CS131 Week 10

Week 10 Lecture 1

==========

OO languages
Ada

- **call by result**
  - callee starts with variable being uninitialized
- **call by value-result**
  - call by value + call by result

C, C++, Scheme, ...

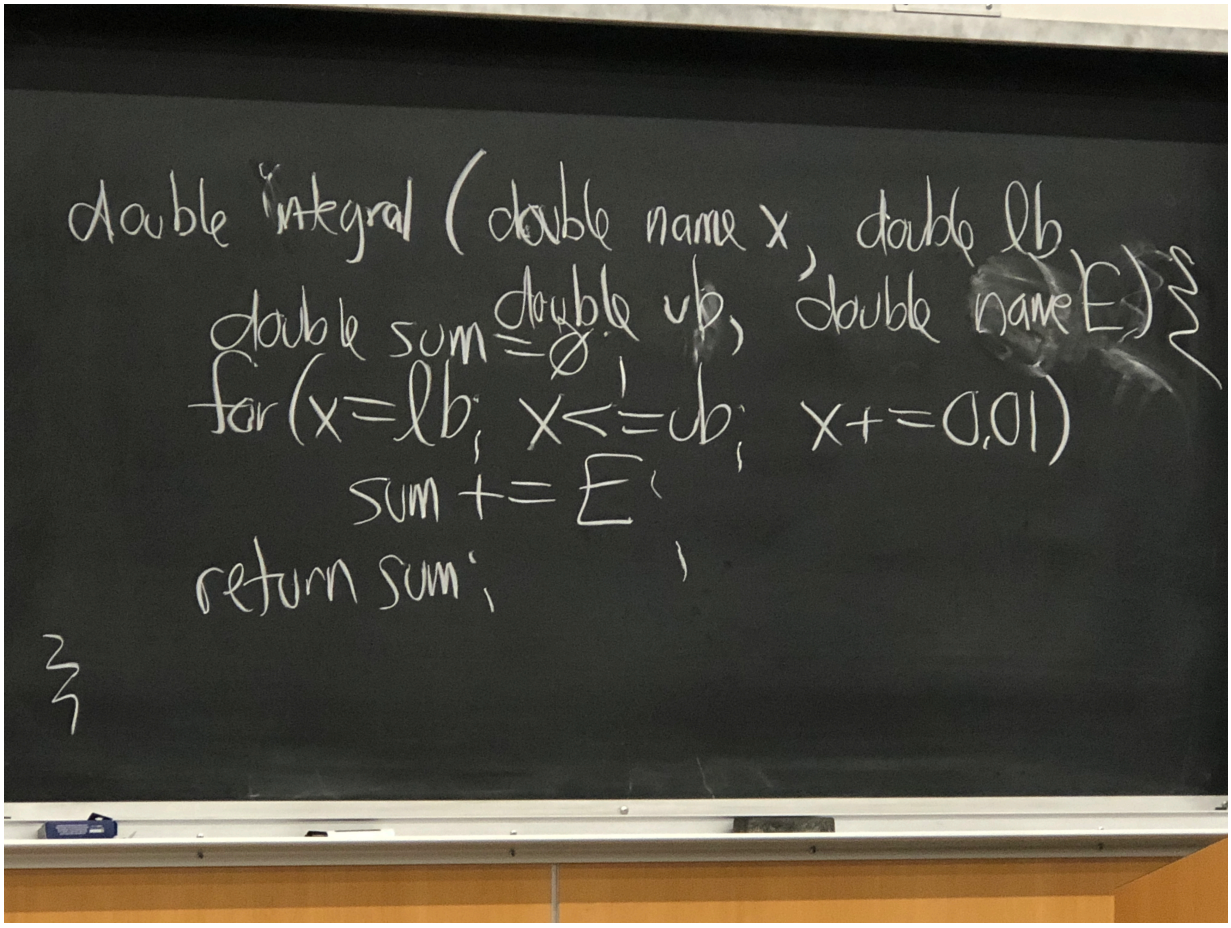
- call by macro expansion
  - getchar() = stdin -> p++ ...
  - scope is tricky (problem of capture)
  - recursion is limited
  - debugging is painful
  - (+) efficiency
  - (+) flexibility

Call by name = procedures
Call by reference = pointers

Callee	Caller
int f(int name x) { return x + 5 ; }	f(m + n)
int f(int (**) (void)) {return x() + 5;}	f(fun() -> m + n)

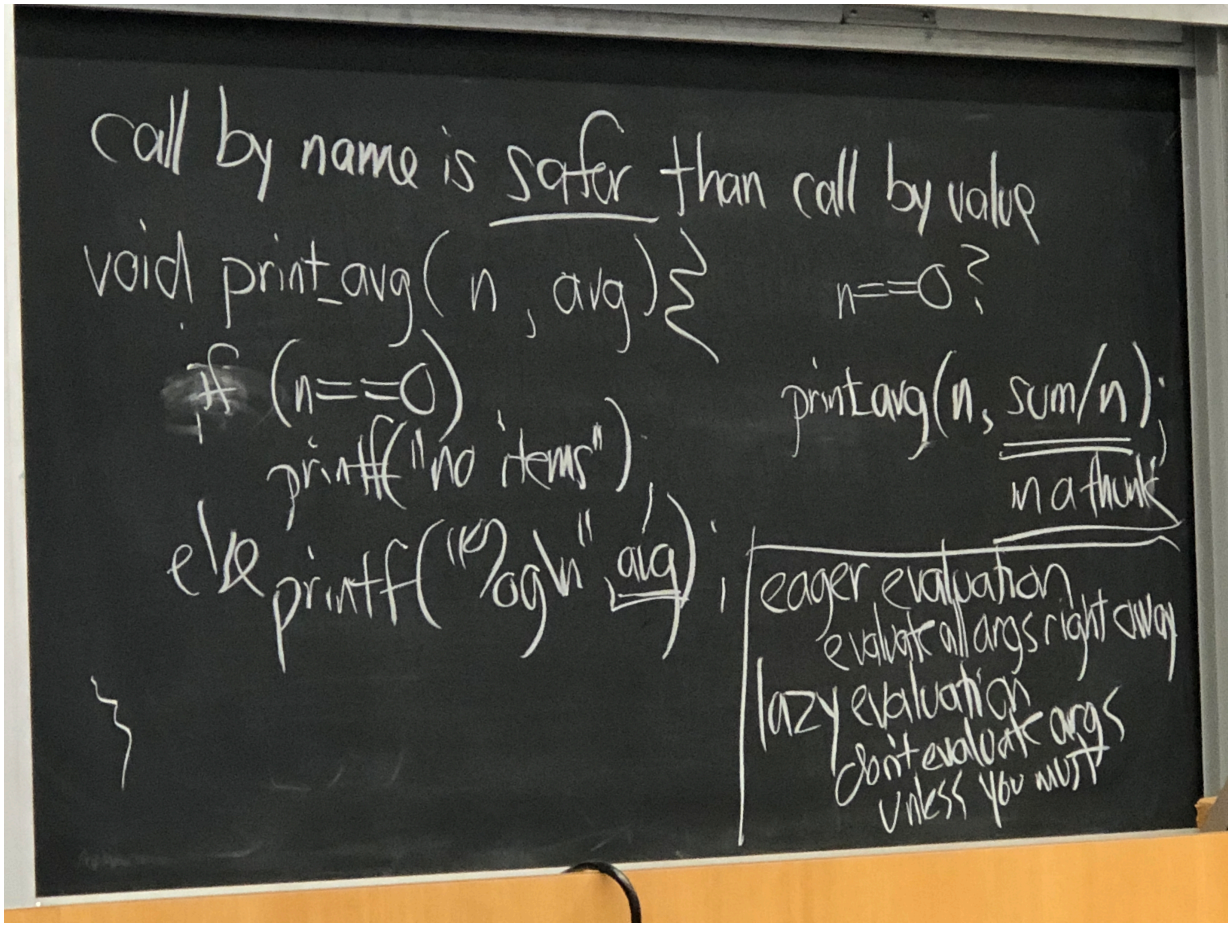
e.g. numeric integration

```
x = 5;
integral (x, 0, 100,          sin(x) + cos(x))
          ^ call by name     ^call by value
```



Single vs multiple inheritance

- can a subclass delete parent methods?
- what do you inherit?
  - methods
  - instance methods
  - Eiffel invariants
- Subtyping vs subclassing
  - e.g private inheritance in C++



Call by Need

- callee: call a thunk at most once, reuses its returned value

In purely functional program, call by need is more efficient version of call by name

analogy between call by need vs value
Haskell: call by need
OCaml: call by value

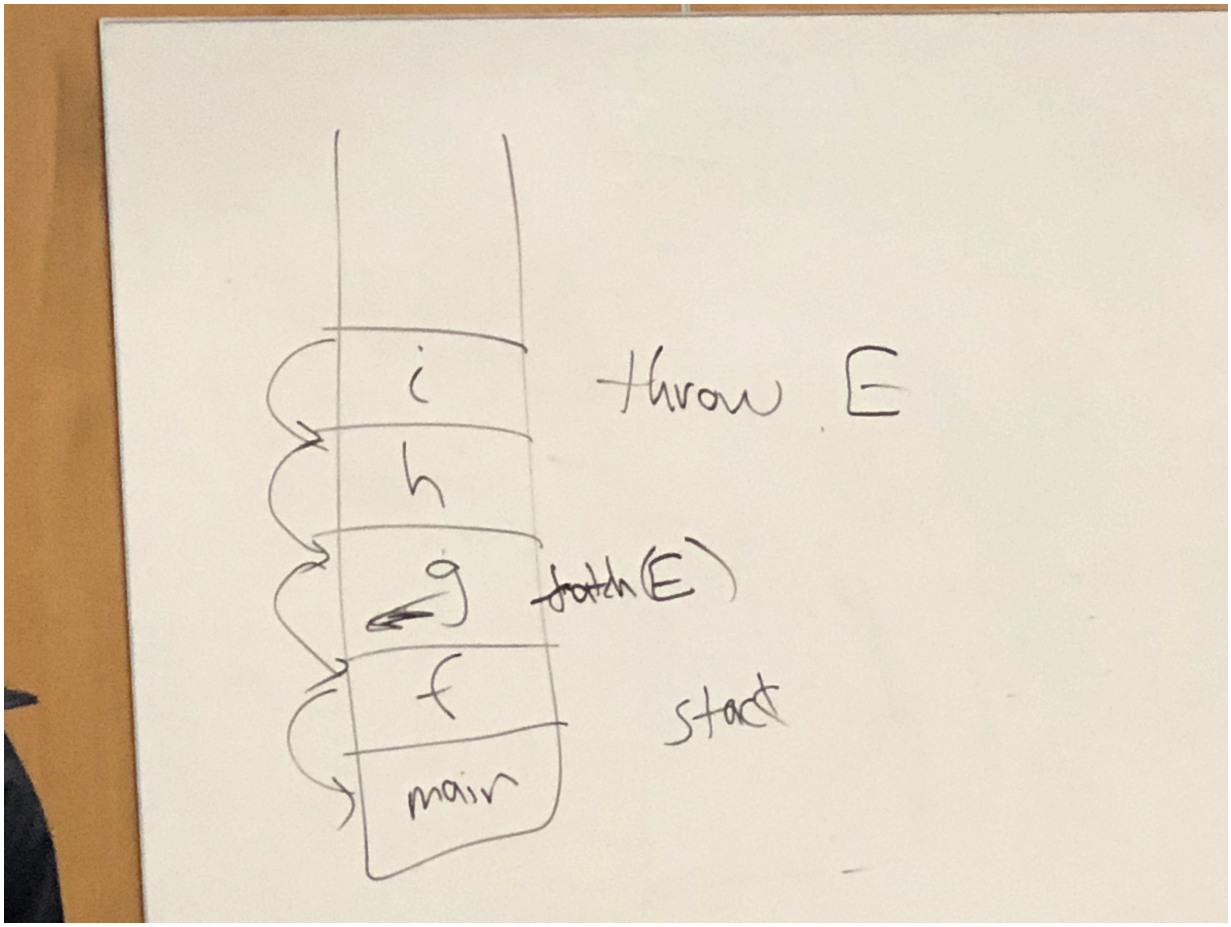
***2nd half of lecture

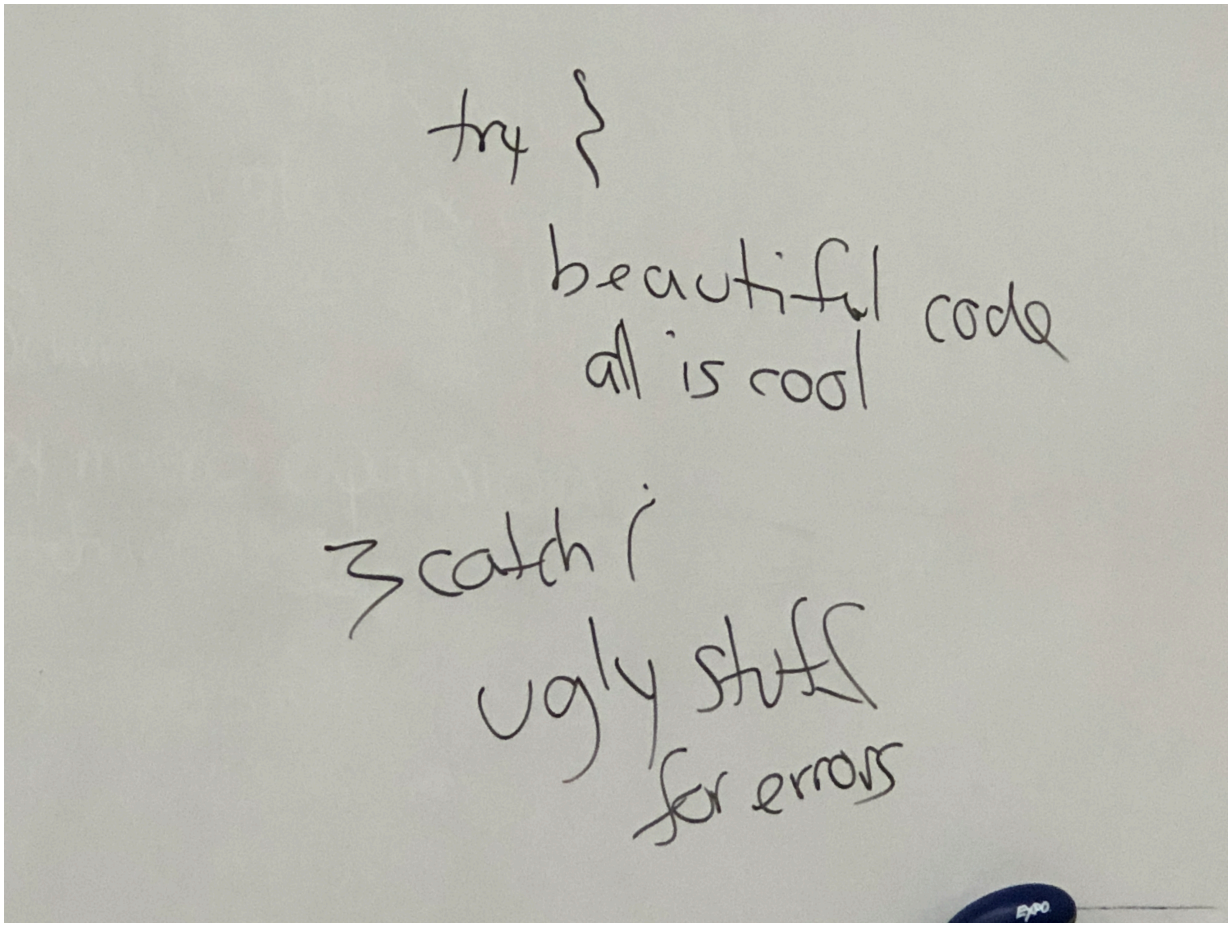
Agenda
OO Language
Parameter Passing
Exception Handling

Error: in programmer's head
Fault: latent problem in program
Failure: bad execution with observable effect on program execution

How to handle "errors"

- Compile-time checks
  - aka static checks (type checking in Java, OCaml)
  - (+) most reliable
  - (-) flexibility
  - e.g. *p; if (p != NULL) return *p ;
    - not safe
  - e.g. int f(int * nonnull p) { *p = 27; }
- Preconditions
  - logical condition that caller's responsibility is to guarantee it's true
  - double sqrt(double x) precondition x >= 0;
    - it's caller's fault if things break: didn't guarantee precondition
  - way of telling whose fault it is if things break
- Total definition
  - sqrt(-1) NaN
- Fatal errors
  - sqrt(-1) dumps core
- undefined behavior
  - sqrt(-1)
- Exception handling
  - Part of program throws exception, the rest of program needs to figure out what to do with it
  - Part static part dynamic
  - Static:
    - Always uses try { .. } catch () {.. }
    - compiler can check for try catch and know it might need to jump to catch code
  - Dynamic
    - Needs to know if thing is thrown after runtime: depends on how machine state goes


i
h
g
f
main
throw E
catch(E)
start



Exception handling might inhibit parallelism:

- i.e. if f(x) + g(y)
  - f(x) might run on one thread
  - g(y) mihgt run on other thread
  - g(y) throws exception but f(x) is ok
  - what happens to values?

Week 10 Lecture 2

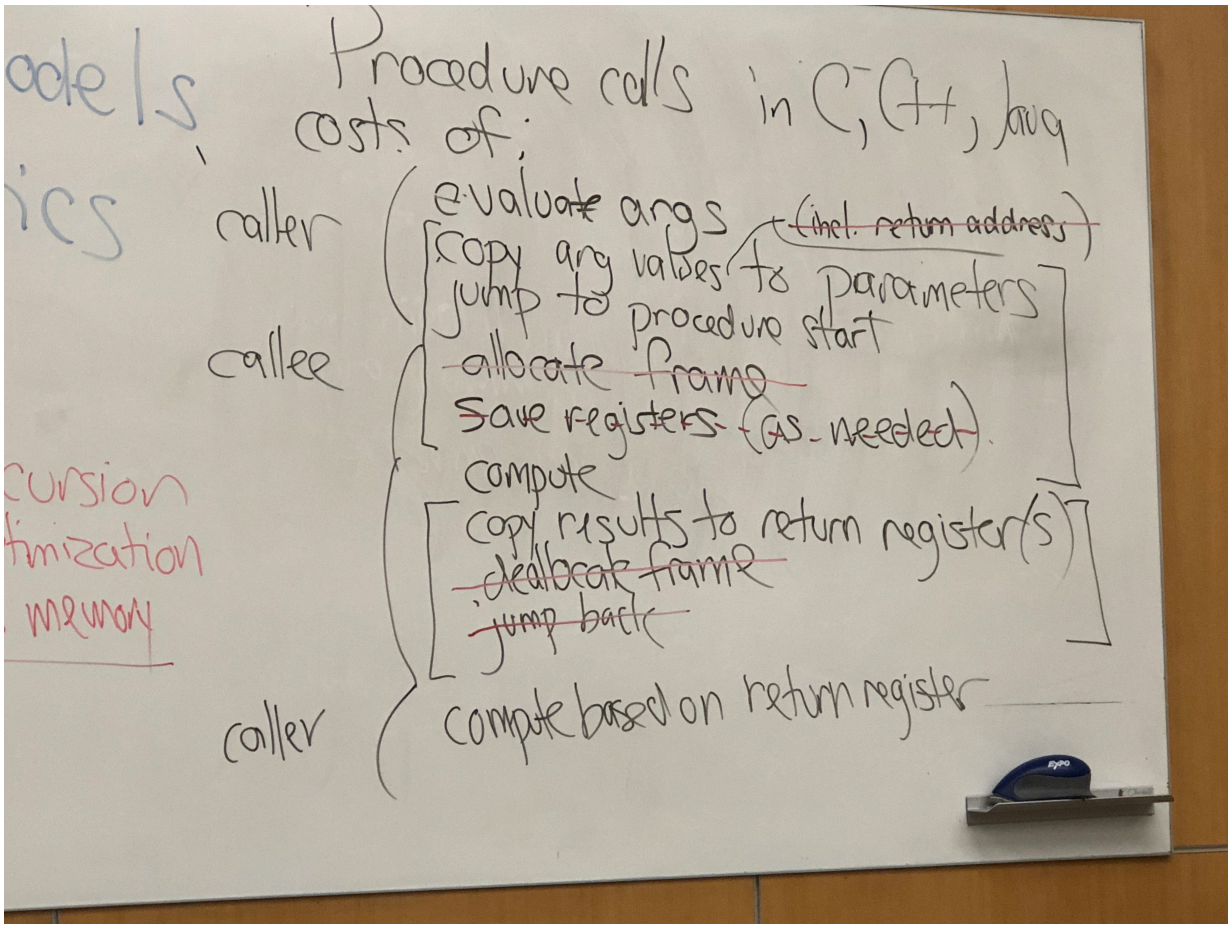
==========

Cost models
Semantics
History

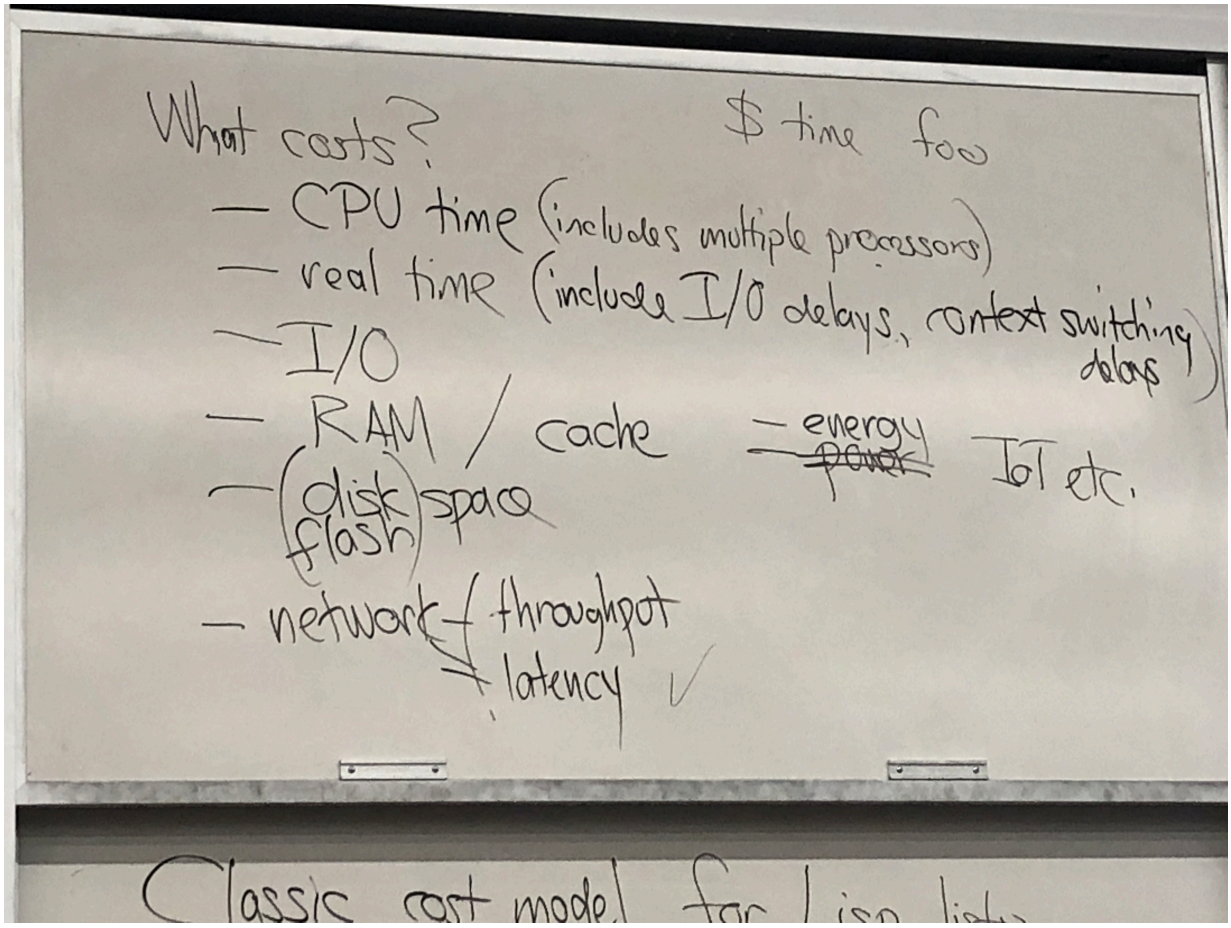
Optimizations to save memory
tail recursion optimization saves memory
inlining saves EVEN MORE

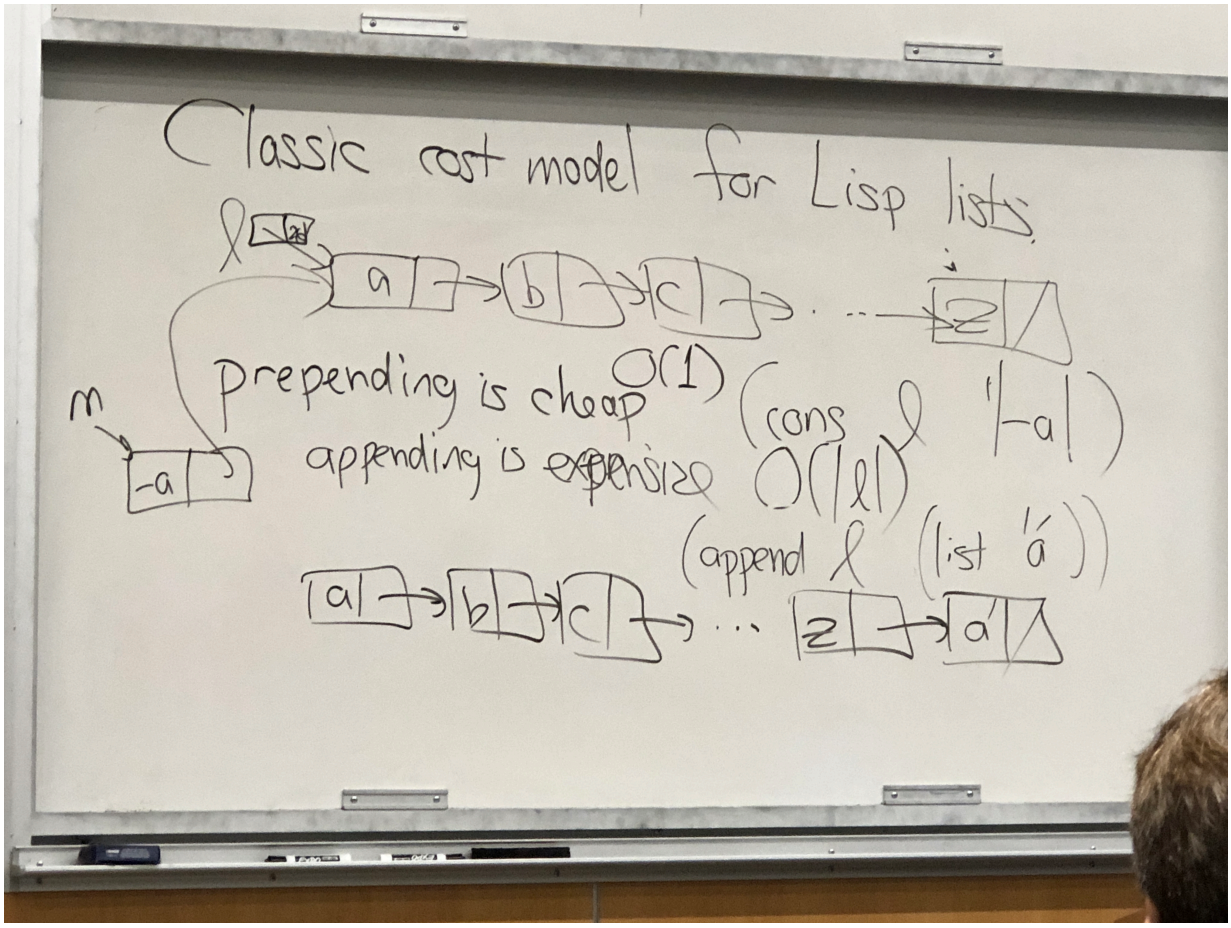
- in diagram below, gets rid of
  - copy arg values to parameters
  - jump to procedure sart
  - allocate frame
  - save registers (as needed)
  - copy results to return registers
  - deallocate frame
  - jump back
- CONS:
  - grows code
  - kills recursion
  - debugging pain
  - tight coupling between caller & callee

Procedure calls in C; C++, Java



Info I missed


What costs?
$ time too
— CPU time (includes multiple processors)
— real time (include I/O delays, context switching delays)
— I/O
— RAM / cache
— energy
power
IoT etc.
— disk
flash
space
— network
throughput
latency
Classic cost model for Lisp list


Classic cost model for Lisp lists
l
a
b
c
z
m
-a
prepending is cheap O(1)
(cons l '-a)
appending is expensive O(|l|)
(append l (list 'a))
a
b
c
z
a