Lecture 26

CIS 341: COMPILERS

Announcements

- Project 7: Oat programming
  - Due: May 5th

- Final Exam:
  - Tuesday, April 30th noon-2:00 pm
  - Moore 216

REFERENCE COUNTING

Reference Counting

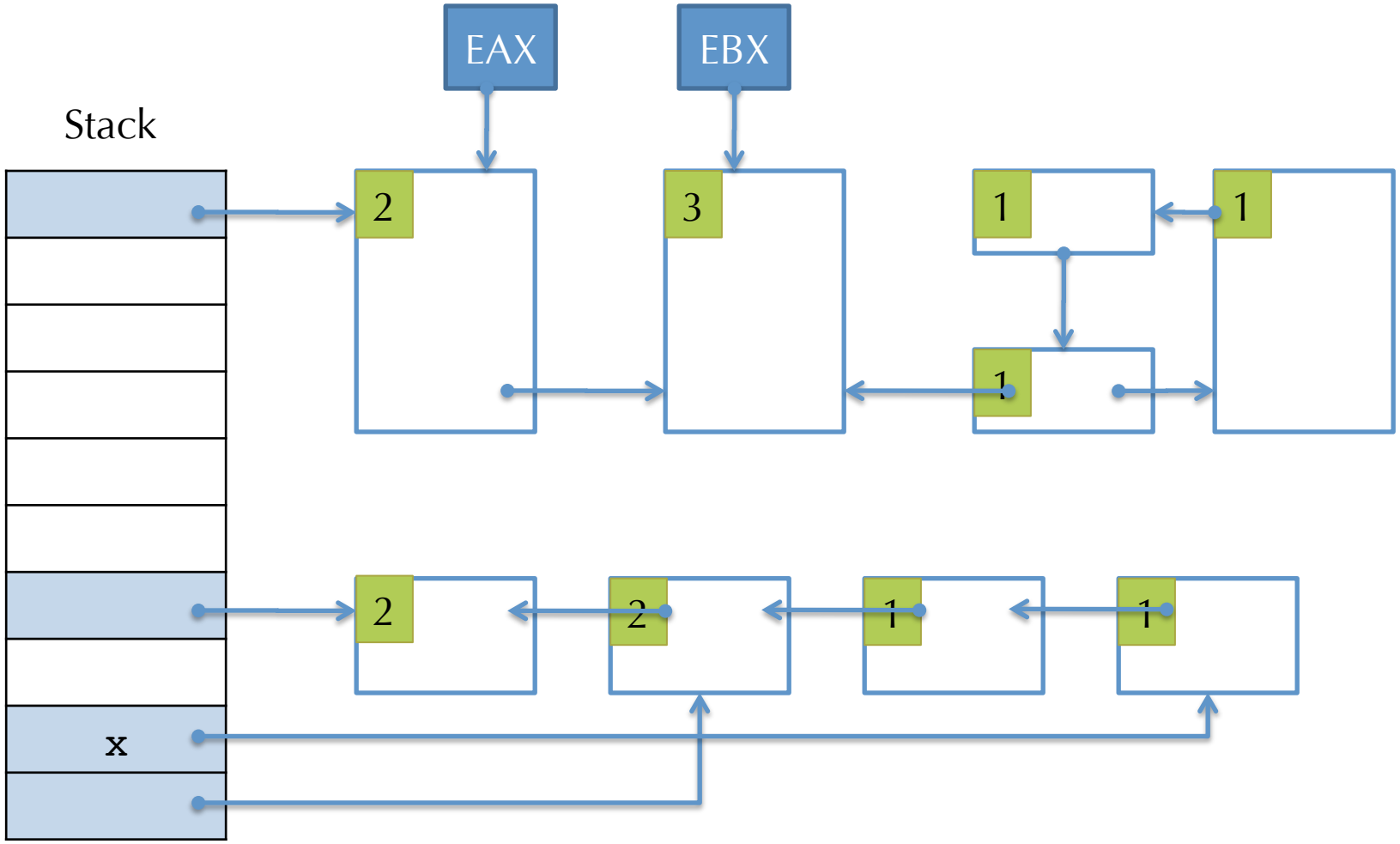
- Idea: Keep track of the number of references to a given object.
  - When creating a new reference to the object, increase the reference count
  - On a call to `free`, decrement the reference count
  - If the reference count is 0, the object can be deallocated immediately

- Deallocating an object will decrement reference counts of objects it points to
  - Deallocations can "cascade," causing lots of objects to be deallocated

- Benefit: immediate reclamation of the space (no need to wait for garbage collector)
- Challenges:
  - Tracking reference counts efficiently
  - Cyclic data structures

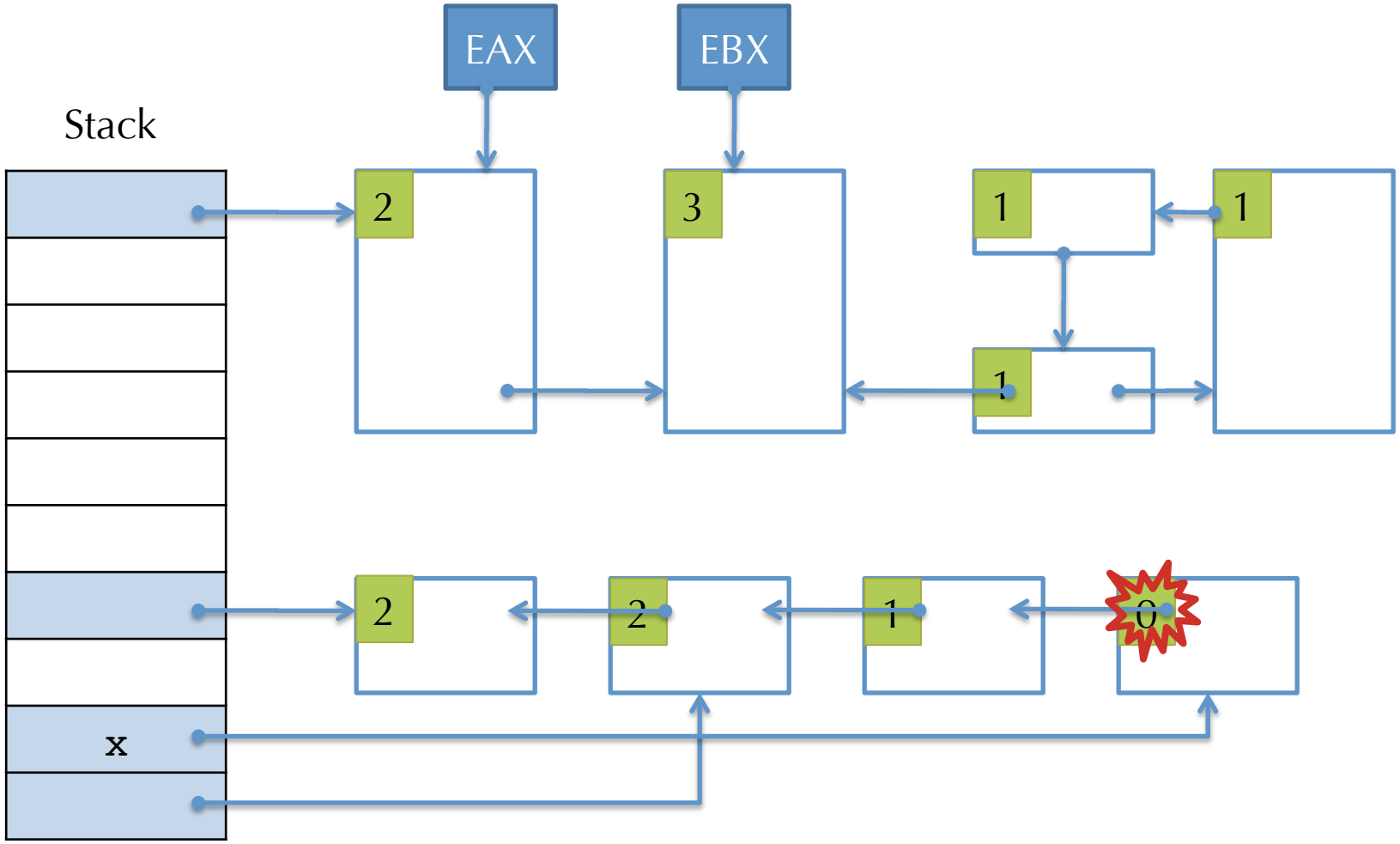
Example Reference Counts

- Objects track reference counts.



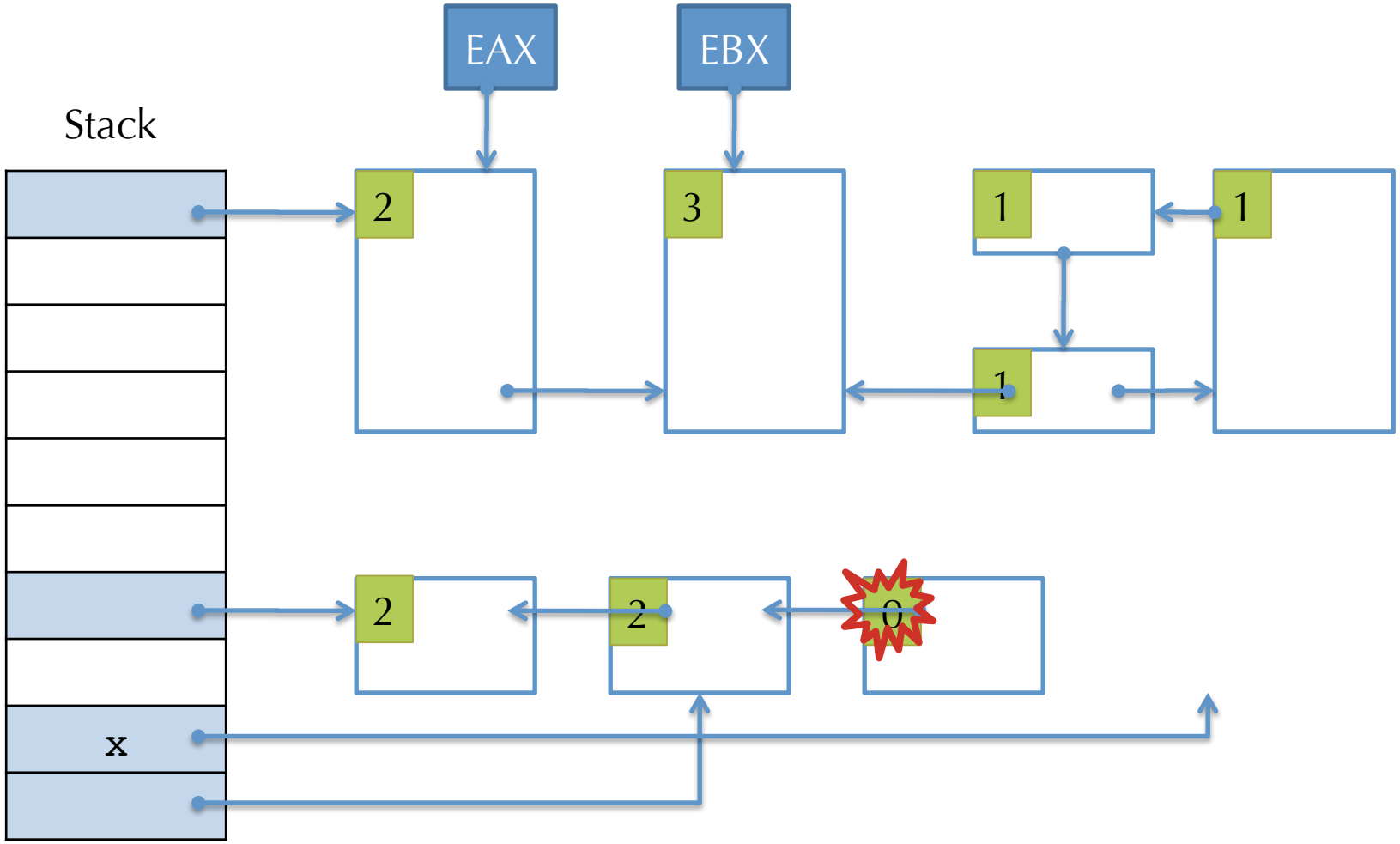
Example Reference Counts

- On `free(x)`



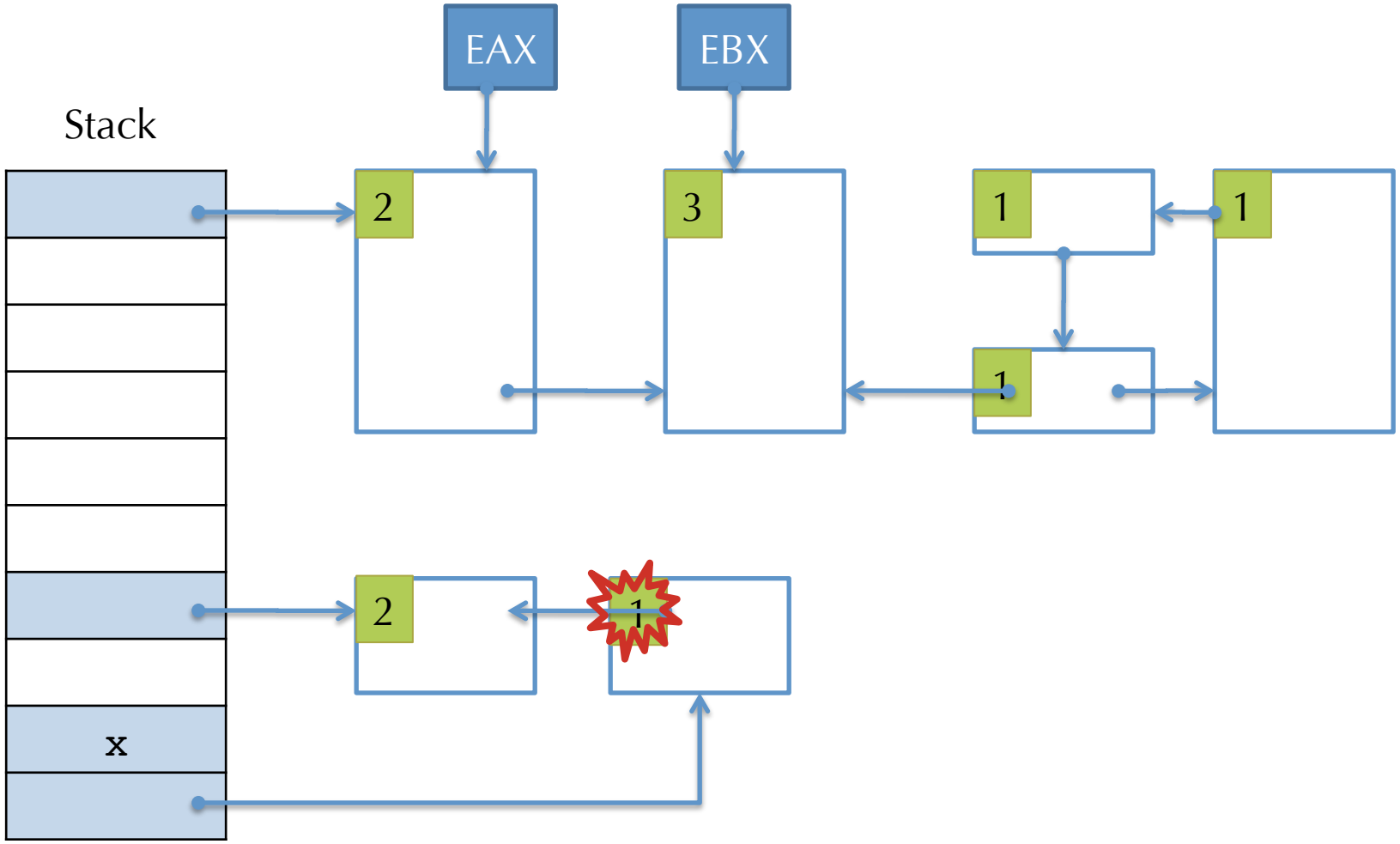
Example Reference Counts

- On `free(x)`

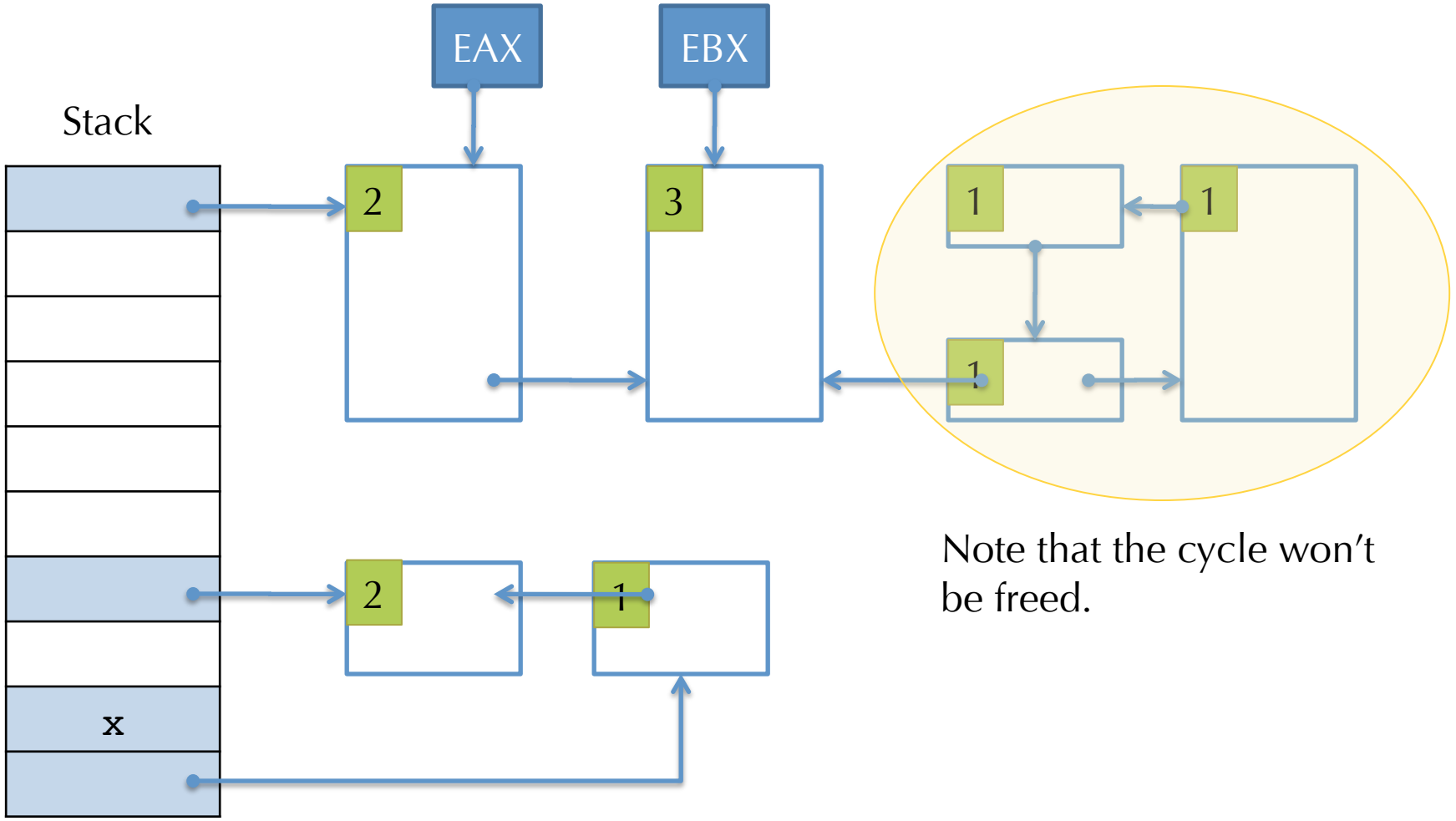


Example Reference Counts

- On `free(x)`



Example Reference Counts



Note that the cycle won't be freed.

Dealing with Cycles

- Option 1: Require programmers to explicitly null-out references to break cycles.

- Option 2: Periodically run GC to collect cycles

- Option 3: Require programmers to distinguish "weak pointers" from "strong pointers"
  - *weak pointers*: if all references to an object are "weak" then the object can be freed even with non-zero reference count.
  - "Back edges" in the object graph should be designated as weak
  - (Aside: weak pointers useful in GC settings too.)

- In practice:
  - Apples Cocoa framework used ref counts, recent versions use GC
  - iOS supports "automatic reference counting"

OAT PROGRAMMING

Oat Programming

- Oat idioms
  - see lib/list.oat, tests/encroach.oat available in Project 7
- Interfacing with C
  - see lib/console.*
- What's missing?
  - Generics/parametric polymorphism
  - Exceptions
  - Concurrency
  - Encapsulation (e.g. `private` keyword)
- Better language design?
  - e.g. combine `cast` and `if?`

FINAL EXAM

Final Exam

- Will cover material since the midterm almost exclusively
  - Starting from Lecture 14 (First-class Functions)
  - Objects, inheritance, types, implementation of dynamic dispatch
  - Basic optimizations
  - Dataflow analysis (forward vs. backward, fixpoint computations, etc.)
    - Liveness
  - Control flow analysis
  - SSA
  - Graph-coloring Register Allocation

- Will focus more on the theory side of things
- Format will be similar to the midterm
  - Simple answer, computation, multiple choice, etc.
  - Sample exam from last time is on the web

What have we learned?
Where else is it applicable?
What next?

COURSE WRAP-UP

Why CIS 341?

- You will learn:
  - Practical applications of theory
  - Parsing
  - How high-level languages are implemented in machine language
  - (A subset of) Intel x86 architecture
  - A deeper understanding of code
  - A little about programming language semantics
  - Functional programming in OCaml
  - How to manipulate complex data structures
  - How to be a better programmer

- Did we meet these goals?

Stuff we didn't Cover

- We skipped stuff at every level…
- Concrete syntax/parsing:
  - Much more to the theory of parsing…
  - Good syntax is art not science!
- Source language features:
  - Exceptions, recursive data types (easy!), advanced type systems, type inference, concurrency
- Intermediate languages:
  - Intermediate language design, bytecode, bytecode interpreters, just-in-time compilation (JIT)
- Compilation:
  - Continuation-passing transformation, efficient representations, scalability
- Optimization:
  - Scientific computing, cache optimization, instruction selection/ optimization

Course Work

- 72% Projects: *The Quaker OAT Compiler*

- 12% Midterm
- 16% Final exam

- Expect this to be a challenging, implementation-oriented course.

I think we met this goal…

Related Courses: Fall 2013

- CIS 500: Software Foundations
  - I will be teaching it
  - Theoretical course about functional programming, proving program properties, type systems, lambda calculus. Uses the theorem prover Coq.
- CIS 501: Computer Architecture
  - Dr. Devietti
  - 371++: pipelining, caches, VM, superscalar, multicore,…
- CIS 552: Advanced Programming
  - Dr. Weirich
  - Advanced functional programming in Haskell, including generic programming, metaprogramming, embedded languages, cool tricks with fancy type systems

Where to go from here?

- Conferences (proceedings available on the web):
  - Programming Language Design and Implementation (PLDI)
  - Principles of Programming Langugaes (POPL)
  - Object Oriented Programming Systems, Languages & Applications (OOPSLA)
  - International Conference on Functional Programming (ICFP)
  - European Symposium on Programming (ESOP)
  - …

- Technologies
  - Yacc, lex, bison, flex, …
  - LLVM – low level virtual machine
  - Java virtual machine (JVM), Microsoft's Common Language Runtime (CLR)
  - Languages: OCaml, F#, Haskell, Scala, Go, Rust, …?

Where else is this stuff applicable?

- General programming
  - In C/C++, better understanding of how the compiler works can help you generate better code.
  - Ability to read assembly output from compiler
  - Experience with functional programming can give you different ways to think about how to solve a problem
- Writing domain specific languages
  - lex/yacc very useful for little utilities
  - understanding abstract syntax and interpretation
- Understanding hardware/software interface
  - Different devices have different instruction sets, programming models

Thanks!

- To the TAs: Dmitri & Bob
  - for doing an amazing job putting together the projects for the course.

- To *you* for taking the class!

- How can I improve the course?
  - Better feedback to students during projects
    (i.e. get the grading done sooner!)

  - Revisit projects to improve clarity and better