Homework 4

Benjamin Jakubowski

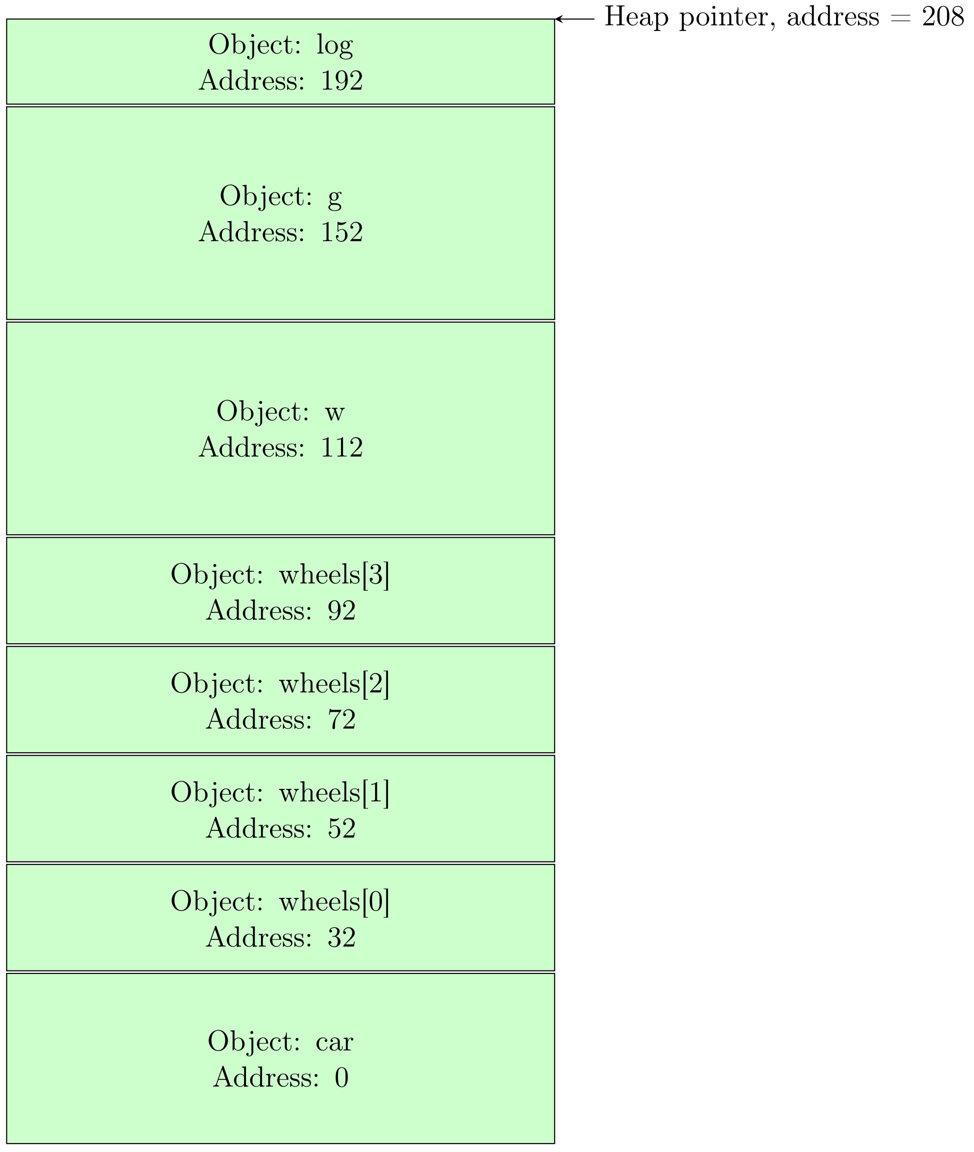
July 29, 2016

1. Garbage Collection

a. Heap pointer allocation method

Assume the 'heap pointer' allocation method is used to allocate memory. We draw the contents of the heap upon entry into `BeginEventLoop`, but before any garbage collection has taken place, identifying and showing the address of each object in the heap. In addition, we show the address of the heap pointer at that moment (assuming heap memory begins at address 0 and that the heap pointer is incremented by the size of the allocated object during each allocation.) Note each object takes:

- **Car**: 24 bytes + 8 byte forwarding address = 32 bytes
- **Wheel**: 12 bytes + 8 byte forwarding address = 20 bytes
- **Carwash**: 32 bytes + 8 byte forwarding address = 40 bytes
- **Garage**: 32 bytes + 8 byte forwarding address = 40 bytes
- **EventLog**: 8 bytes + 8 byte forwarding address = 16 bytes


Heap pointer, address = 208
Object: log
Address: 192
Object: g
Address: 152
Object: w
Address: 112
Object: wheels[3]
Address: 92
Object: wheels[2]
Address: 72
Object: wheels[1]
Address: 52
Object: wheels[0]
Address: 32
Object: car
Address: 0

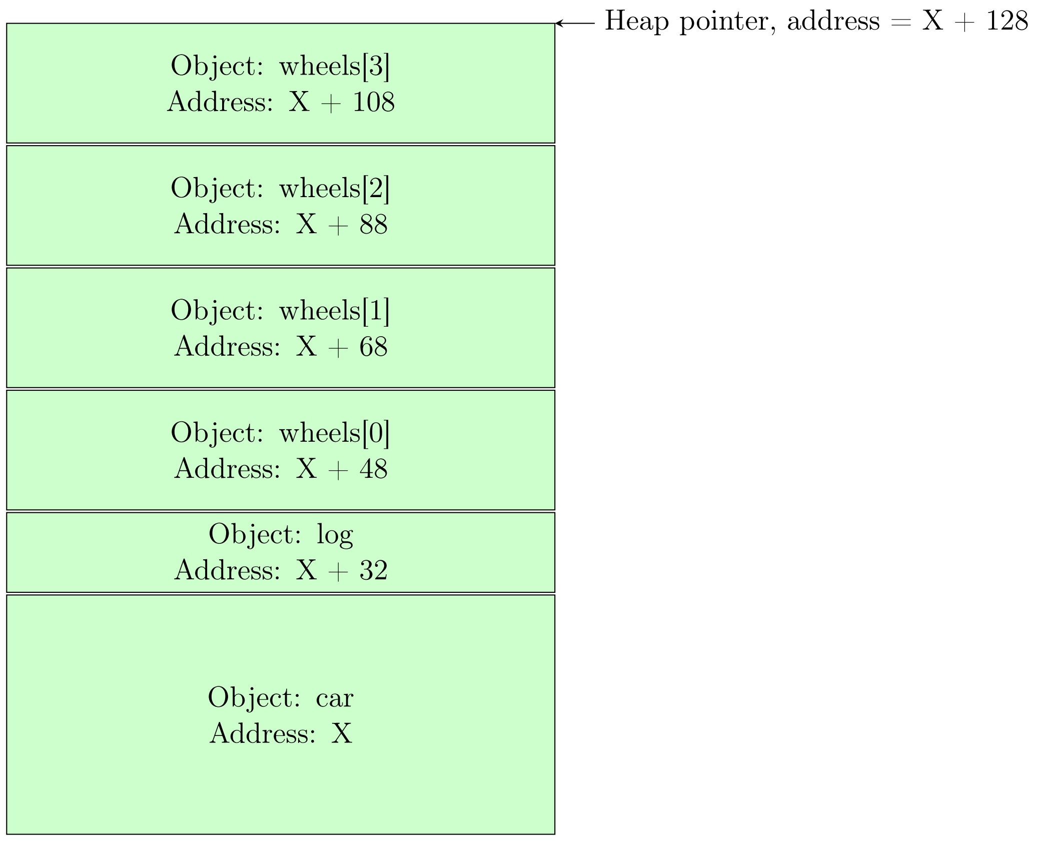
B. Copy collection during `BeginEventLoop`

Now assume the copy garbage collection algorithm discussed in class executes during the call to BeginEventLoop. We show the state of heap memory after the first collection has concluded. Note we only show the TO space, not the FROM space (which is shown above). Additionally, we assume the TO space memory addresses begin at X.

In addition, we identify the root pointers and specify the order in which they were processed by the copy collection algorithm to arrive at the heap state identified. These root pointers, and their processing order are enumerated below:

1. We start with local variables in the `Main` stack frame, finding `car` and `log`. We copy these two objects to the TO space.
2. Next, the found `Car` object has pointers to the four `Wheel` objects (in the `wheels` list). Thus we copy these objects to the TO space.
3. The objects `w` and `g` were local to `Initialize(car)`. This function has already resolved when `BeginEventLoop` is called; hence `w` and `g` are dead. Thus they are not copied to TO space.

The final state of TO space memory is shown in the diagram on the next page.


Heap pointer, address = X + 128
Object: wheels[3]
Address: X + 108
Object: wheels[2]
Address: X + 88
Object: wheels[1]
Address: X + 68
Object: wheels[0]
Address: X + 48
Object: log
Address: X + 32
Object: car
Address: X

2. Prolog

See attached .pl knowledge base and queries.

3. Prolog Performance

Consider the following:

```
foo(ashwin).
foo(roberta).

hello(brock).
hello(roberta).
hello(john).

world(ashwin).
world(roberta).

goal(X) :- sub1(X),sub2(X).
sub1(X) :- foo(X).
sub2(X) :- hello(X),world(X).
```

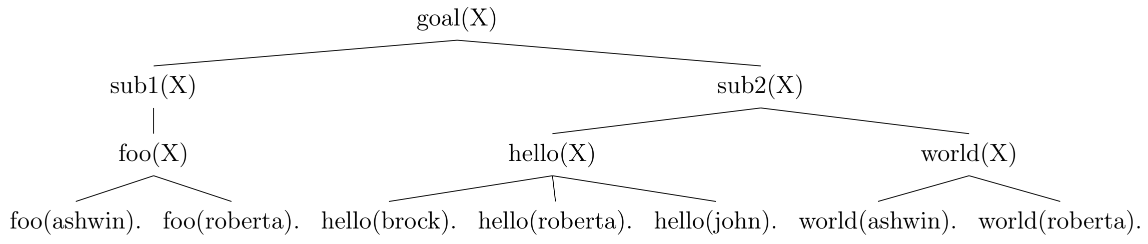
a. Faster query time for `goal(X)`

We can provide faster execution time when querying `goal(X)` by reordering the facts as follows:

```
foo(roberta).
foo(ashwin).


hello(roberta).
hello(brock).
hello(john).

world(roberta).
world(ashwin).

goal(X) :- sub1(X),sub2(X).
sub1(X) :- foo(X).
sub2(X) :- hello(X),world(X).
```

B. Explaining why this ordering improves execution time

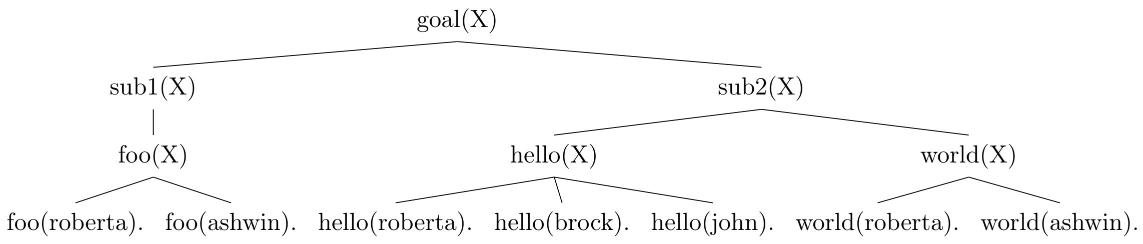
Reordering the facts improves the execution time because it avoids backtracking. Under the original ordering, our knowledge base forms the following tree:



When we query `goal(X).`, we first traverse the tree down to `foo(ashwin)`. Thus, we initially unify X and ashwin. This proceeds until `hello(ashwin)` fails, and we have to backtrack all the way to `foo(X)`, where we next try unifying X and roberta. This succeeds, but we wasted time exploring the tree with X bound to ashwin.

```
[trace]  ?- goal(X).
   Call: (7) goal(_G2397) ? creep
   Call: (8) sub1(_G2397) ? creep
   Call: (9) foo(_G2397) ? creep
   Exit: (9) foo(ashwin) ? creep
   Exit: (8) sub1(ashwin) ? creep
   Call: (8) sub2(ashwin) ? creep
   Call: (9) hello(ashwin) ? creep
   Fail: (9) hello(ashwin) ? creep
   Fail: (8) sub2(ashwin) ? creep
   Redo: (9) foo(_G2397) ? creep
   Exit: (9) foo(roberta) ? creep
   Exit: (8) sub1(roberta) ? creep
   Call: (8) sub2(roberta) ? creep
   Call: (9) hello(roberta) ? creep
   Exit: (9) hello(roberta) ? creep
   Call: (9) world(roberta) ? creep
   Exit: (9) world(roberta) ? creep
   Exit: (8) sub2(roberta) ? creep
   Exit: (7) goal(roberta) ? creep
X = roberta.
```

In contrast, after reordering the facts, we have the following tree:



When we query `goal(X).`, we first traverse the tree down to `foo(roberta)`. Thus, we initially unify X and roberta, and avoid having to backtrack (as this unification instantiates all the subgoals).

```
[trace]  ?- goal(X).
   Call: (7) goal(_G2397) ? creep
   Call: (8) sub1(_G2397) ? creep
   Call: (9) foo(_G2397) ? creep
   Exit: (9) foo(roberta) ? creep
   Exit: (8) sub1(roberta) ? creep
   Call: (8) sub2(roberta) ? creep
   Call: (9) hello(roberta) ? creep
   Exit: (9) hello(roberta) ? creep
   Call: (9) world(roberta) ? creep
   Exit: (9) world(roberta) ? creep
   Exit: (8) sub2(roberta) ? creep
   Exit: (7) goal(roberta) ? creep
X = roberta .
```

c. Adding break `sub1(X) :- foo(X), !`

Now, going back to the original ordering of facts, now suppose we rewrite goal `sub1` to read: `sub1(X) :- foo(X), !`. Upon returning to query mode and querying `goal(X)`, the interpreter displays false.

This occurs because under the original ordering of facts, X is first bound to ashwin. Since `foo(ashwin)` is `true`, `sub1(X) :- foo(X), !` proceeds to the break "!", which prevents backtracking (i.e. binding X to roberta, such that `goal(X)` is true).
We can see this in the trace:

```
[trace]  ?- goal(X).
   Call: (7) goal(_G2397) ? creep
   Call: (8) sub1(_G2397) ? creep
   Call: (9) foo(_G2397) ? creep
   Exit: (9) foo(ashwin) ? creep
   Exit: (8) sub1(ashwin) ? creep
   Call: (8) sub2(ashwin) ? creep
   Call: (9) hello(ashwin) ? creep
```

```
    Fail: (9) hello(ashwin) ? creep
    Fail: (8) sub2(ashwin) ? creep
    Fail: (7) goal(_G2397) ? creep
false.
```

D. Adding break `sub2(X) :- hello(X), !, world(X).`

Now, going back to the original ordering of facts, now suppose we rewrite goal `sub2` to read: `sub2(X) :- hello(X), !, world(X).`. Upon returning to query mode and querying `goal(X)`, this time the goal succeeds.

This is because the break is **not** reached until X is bound to roberta. Initially, X is bound to ashwin. However, when we reach `sub2`, `hello(ashwin)` fails and ! is not reached. Instead, we backtrack all the way to the subgoal `sub1(X) :- foo(X)`, and bind X to roberta. Thus, the break is not reached until X is bound to roberta, which is fine since world(roberta) is true.
To see this more clearly, we show the trace for a modified version of this rule- the modified version is:
`sub2(X) :- hello(X), write("Before break"), !, write("After break"), world(X).`

The trace for this modified version is shown below.

```
[trace]  ?- goal(X).
    Call: (7) goal(_G2397) ? creep
    Call: (8) sub1(_G2397) ? creep
    Call: (9) foo(_G2397) ? creep
    Exit: (9) foo(ashwin) ? creep
    Exit: (8) sub1(ashwin) ? creep
    Call: (8) sub2(ashwin) ? creep
    Call: (9) hello(ashwin) ? creep
    Fail: (9) hello(ashwin) ? creep
    Fail: (8) sub2(ashwin) ? creep
    Redo: (9) foo(_G2397) ? creep
    Exit: (9) foo(roberta) ? creep
    Exit: (8) sub1(roberta) ? creep
    Call: (8) sub2(roberta) ? creep
    Call: (9) hello(roberta) ? creep
    Exit: (9) hello(roberta) ? creep
    Call: (9) write("Before break") ? creep
Before break
    Exit: (9) write("Before break") ? creep
    Call: (9) write("After break") ? creep
After break
    Exit: (9) write("After break") ? creep
    Call: (9) world(roberta) ? creep
    Exit: (9) world(roberta) ? creep
    Exit: (8) sub2(roberta) ? creep
    Exit: (7) goal(roberta) ? creep
X = roberta.
```

Clearly we don't reach the break (which is bound by `write("Before break")` and `write("After break")`) until we've rebound X to roberta.

4. Unification

a. d(15) & c(X)

This pair does not unify, since the constants d and c differ. and constants only unify with themselves.

b. a(X, b(3, 1, Y)) & a(4, Y)

This pair does not unify, since the second argument to the functor a would need to unify, but Y cannot unify with b(3,1, y), since this yields an infinite recursion.

c. a(X, c(2, B, D)) & a(4, c(A, 7, C))

This pair does unify, with the following bindings:

- X = 4,
- A = 2,
- B = 8
- C = D

d. a(X, c(2, A, X)) & a(4, c(A, 7, C))

This pair does not unify, since we would obtain the following contradictory bindings:

- X = 4,
- A = 2 and 7 (**contradiction**!),
- C = 4,

e. e(c(2,D)) & e(c(8, D))

This pair does not unify, since the constants 2 and 8 differ, and constants only unify with themselves.

F. X & E(F(6,2), G(8, 1))

This pair does unify, with the binding

- X = e(f(6,2), g(8, 1))

G. B(X, G(8, X)) & B(F(6,2), G(8, F(6,2)))

This pair does unify, with the binding

- X = f(6,2)

H. A(1, B(X, Y)) & A(Y, B(2, C(6, Z), 10))

This pair does not unify for a number of reasons, primarily because b/2 and b/3 can't unify since they differ in arity.

I. D(C(1,2,1)) & D(C(X, Y, X))

This pair does unify, with the bindings

- X = 1
- Y = 2