Lifetime Management

We know about many examples of *heap-allocated* objects:

- Created by `new` or `malloc` in C++
- All `Objects` in Java
- *Everything* in languages with lexical frames (e.g. Scheme)

When and how should the memory for these be reclaimed?

Manual Garbage Collection

In some languages (e.g. C/C++), the programmer must specify when memory is de-allocated.

This is the *fastest* option, and can make use of the programmer's expert knowledge.

Dangers:

Automatic Garbage Collection

Goal: Reclaim memory for objects that are no longer *reachable*.

Reachability can be either direct or indirect:

- A name that is *in scope* refers to the object (direct)
- An data structure that is reachable contains a reference to the object (indirect)

Should we be conservative or agressive?

Reference Counting

Each object contains an integer indicating
how many references there are to it.

This count is updated continuously as the program proceeds.

When the count falls to zero, the memory for the object is deallocated.

Analysis of Reference Counting

This approach is used in filesystems and a few languages
(notably PHP and Python).

Advantages:

- Memory is freed as soon as possible.
- Program execution never needs to halt.
- Over-counting is wasteful but not catastrophic.

Disadvantages:

- Additional code to run every time references are created, destroyed, moved, copied, etc.
- *Cycles* present a major challenge.

Mark and Sweep

Garbage collection is performed periodically, usually halting the program.

During garbage collection, the entire set of reachable references is
traversed, starting from the names currently in scope.

Each object is *marked* when it is seen.
After the traversal, any unmarked objects are deallocated.

Example

```
(define (f x y)
  (lambda (z)
    (if (<= (abs (- x z)) (abs (- y z)))
        x
        y)))

(define posneg (f -1 1))

(display (posneg 21))
(display (posneg -3))
(display ((f 10 20) 18))
```


Analysis of Mark and Sweep

This is the most common GC technique in programming languages.

Advantages:

- Very aggressive strategy; there will be no un-referenced objects left in memory.
- Does not slow down normal execution. No effect whatsoever on programs with a small memory footprint.

Disadvantages:

- Potential to halt execution unpredictably — not suitable for real-time systems.
- *Undercounting* will cause dangling references.
- Deallocation is always delayed.

GC Tricks and Tweaks

- **Generational garbage collection**: Takes advantage of the observation that newer objects are more likely to be garbage.
- **Stop and Copy**: Like mark-and-sweep, but instead of marking reachable objects, copy them to another part of memory. Then free all of the old memory space at once.
- **Weak References**: Allow programmer to specify that some references should not keep an object alive by themselves. Example: keys in a hash table
- **Conservative GC**: Assume every integer is a pointer.
- Reference counting with **delayed cycle detection**
- **Incremental** mark and sweep

Whose responsibility?

Automatic GC takes responsibility from the programmer
and puts it on the language implementor.

Where should GC happen?

- In the interpreter
- In the compiled code
- In a *virtual machine*

Virtual Machines

- An alternative to interpreted vs. compiled

Source code is compiled to *byte code*, which runs on a virtual machine.

The VM is a program which is written once for each platform.

Advantages:

Class outcomes

You should know:

- Manual vs. Automatic Garbage Collection
- Reference counts vs. Mark-and-sweep
- What *reachability* is, and how it is determined
- The shortcomings of different methods for automatic garbage collection, and how they are mitigated in practice.
- How VMs provide a compromise between compiled and interpreted languages.

You should be able to:

- Show the reference counts of objects in a program trace.
- Perform a mark-and-sweep operation at any point during the execution of an example program.

LLVM IR

LLVM is a tool to build optimizing compilers, used in many compilers for languages you know:

- Clang (C, C++)
- Rust
- Swift
- Haskell (optional backend for ghc)
- Julia
- Kotlin
- ... **and SPL** (your last lab!)

LLVM uses an **Intermediate Representation** (LLVM IR).
This is the *input* to LLVM, which then produces actual machine code.

3AC/TAC

LLVM IR is like a simplified assembly language.

Example instruction:

```
%x17 = mul i32 %x5, %x6
```

Called a **three-address code** (3AC or TAC) because each operation has (at most) one destination and two arguments.

SSA

Names like `%x17` are called *registers* in LLVM IR.

Each register **can only be assigned once** and never changed.
This is called **Single Static Assignment** (SSA).

In practice: pick a new register name as the destination of each instruction.

TAC SSA practice

Translate to LLVM IR:

```
int x, y; // assume these values are already set somewhere
int z = y - 3*x + 4;
```


Control flow

We need loops and if statements!
LLVM IR code is divided into **basic blocks**.

- Each basic block starts with a label, like `MY_LABEL:`
- Each basic block ends with a branch or return statement.
  - Unconditional branch: `br label %SOME_LABEL`
  - Conditional branch (on boolean value %x7): `br i1 %x7, label %IF_LABEL, label %ELSE_LABEL`
  - Return (this example returns 0 from main): `ret i32 0`

SSA with branches

Try to write LLVM IR instructions for a while loop, like

```
int x = 0;
while (x < 10) {
  x += 3;
}
```

What is the **inherent problem** with SSA and control flow?

Code generation steps

What happens to turn LLVM IR into machine code?

Register allocation vocab

Virtual register: Registers in LLVM IR

Live range: Lines where (virtual) register value must not be disturbed

Physical register: Actual CPU register

Spill: Copy register to memory

Fill: Copy memory back to register

Register allocation algorithms

- 1981: Greg Chaitin, graph coloring
- 2005, 2006: Brisk; Perriera & Palsberg; Hack et al
  Specific algorithm for **SSA** programs
- Remaining challenge: