Gerenciamento de Memória

Em linguangens como **C**, a memória é solicitada e liberada explicitamente, com *malloc()* e *free()*. Mesmo assim, variáveis locais temporárias são alocadas dinamicamente e liberadas ao se deixar o escopo.
Por outro lado, **Java** e **Perl** sempre liberam a memória que não é mais utilizada por conta própria.

De quem é a responsabilidade de liberar memória?

O grande problema é que o uso das informações depende da semântica, que o compilador não consegue inferir.

Entretando, colocar liberação de memória explicitamente atrapalha a descrição do programa (baixo nível $\times$ alto nível)

Sutilezas

Um filtro de números positivos.

```
(define (filter-pos l)
        (cond
              [(empty? l) empty]
              [else
                   (begin
                         (reclaim-memory! (first l))
                         (if (> (first l) 0)
                             (cons (first l) (filter-pos (rest l)))
                             (filter-pos (rest l))))]))
```

Recursão de cauda, mas temos um problema!

Além disto faltou liberar a segunda parte da lista: (rest l)

Arrumando

```
(define (filter-pos l)
        (cond
              [(empty? l) empty]
              [else
                   (local([define result
                             (if (> (first l) 0)
                                 (cons (first l) (filter-pos (rest l)))
                                 (filter-pos (restl)))])
                             (begin
                                  (reclaim-memory! (first l)) (reclaim-memory! (rest l))
                                  result))])))
```

Foi-se a recursão de cauda!!!

Problemas gerais

Mesmo sabendo quando é possível fazer a coleta, existem alguns problemas relativamente comuns:

- A estrutura do programa pode mudar significativamente
- Erros de liberação são sutis, especialmente em sistemas concorrentes (e *multicore*!)
- Laços podem perder recursão de cauda
- Fica mais difícil fazer abstrações (pode existir chamadas que não querem que a função libere memória). Aumento exponencial no número de argumentos.
- É preciso definir quem é reponsavel por cada dado, o que nem sempre é natural

O sistema de execução faz a tarefa: **Gerenciamanto Automático de Memória** ou **Coleta De lixo (*Garbage Collection*)**

Certeza absoluta ou relativa???

O coletor de lixo deve saber quando um dado irá ser usado novamente ou não: inferência, que leva a inteligência artificial.

O coletor de lixo libera a memória dos dados que não são mais necessários.Quais são eles?? Como saber???
Qual a diferença entre o verdadeiro e o demonstrável?

O que se demonstra é verdade, mas nem toda verdade é demonstrável (dá-lhe Gödel).

O sistema de execução não tem como saber das intenções do programador, portanto tem tem como sempre ter certeza do que pode ser liberado.

Um coletor de lixo que nunca libera nada é muito rápido, mas inútil.

Um outro que apaga toda a memória também é rápido, mas desastroso.

Características

O que deve ser levado em consideração quando se constrói um coletor de lixo?

- Utilidade — deve liberar quantidades úteis de memória
- Robustez — nunca apagar algo que ainda seria usado
- Eficiência — deve ser rápido quando comparado com a execução do programa.

Aproximação da verdade: *atingibilidade*.

Parte-se de um conjunto de dados que se tem certeza do uso, o *root set* e se inclui outros dados referenciados.

Problemas

```
(define v (make-vector 1000))
(define k (vector-length v))
⋮
```

v nunca será liberado, mesmo que não seja referenciado no resto do programa, pois é global e faz parte do *root-set*.

Outro problema: referências cíclicas.

Por quê existe *garbage collection* em **Java** e **Scheme**, mas não existe em **C** e **C++**?

Estratégias

1. Contagem de referências
2. *Mark and Sweep*