
    - <script>alert(“Hello!)</script>
        - Missing Quote, JS wont do anything
    - <script>alert(“Hello!”)<script>
        - Begins reading HTML as JS as it’s never closed