Universidade da Beira Interior

Desenho de Linguagens de Programação e de Compiladores

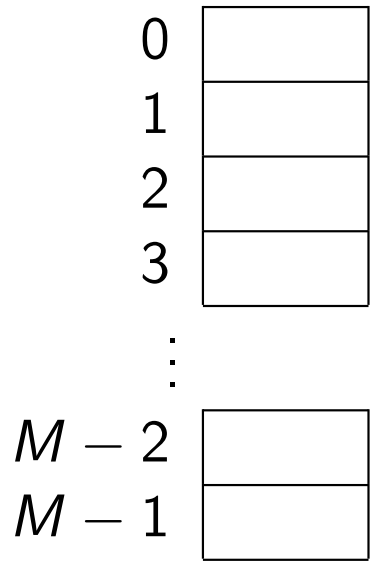
Simão Melo de Sousa

Aula 10 - Alocação de memória

a memória física de um computador é um grande vector de M bytes,

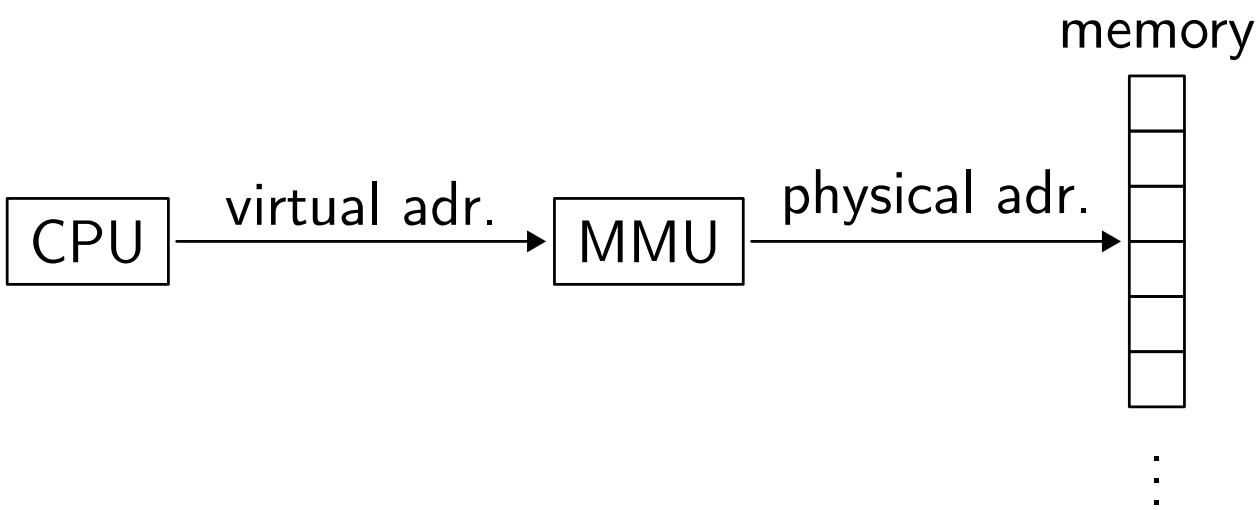
aos quais o CPU pode aceder em leitura e escrita com recurso aos endereços físicos 0, 1, 2, etc.

M é da ordem dos milhares de milhões nos dias de hoje
(por exemplo, $M = 2^{32}$ para 4 Gb de memória)



No entanto, já não é de uso habitual aceder directamente a memória

utiliza-se um mecanismo de **memória virtual** oferecido pelo material, ou seja, o MMU (isto é *memory management unit*)



este traduz os endereços virtuais (de $0, 1, \ldots, N-1$)
para os endereços físicos (para $0, 1, \ldots, M-1$)

é tipicamente o **sistema operativo** que programa e gere o MMU

a memória virtual é dividida em **páginas** (por ex. de 4 kb cada uma)

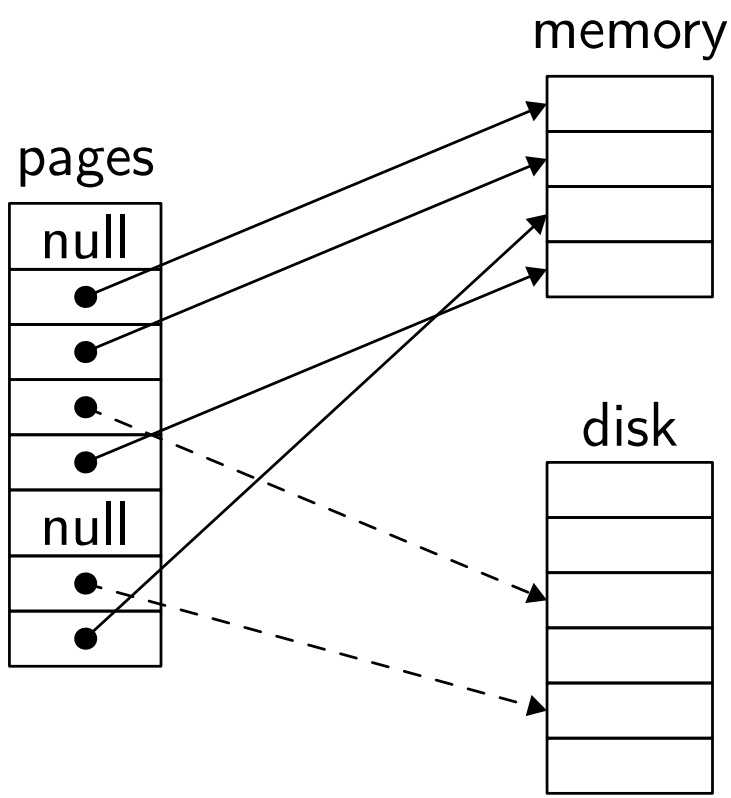
cada página é alternativamente

- não alocada
- alocada em memória física (e o MMU tem esta informação)
- alocada no disco

o sistema operativo mantém uam tabela das páginas

8 páginas

- 2 não alocadas
- 4 em memória física
- 2 no disco



quando o CPU quer ler ou escrever num endereço virtual
o MMU calcula a tradução para um endereço físico

- consegue, e então a instrução é executada
- falha, e
  1. uma interrupção é acionada (*page fault*)
  2. o gestor instala a página em memória física
     (em potência, deslocando uma outra página para o disco)
  3. a execução retoma na mesma instrução

o sistema operativo mantém uma tabela de páginas **por processo**

cada programa tem então a ilusão de dispor da integralidade da memória (virtual) para ele só

facilita

- a edição das ligações (*linkagem*)
  (o código está sempre no mesmo endereço,
  por ex. `0x400000` no Linux 64 bits)

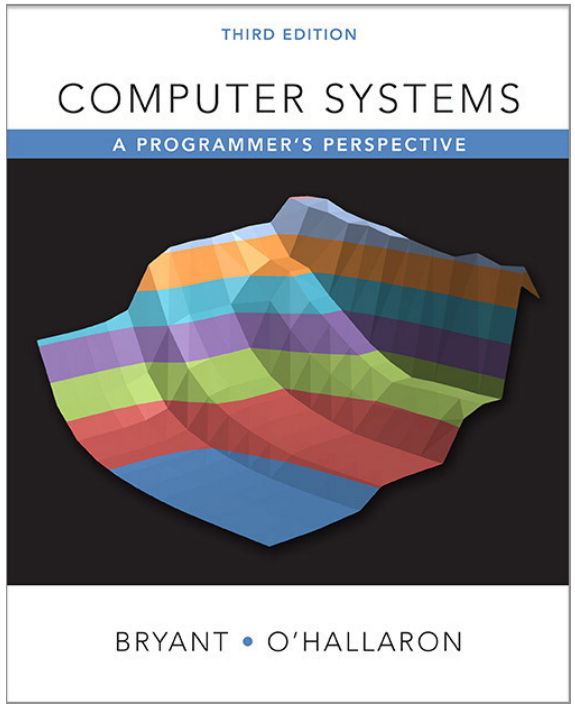
- o carregamento de um programa
  (as páginas já se encontram no disco)

- a partilha de páginas entre vários processos
  (uma mesma página física = várias páginas virtuais)

- alocação de memória
  (as páginas físicas não precisam de ficar contíguas)

para saber mais, em particular sobre o mecanismo de tradução dos endereços, ler

Randal E. Bryant et David R. O'Hallaron
Computer Systems: A Programmer's Perspective
capítulo 9 Virtual Memory



alocação memória

é fácil alocar memória **estaticamente**

- quer seja no segmento `.data` (inicializado explicitamente)
- quer seja no segmento `.bss` inicializado a zero)

no entanto...

a maioria dos programas devem alocar memória **dinamicamente**

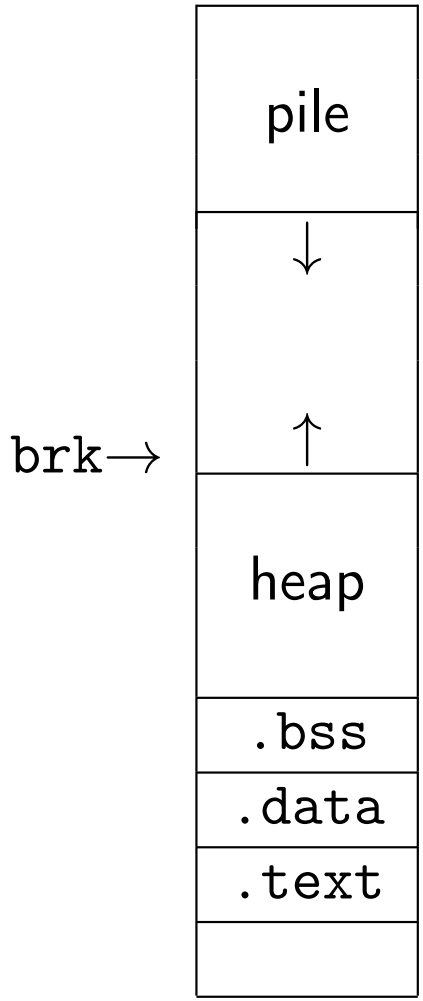
- quer seja implicitamente, via construções da linguagem (objetos, fechos, etc.)
- quer seja explicitamente, para arquivar dados cujo tamanho não é conhecido estaticamente

é igualmente oportuno **libertar** a memória alocada em desuso

esta locação dinâmica faz-se na ***heap***

este está imediatamente por baixo do segmento de dados

o sistema mantém o topo deste numa variável `brk` (*program break*)



a forma mais simples de alocar memória consiste em aumentar `brk`

a chamada sistema

```
void *sbrk(int n);
```

incrementa `brk` de `n` bytes e retorna o seu antigo valor

mas continuamos em não saber libertar espaço memória

para tal, vamos precisar programar o nosso próprio gestor de memória, que possa alocar e libertar blocos de memória

libertar memória pode ser

- explicitamente realizada pelo programador
  exemplo : a biblioteca C `malloc`

- automaticamente realizada pelo gestor
  fala-se então de GC (*garbage collector*)

a partir de `sbrk`, queremos fornecer duas operações

```
void *malloc(int size);
  // retorna um apontador para um novo bloco
  // de pelo menos *size* bytes ou NULL se falhou
```

e

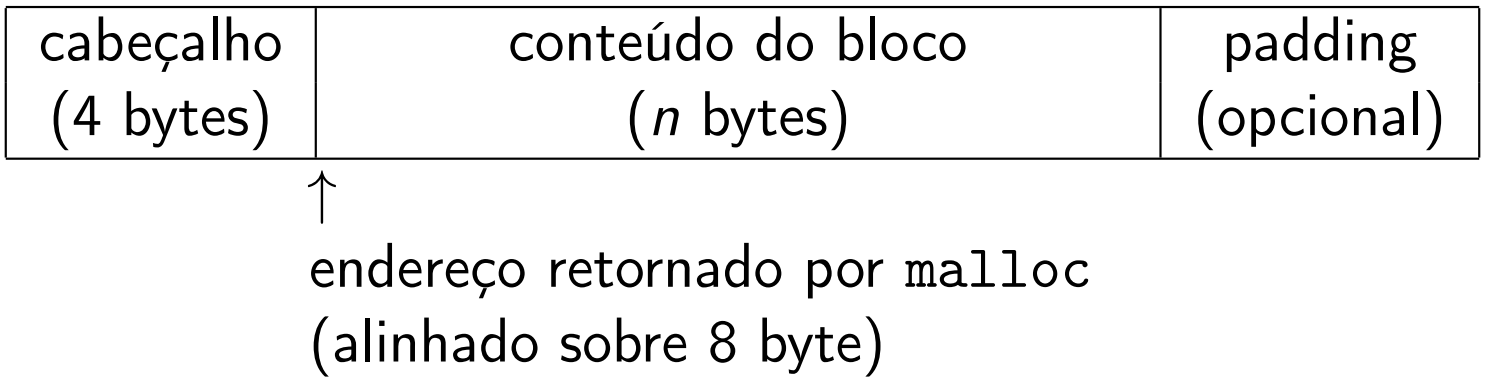
```
void free(void *ptr);
  // liberta o bloco localizado no endereço ptr
  // (que foi alocado por malloc,
  //  senão o comportamento é não-especificado)
```

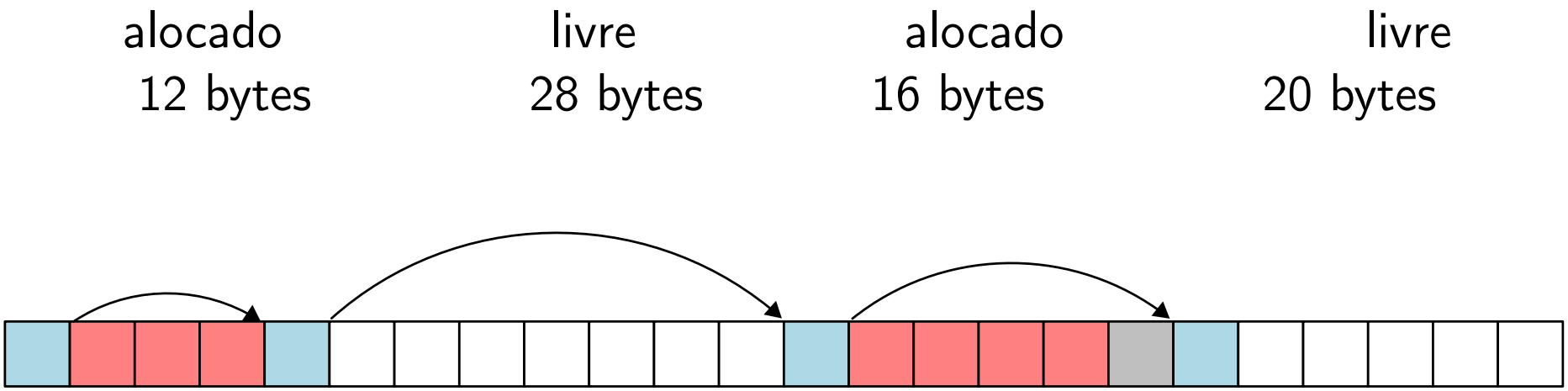
- não assumimos nada sobre a sequência de `malloc` e de `free` por acontecer
- a resposta ao `malloc` não pode ser diferida
- qualquer estrutura de dados necessária ao `malloc` e ao `free` deve ser ela própria alocada na *heap*
- qualquer bloco retornada por `malloc` deve estar alinhada sobre 8 bytes
- qualquer bloco alocado não podem nem ser alterado nem movido

os blocos, alocados ou livres, são **contíguos** em memória

estes são **encadeados** : dado o endereço de um bloco, podemos calcular o endereço do seguinte

- um cabeçalho contém o tamanho (total) e o estado (alocado / livre)
- seguem os *n* bytes do bloco
- e um eventual preenchimento (*padding*) para garantir um tamanho total múltiplo de 8





- un quadrado = 4 bytes
- azul = cabeçalho / vermelho = alocado / cinza = padding / branco = livre

o tamanho total sendo um múltiplo de 8, os seus três bits de peso fraco são nulos

podemos utilizar um destes bits para arquivar o estatuto (alocado / livre)

no exemplo anterior

bit	5	4	3	2	1	0	
...	0	1	0	0	0	1	tamanho 16, alocado
...	1	0	0	0	0	0	tamanho 32, livre
...	0	1	1	0	0	1	tamanho 24, alocado
...	0	1	1	0	0	0	tamanho 24, livre

percoremos a lista dos blocos a procura de um bloco livre suficientemente grande

- se encontramos um, então
  - cortamos, se necessário, em dois blocos (um alocado + um livre)
  - retornamos o bloco alocado
- senão,
  - alocamos um novo bloco no fim da lista, com recurso ao `sbrk`
  - retornamos o bloco

para encontrar um bloco livre, várias estratégias são possíveis

- escolhemos o primeiro bloco com o tamanho suficientemente grande (*first fit*)
- mesma coisa, mas a procura começa onde terminou a precedente (*next fit*)
- escolhemos o menor bloco de tamanho suficientemente grande (*best fit*)

basta mudar o estatuto do bloco *p* (de alocado para livre)

a memória **fragmenta-se** : blocos livres com tamanhos cada vez menores

$\Rightarrow$ memória desperdiçada

$\Rightarrow$ a procura é cada vez mais custosa

é necessário **compactar**

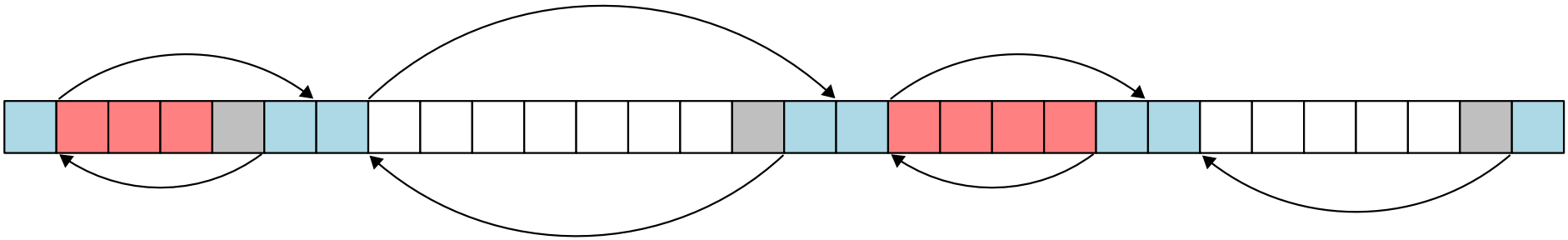
melhoramos o processo com a ideia seguinte : quando um bloco é libertado determinamos se pode ser **fundido** com um bloco adjacente (*coalescing*)

é fácil determinar se o bloco **seguinte** é livre
e de fundir os dois blocos considerados (basta somar os tamanhos)

no entanto, já não é fácil fundir com o bloco anterior

para conseguir tal feito, duplicamos o cabeçalho no **fim** de cada bloco (ideia da autoria de Knuth, chamada de *boundary tags*)

os blocos ficam duplamente encadeados



quando libertamos um bloco p, examinamos o bloco seguinte e o bloco anterior

há 4 situações possíveis

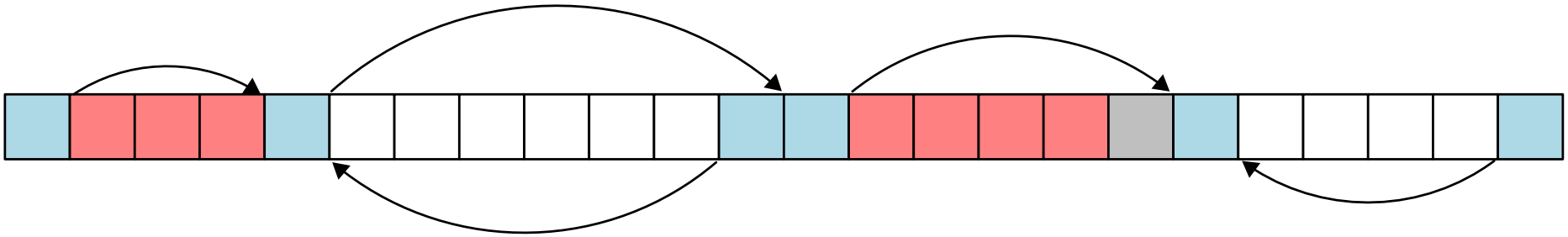
- alocado | p | alocado : nada por fazer
- alocado | p | livre : fusão com o seguinte
- livre | p | alocado : fusão com o anterior
- livre | p | livre : fusão dos três blocos

mantemos assim o **invariante** que nunca há dois blocos livres adjacentes

duplicar o cabeçalho ocupa espaço

mas podemos

- fazê-lo somente nos blocos livres
- utilizar um bit no cabeçalho para indicar se o bloco anterior está livre



não deixa de ser custoso percorrer todos os blocos por alocar

daí a ideia de encadear entre eles os blocos livres, exclusivamente (*free list*)

para isso, utilizamos o conteúdo do bloco, que se encontra livre, para arquivar dois apontadores (isto implica um tamanho mínimo para o bloco livre)

quando libertamos um bloco, temos várias hipóteses para o reinserir na lista

- inserção à cabeça
- lista ordenada por endereços crescentes
- lista ordenada por tamanho de bloco
- etc.

percorrer toda a *free list* pode ainda ser custoso se numerosos blocos tem um tamanho pequeno

daí a ideia de ter **várias listas** de blocos livres, organizados por tamanho

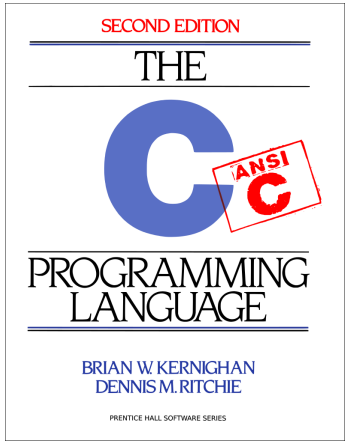
exemplo : uma lista de blocos livres de tamanho compreendidas entre 2^n e $2^{n+1} - 1$, para cada n

como vimos, `malloc/free` são muito mais súbtil do que parece
(o `malloc.c` de Linux tem mais do que 5000 linhas)

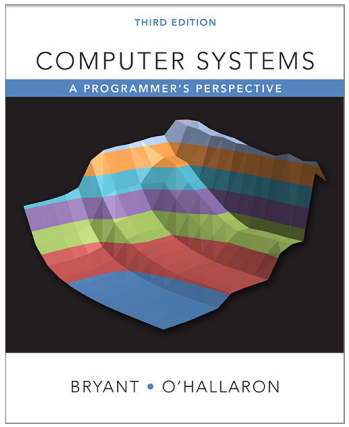
muito parâmetros, muitas estratégias possíveis

uma literatura volumosa sobre esta questão, com muitas avaliações empíricas

encontramos código C implementando estas ideias em



- Brian W. Kernighan et Dennis M. Ritchie
  The C programming language



- Randal E. Bryant et David R. O'Hallaron
  Computer Systems: A Programmer's Perspective

muitas linguagens de programação (Lisp, OCaml, Haskell, Java, etc.) assentam sobre um mecanismo **automático** de libertação de blocos de memória,
chamado **GC** (de *Garbage Collector*)

em português, GC poderia ser traduzido, para além de recolha de lixo/células, por « limpa migalhas »

princípio : o espaço alocado na *heap* para um dado (fecho, registo, vector, construtor, etc.) que deixou de ser **atingível / acessível** a partir das variáveis do programa pode ser **recuperado** para poder ser reutilizado para outros dados

dificuldade : não podemos em geral determinar estaticamente (durante a compilação) o momento em que um dado deixa de ser acessível
$\Rightarrow$ o GC ter de ser então componente do executável

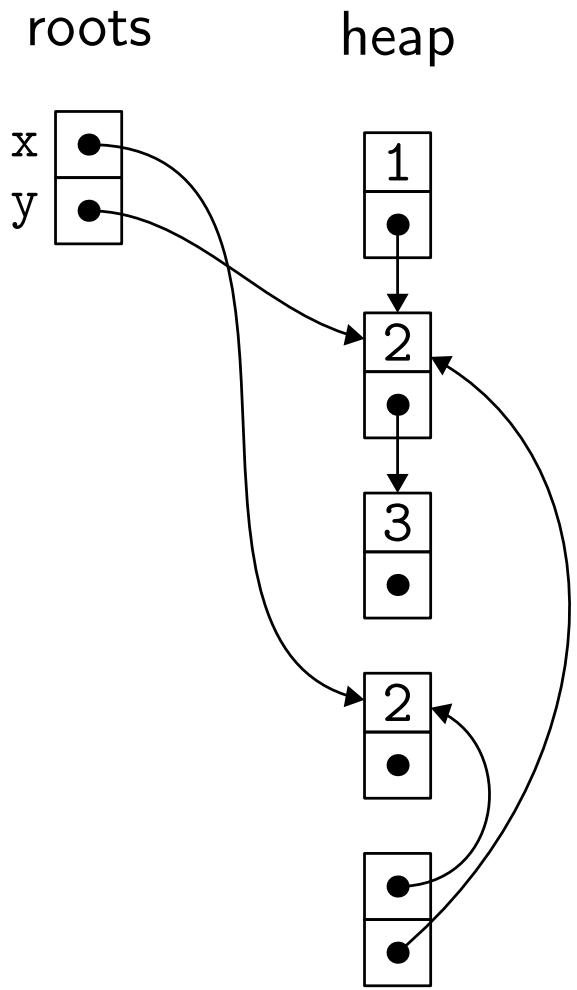
- ou como uma parte do intérprete para uma linguagem interpretada
- ou como uma biblioteca/API *ligada (linked)* com o código compilado para uma linguagem compilada (*runtime*)

um bloco pode conter um ou mais apontadores para outros blocos (os seus **filhos**) mas também outros dados (caracteres, inteiros, apontadores fora da *heap*, etc.)

dado um instante da execução do programa, chamamos **raíz** toda a variável activa neste instante (variável global, variável local contido numa tabela de activação ou num registo)

dizemos que um bloco está **vivo** se é acessível a partir de uma raíz *i.e.* se existe um caminho de apontadores que liga uma raíz ao bloco em causa

```
let x,y =
  let l = [1; 2; 3] in
  (List.filter even l, List.tl l)
...
```



consideramos uma primeira solução, chamada **contagem das referências** (*reference counting*)

a ideia é associar a cada bloco o número de apontadores que apontam para este bloco (desde as raízes ou desde outros blocos)

quando este número passa a zero, sabemos que podemos libertar este bloco, e decrementamos o contador de todos os seus filhos

la actualização do contador tem lugar aquando de uma **atribuição** (explícita ou implícita como em `1::x`) da forma $b.f \leftarrow p$; é necessário

- decrementar o contador correspondente ao antigo apontador $b.f$; se passa a 0, desalocar este bloco
- incrementar o contador de bloco p

problema :

- a actualização dos contadores é (muito) custosa
- os **ciclos** nas estruturas de dados impossibilita a recuperação dos blocos correspondentes

a contagem de referências é raramente utilizada nos GC (uma excepção: a linguagem Perl) mas as vezes utilizada explicitamente por programadores

consideremos uma outra solução, mais eficaz, designada de ***mark and sweep***

prossegue em dois tempos

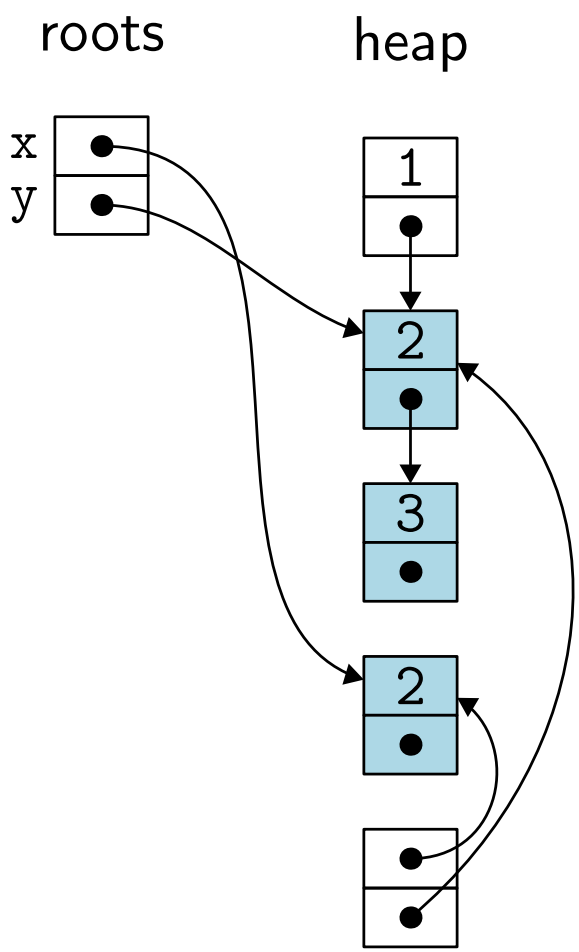
1. marcamos todos os blocos atingíveis a partir das raízes (utilizando um percurso em profundidade primeiro e um bit em cada bloco)
2. examinamos cada bloco e
   - recuperamos quem não tem marca (colocados novamente na *free list*)
   - removemos as marcas presentes nos outros blocos

quando queremos alocar um bloco, examinamos a *free list* ; se esta está vazia, é uma boa altura para realizar um GC

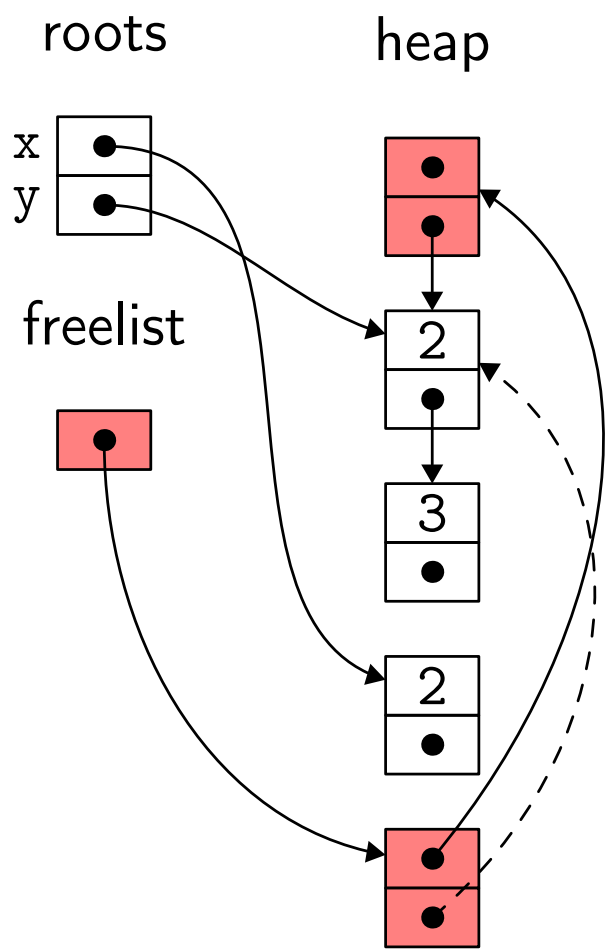
a marcação utiliza um percurso em profundidade, da seguinte forma

```
percurso(x) =
   se x é um apontador para a heap ainda por marcar
      marcar x
      para cada campo f de x
         percurso(x.f)
```

marcar



varrer



problema : a marcação prossegue de forma **recursiva**, que, sem chamadas terminais, vai utilizar um tamanho em pilha proporcional à profundidade da *heap*; esta pode ser tão grande quanto a própria *heap*

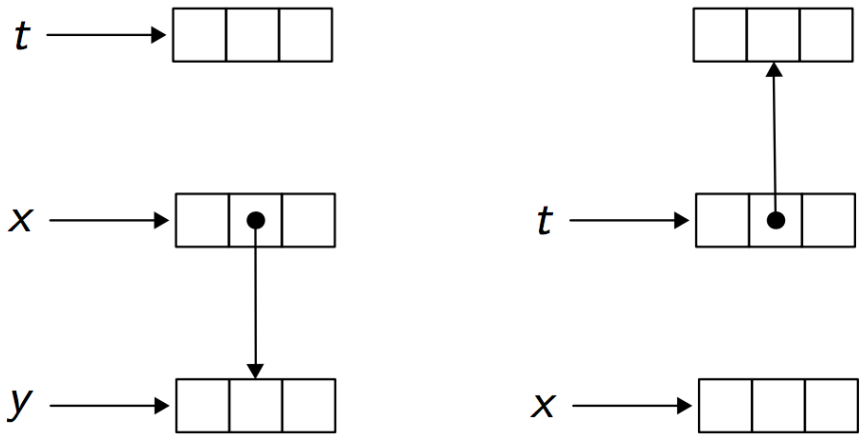
solução : utilizamos a própria estrutura auscultada para codificar a pilha de chamadas recursivas (*pointer reversal*)

pointer reversal (Deutsch / Schorr & Waite, 1965)

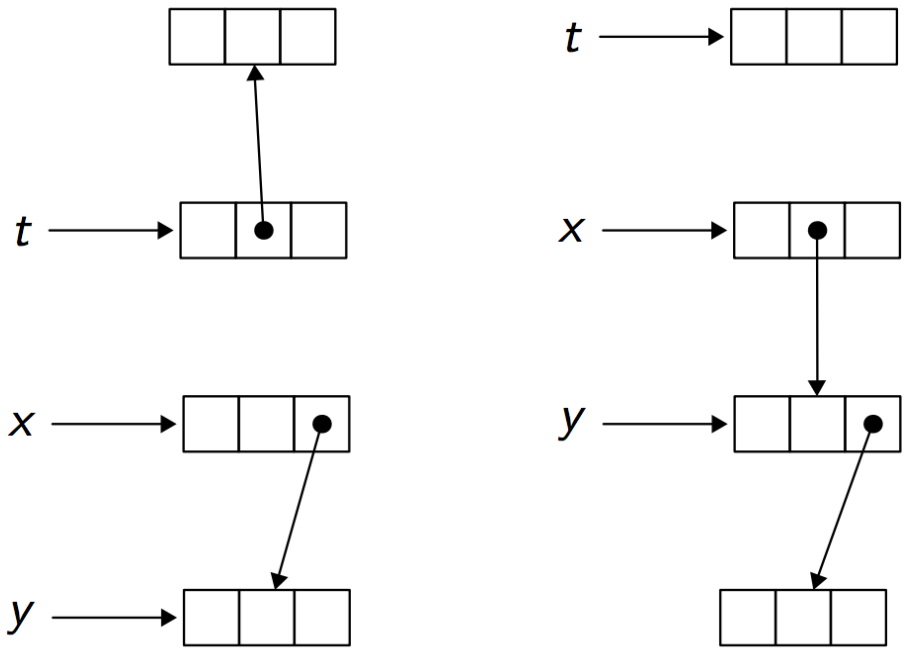
o percurso é mais complexa, mas já só utiliza duas variáveis
e um campo inteiro done[x] em cada bloco

```
percurso(x) =
   se x é um apontador para a heap ainda por marcar
      t <- null; marcar x; done[x] <- 0
      enquanto verdade
         i <- done[x]
         se i < número dos campos de x
            y <- x.fi
            se y é um apontador para a heap ainda por marcar   //descer
               x.fi <- t; t <- x; x <- y
               marcar x; done[x] <- 0
            senão
               done[x] <- i+1
         senão //subir
            y <- x; x <- t
            se x = null então terminamos
            i <- done[x]; t <- x.fi; x.fi <- y; done[x] <- i+1
```

```
...
  se y é um apontador para a heap ainda por marcar  //descer
      x.fi <- t; t <- x; x <- y
      marcar x; done[x] <- 0
...
```



```
... //subir
    y <- x; x <- t
    se x = null então terminamos
    i <- done[x]; t <- x.fi; x.fi <- y; done[x] <- i+1
```



é uma boa solução para determinar os blocos por recuperar
(em particular, recuperamos convenientemente os ciclos inatingíveis)

mas ainda não consegue propor uma solução real ao problema da fragmentação

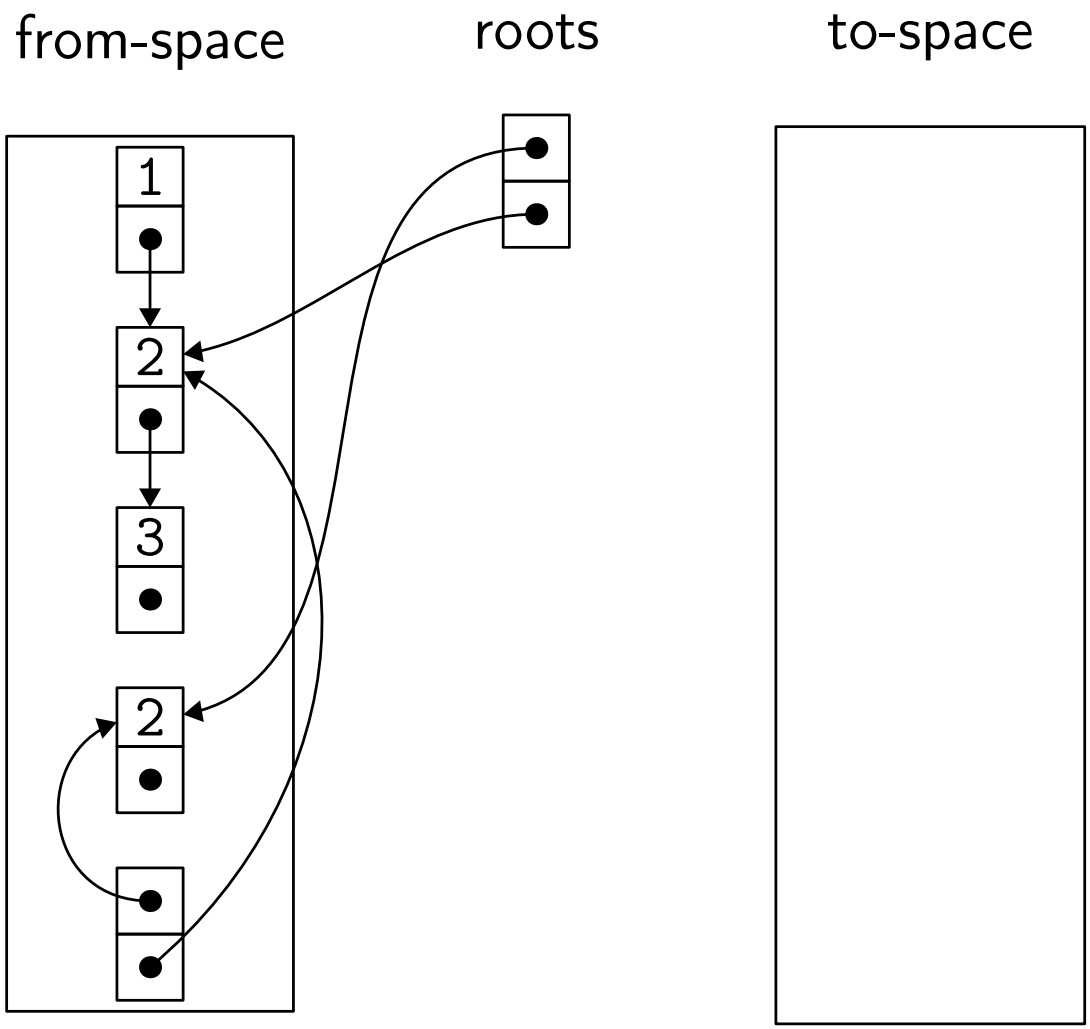
consideremos ainda uma solução diferente, designada de **parar e copiar** (*stop and copy*)

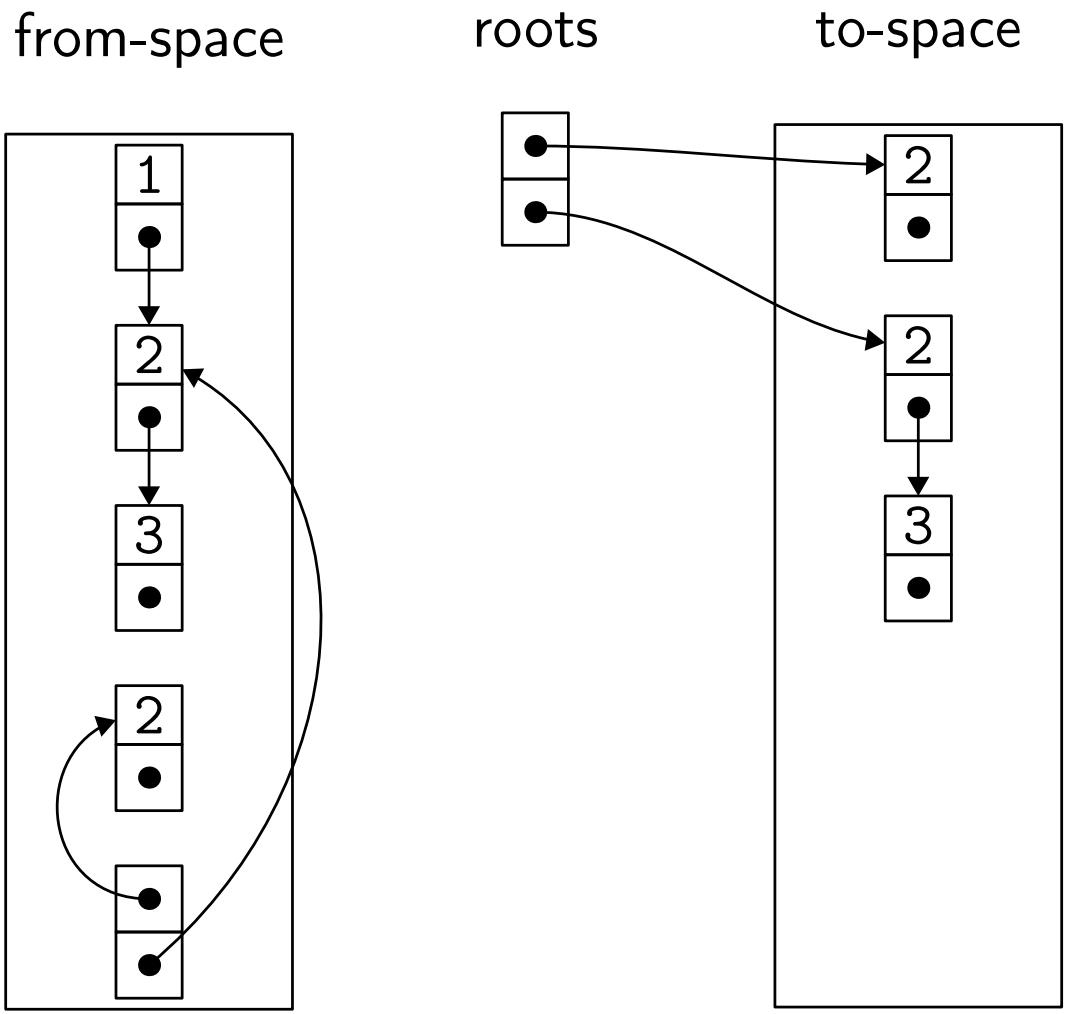
a ideia subjacente é cortar a *heap* em duas metades

1. utilizamos somente uma delas, na qual alocamos linearmente
2. quando está cheia, copiamos tudo o que for atingível para a outra metade, e trocamos os papeis das duas metades

benefícios imediatos :

- a alocação é muito pouco custosa (uma soma e uma comparação)
- já não há o problema da fragmentação


from-space
roots
to-space
1
2
3
2


from-space
roots
to-space
1
2
3
2
2
2
3

problema : é necessário realizar a cópia utilizando um espaço constante

solução : vamos efectuar um percurso em largura e utilizar o espaço de chegada como zona de arquivo dos apontadores por copiar

quando um bloco foi deslocado da primeira zona (`from-space`) para a segunda (`to-space`) então utilizamos o seu primeiro campo para indicar para onde foi copiado

começamos por escrever uma função que copia o bloco no endereço p, se isso ainda não foi feito

`next` designa o primeiro local livre em `to-space`

```
forward(p) =
   se p aponta para from-space
      se p.f1 aponta para to-space
         retornar p.f1
      senão
         para cada campo fi de p
            next.fi <- p.fi
         p.f1 <- next
         next <- next + tamanho do bloco p
         retornar p.f1
   senão
      retornar p
```

podemos então realizar a cópia, começando pelas raízes

```
scan <- next <- início de to-space
para cada raíz r
   r <- forward(r)
enquanto scan < next
   para cada campo fi de scan
      scan.fi <- forward(scan.fi)
   scan <- scan + tamanho do bloco scan
```

a zona de `to-space` situada entre `scan` e `next` representa os blocos cujos campos ainda não foram actualizados

de notar que `scan` avança, mas que `next` também !

apesar de elegante, este algoritmo tem pelo menos um defeito : ele modifica a *localidade* dos dados *i.e.* blocos que estavam mutuamente próximos antes da cópia não o serão necessariamente mais depois

num sistema de memória cache, a propriedade de localidade é importante

é possível modificar o algoritmo de cheney para efectuar uma mistura entre percurso DFS e BFS
(ver Appel, capítulo 13)

em muitos programas, a maior parte dos valores tem uma duração de vida muito curta, e as que sobrevivam a várias recolhas do GC são normalmente susceptíveis em sobreviver a muitas outras recolhas.

daí a ideia em organizar a *heap* em várias **gerações**

- G_0 contém os valores mais recentes e realizamos ali recolhas frequentes
- G_1 contém valores mais antigos do que as de G_0, e faz-se ali recolhas com menos frequência
- etc.

em prática, há algumas dificuldades para identificar as raízes de cada geração, em particular porque uma atribuição pode introduzir um apontador dee G_1 para G_0, por exemplo (ver Appel, capítulo 13)

enfim, não é desejável que o programa seja interrompido demasiado tempo por causa de uma recolha (um estorvo para programas interactivo, crítico para programas de tempo real)

para remediar esta situação, podemos utilizar um **GC incremental**, que marca os blocos pouco a pouco, à medida das chamadas ao GC

uma simples marca já não é suficiente, são preciso tres tipos de marca (três cores)
(ver Appel, capítulo 13)

o GC de OCaml é um GC de duas gerações

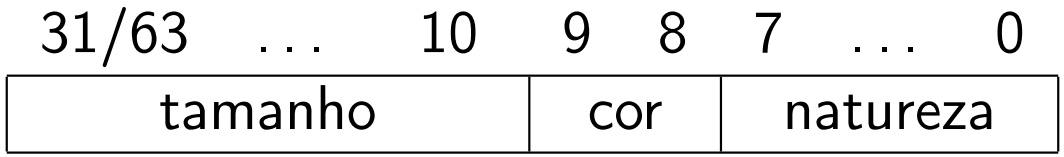
- um GC menor (valores recentes) : *Stop & Copy*
- um GC maior (valores antigos) : *Mark & Sweep* incremental

a zona `to-space` do GC menor é a zona do GC maior

agora que conhecemos as necessidades do GC, podemos explicar a **representação dos dados** na *heap* (aqui no caso de OCaml)

cada bloco na *heap* é precedido de um **cabeçalho** cujo tamanho é uma **palavra**
(4 bytes numa arquitectura 32 bits, 8 numa arquitectura 64 bits)

o cabeçalho contém o tamanho do bloco, a sua natureza e 2 bits utilizados pelo GC



o tamanho é aqui um número de palavras ; se o tamanho é n, o bloco todo inteiro, com o seu cabeçalho, ocupa então $n + 1$ palavras

(cuidado : é um cabeçalho diferente do cabeçalho de `malloc`)

a natureza do bloco é um inteiro codificado sobre 8 bits (0..255) ; permite distinguir

- flutuantes
- strings
- vectores de flutuantes
- objetos
- fechos
- o caso geral de um bloco estruturado: estruturas, vectores, *n*-tuplos, construtores ; neste último caso, o inteiro indica de qual construtor se trata (para a filtragem)

o tamanho do bloco estando codificada sobre 22/54 bits, temos

```
# Sys.max_array_length;;
- : int = 4194303                (* máquina 32 bits *)
```

```
# Sys.max_array_length;;
- : int = 18014398509481983 (* máquina 64 bits *)
```

a strings são no entanto representadas de forma compacta (4 caracteres por palavra), o que dá

```
# Sys.max_string_length;;
- : int = 16777211                 (* máquina 32 bits *)
```

```
# Sys.max_string_length;;
- : int = 144115188075855863 (* máquina 64 bits *)
```

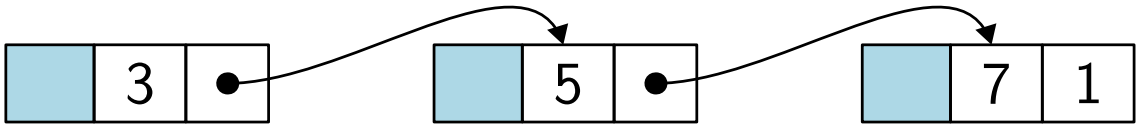
um valor OCaml cabe numa só palavra ; é

- ou um inteiro ímpar $2n + 1$, representando então o valor n de tipo `int` ou um construtor constante codificado por n (`true`, `false`, `[]`, etc.)
- ou um apontador (necessariamente par por razões de alinhamento), que pode apontar dentro ou fora da *heap*

o GC testa assim o bit de peso fraco para determinar se um campo é um apontador ou não, porque na presença de polimorfismo, o compilador não sabe indicar ao GC quais são os campos que são apontadores

```
let f x = (x, x)
```

exemplo : o valor `1 :: 2 :: 3 :: []` é representado da seguinte forma



consequência : os inteiros de OCaml são inteiros 31/63 bits com signo (mas a biblioteca fornece módulos `Int32` e `Int64`)

enfim, é importante relembrar que o modo de passagem de OCaml é a passagem **por valor**, mesmo se uma parte consequente dos valores são na realidade apontadores

conclusão

- Prática Laboratorial desta semana
  - programação de um GC *stop & copy*
- Próxima aula
  - produção de código eficiente, parte 1

estes acetatos resultam essencialmente de uma adaptação do material pedagógico gentilmente cedido pelo Jean-Christophe Filliâtre (link1, link2)

adicionalmente poderá consultar as obras seguintes

- **Modern Compilers: Principles, Techniques, and Tools**, Alfred V. Aho, Monica S. Lam, Ravi Sethi et Jeffrey D. Ullman
- **Types and Programming Languages**, Benjamin C. Pierce
- **Modern Compiler Implementation**, Andrew W. Appel (3 versões: ML, C, Java)