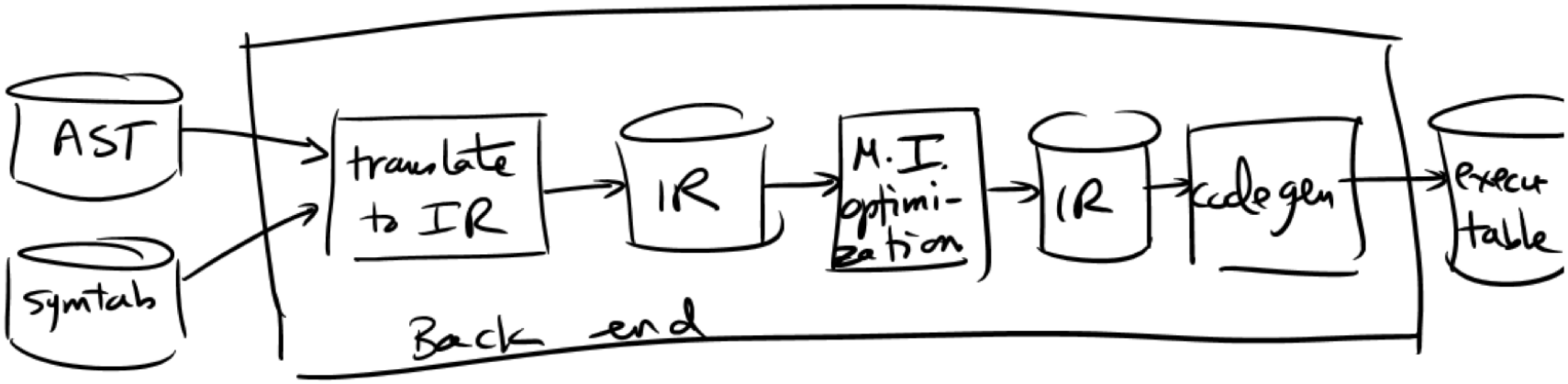
Lecture 13 — Garbage collection, and miscellaneous

- Modern "dynamic" languages — including not only scripting languages, but also Java, OCaml, etc. — rely on automatic memory management. Knowing the basic methods of doing this is important for implementing these languages.
  - Automatic memory management
    - Non-reachable heap nodes
    - Reference-counting
    - Garbage collection via mark-and-sweep
    - Garbage collection via stop-and-copy
  - Start with a couple of miscellaneous topics...

Compiler structure

- **Compilers don't usually translate directly from AST to native machine code.**



- **IR = "intermediate representation" (often a simplified machine language)**
- **MI = "machine-independent"**
- **Code gen. incorporates machine-dependent optimization**

Machine-independent optimizations

- **Reducing total number of operations**
- **Reducing amount of work in loops**

⇒ **Optimizations that are likely to increase speed on *any* machine.**

Source

```
int A[100];

while (j<n) {
   x = x + A[i];
   j++;
}
```

Initial IR

```
L1: r1 = &A
    r2 = i*4
    r3 = r1+r2
    r4 = LOADIND r3
    x = x+r4
    j = j+1
    CMP j,n
    JMPIFLESS L1
```

Optimized IR

```
    r1 = &A
    r2 = i*4
    r3 = r1+r2
    r4 = LOADIND r3
L1: x = x+r4
    j = j+1
    CMP j,n
    JMPIFLESS L1
```

Compilation targets: Real vs. abstract machines

- *Traditional*: Compile to target machine code
- *Java/C#*: Compile to virtual (i.e. fake) machine code (JVM or CLR/CIL)
  - Execute by interpreting that machine, *or*
  - Translate to native machine code *at run time* (called "just-in-time compilation")
  - Advantages: portability, security
  - Disadvantage: Non-optimal performance (mainly because optimization process is time-constrained)
- "Virtual machine code" is also called *bytecode* or sometimes *bitcode*.

Machine-dependent optimization

- When translating to native machine code — either at initial compilation time or at "just-in-time" compilation time — want to generate most efficient code.
- Machine-dependent optimization = optimizations that exploit features of target machine such as registers, pipeline, special instructions
  - Register allocation
  - Instruction selection
  - Instruction scheduling

Interpreters vs. compilers

- Traditionally, languages where programs are normally executed interactively — i.e. without producing an explicit compiled version (Python, Javascript, OCaml, etc.) are said to be "interpreted."
- However, this is different from our use of the term "interpret," which refers to a method of executing programs (the method we use in MP6 and 7).
- In fact, we cannot know what method of execution these languages use without looking inside the "interpreter."
- In practice, all such languages compile to executable form — usually a virtual machine — internally, because pure interpreted execution is inefficient.

Dynamic languages vs. static languages

- Dynamic: Python, Perl, Ruby, JavaScript
- Static: C, C++, Fortran
- Intermediate: Java, C#, OCaml
- Two primary distinguishing features:
  - Dynamic type-checking; tagged values
  - Automatic memory management

Automatic memory management

- **Consider this code in OCaml:**

```
let rec append x y = if x=[] then y else hd x :: append (tl x) y
let rec rev l = if l=[] then [] else append (rev (tl l)) @ [hd l]
```

- **Suppose `lis` is a list of length 10. When `rev lis` is called, _____ cells of garbage are created.**

- **Without a way to reuse these cells, programs would quickly run out of memory.**

- **Similar examples can be constructed in Java, Python, or any other modern language. These languages would be unusable without *automatic memory management*.**

Reachability (aka accessibility) of heap cells

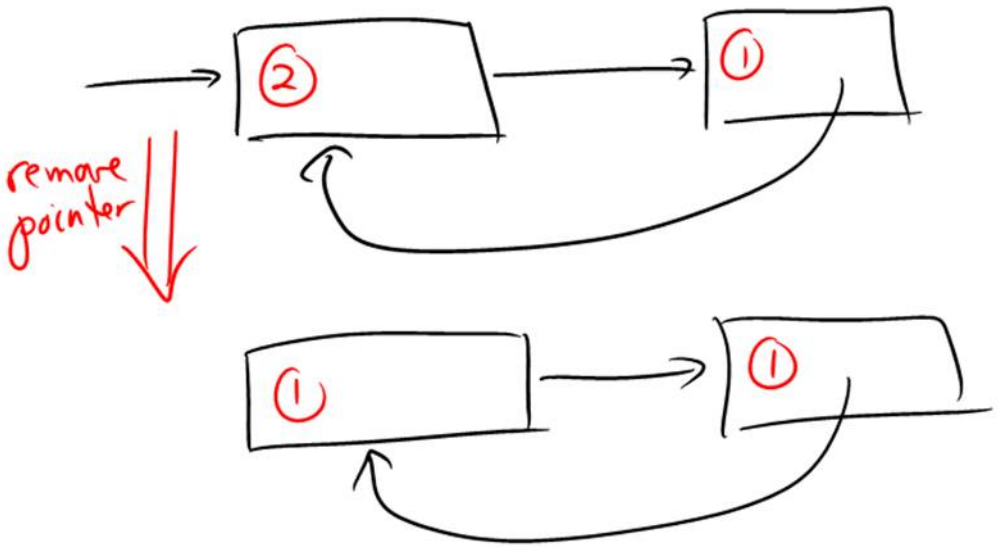
- **Data in state is accessible only through variables on stack, so heap objects are accessible only through pointers on stack.**
- **Think of heap as a directed graph, with entry points from the stack.**
- **The only useful data in heap is what is accessible, directly or indirectly, from those entry points. Other data is called *garbage*. Garbage nodes can never affect computation, because they can never be seen by the program.**
- ***Automatic memory management attempts to make garbage cells available for allocation.***

Reference-counting

- Keep free memory areas on a list
- Track number of pointers to every object — every object has an additional field giving count of in-pointers.
- Adjust count each time a pointer is copied/assigned
  - `p = q`:
    - Increment refcnt(*q)
    - Decrement refcnt(*p)
    - if refcnt(*p)=0 then return memory of p to free list *and* decrement refcnt of all objects that *p points to

Reference-counting (cont.)

- **Advantage: Recover memory of an object as soon as the object becomes non-reachable**
- **Big disadvantage: Cannot handle cycles in the heap:**



Garbage collection

- Don't keep track of reachability continually — instead, wait until memory runs out, then run a program to find all the non-reachable objects and recover them - a *garbage collector*.
- Two basic methods (with many variations and combinations):
  - Mark-and-sweep
  - Stop-and-copy
- Advantage: Handles cyclic structures easily
- Disadvantage: Creates a pause in the calculation while g.c. algorithm runs

Mark-and-sweep

- Reserve one bit in each object header, called the *reachable bit*
- Start with reachable bit zero in every header
- Traverse reachable data (depth-first search), setting reachable bit
- Sweep over entire heap. For each object, if reachable bit is 1, reset it; if it is zero, place that memory chunk on free list.
- Observations:
  - Reachable data is not moved
  - Reachable data remains spread across memory
  - Cost is linear in total size of heap

Stop-and-copy

- Only half of memory available for the heap is used at any time; that half is called *half-in-use*, the other half *reserved*.
- Half-in-use is divided into used area and free area.
- Allocate memory from top of used area (bottom of free area). When free area is exhausted, do g.c.
- Garbage collection method:
  - Traverse *reachable* objects, moving each object encountered to reserved area, allocating sequentially from bottom.
  - Complicated part is adjusting pointers.
  - Reserved area now becomes half-in-use.
- Advantage: Cost proportion to amount of reachable data.

Wrap-up

- **Today we discussed:**
  - Compilation to native code or VM
  - Compiler optimizations
  - Automatic memory management
- **We discussed it because:**
  - Needed to understand how different compilers and run-time systems work.
- **What to do now:**
  - MP7
  - You will not be asked to implement garbage collection.