

# Compiladores (CC3001)

## Aula 8: Âmbitos e Tabelas de Símbolos

Pedro Vasconcelos

DCC/FCUP

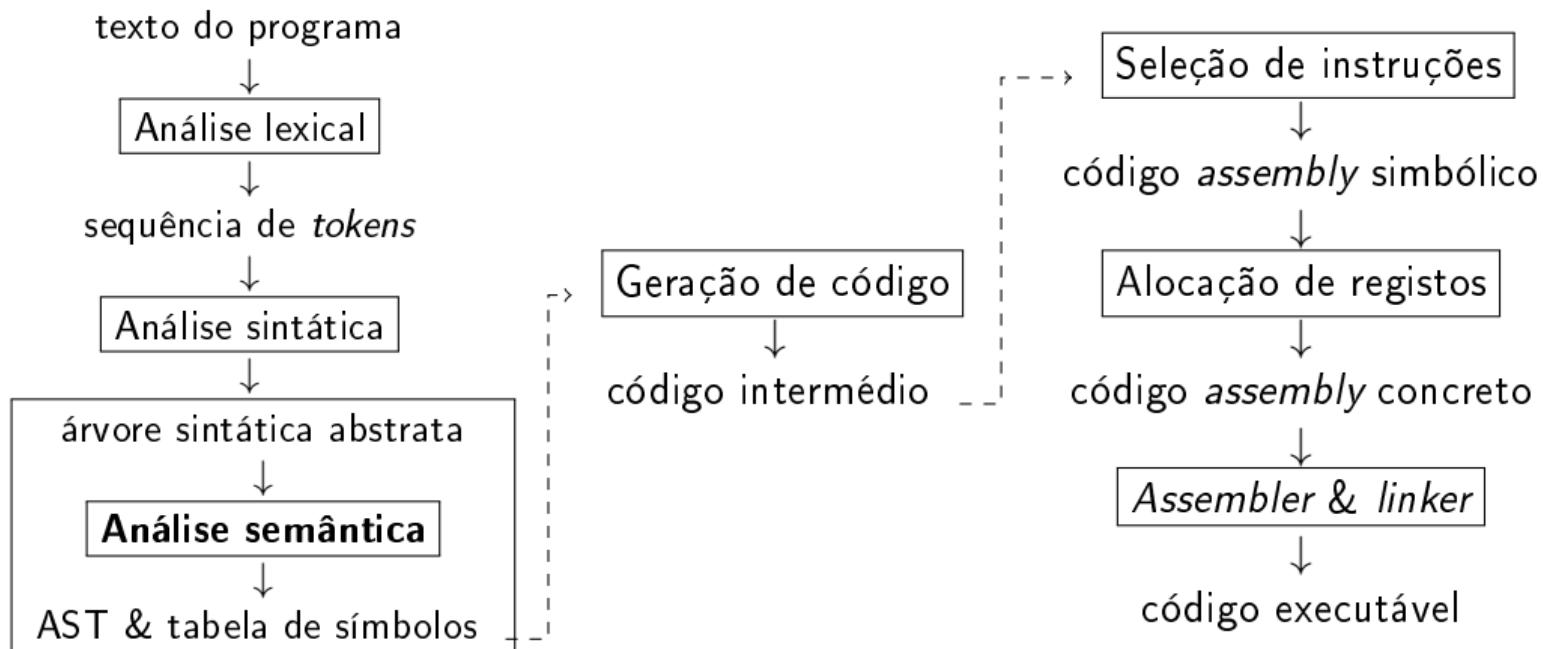
2020



Análise semântica

Âmbitos

Tabelas de símbolos



Todas as verificações de correção feitas pelo compilador que são **dependentes de contexto**.

Exemplos:

- ▶ garantir que as variáveis são declaradas antes do uso
- ▶ verificar que tipo declarado para cada variável é compatível com as operações efetuadas
- ▶ garantir que as funções são invocadas com o número e tipo correto de argumentos

- ▶ Em algumas linguagens e compiladores estas verificações podem ser feitas durante a análise sintática (e.g. primeiros compiladores de C)
- ▶ Em geral são dependentes de contexto, logo é melhor efetua-las como uma passagem separada sobre árvore sintática abstrata (AST)
- ▶ Frequentemente a análise semântica coleciona informação sobre **identificadores** (variáveis, funções, etc.) numa **tabela de símbolos**
- ▶ A tabela de símbolos será também usada para geração de código (e.g. para guardar informação sobre localização de cada variável)

- ▶ Os **identificadores** numa linguagem de programação são nomes para diferentes entidades (variáveis, funções, tipos, ...)
- ▶ Cada identificador é **declarado uma vez** e **referido múltiplas vezes**
- ▶ É comum que as declarações tenham **âmbito limitado** e que o identificador possa ser re-utilizado sem colisões
  - ▶ e.g. podemos declarar uma variável local a uma função com um mesmo nome que uma variável global

Identificar os âmbitos neste fragmento de C.

```
int i, j;          // global: i, j
int f(int size)   // global: f
{   char i, temp; // função: size, i, temp
    ...
    { double j;   // bloco 1: j esconde a global
        ...
    }
    ...
    { char * j;   // bloco 2: j esconde a global
        ...
    }
}
```

Âmbito lexical (também designado *âmbito estático*) cada uso de um identificador é sempre associado à **declaração envolvente mais próxima** na AST

- ▶ Esta é a forma usual de tratamento do âmbito de variáveis em todas as linguagens de programação genéricas (Pascal, C, C++, Java, Haskell, SML, etc.)
- ▶ Poucas linguagens usam outras regras para âmbitos (exemplo: primeiras versões de LISP e algumas linguagens de *scripting* usam *âmbito dinâmico*)

- ▶ Algumas linguagens permitem **declarar funções em qualquer âmbito** (inclusivamente dentro de outras funções)
  - ▶ ALGOL 68 e família: Pascal, Modula e Ada,...
  - ▶ todas as linguagens funcionais: Scheme, ML, Haskell,...
  - ▶ algumas linguagens de *scripting*: JavaScript, Ruby, Python,...
- ▶ Linguagens da família do C **limitam a declaração das funções** ao âmbito global (C) ou das classes (C++/Java)
  - ▶ em C *standard* não podemos declarar uma função *local* a uma outra função
  - ▶ mas o compilador GCC permite tais declarações (uma extensão)

```
// Pascal
function E(x: real): real;
    function F(y: real): real;
    begin
        F := x + y
    end;
begin
    E := F(3) + F(4)
end;
```

```
-- Haskell
e :: Float -> Float
e x = f 3 + f 4
    where f y = x + y
```

```
// GNU C (não é standard)
float e(float x) {
    float f(float y) {
        return x + y;
    }
    return f(3) + f(4);
}
```

Porquê declarar funções dentro de funções?

- ▶ esconder informação (funções auxiliares são locais)
- ▶ permite simplificar o controlo de fluxo
- ▶ nas linguagens funcionais: funções locais auxiliares são usadas em vez de ciclos

Exemplo: testar primos (C vs. Haskell).

```
int is_prime(int n) {  
    int d = 2;  
    if(n<=1)                isPrime :: Int -> Bool  
        return FALSE;          isPrime n = n>1 && checkDivs 2  
    while(d*d <= n) {        where checkDivs d  
        if (n%d == 0)          | n`mod`d==0 = False  
            return FALSE;        | d*d > n    = True  
        d++;                  | otherwise  = checkDivs (d+1)  
    }  
    return TRUE;  
}
```

- ▶ Estruturas de dados que associam **identificadores** no código fonte a alguma **informação**; exemplos:
  - ▶ *valores* de variáveis (num interpretador);
  - ▶ *tipos* declarados para variáveis (num compilador);
  - ▶ tipo de *argumentos* e *resultado* (para funções)
  - ▶ *localização* em registos/memória (durante a geração de código)
- ▶ Num compilador a tabela de símbolos é construída na análise semântica e usada durante a geração de código

- ▶ É comum as linguagens dividirem identificadores em **espaços de nomes separados**
  - ▶ e.g. em Haskell podemos usar o mesmo nome para um *tipo* e um *construtor*

```
data Expr = Int Int | Add Expr Expr
```

- ▶ sistemas de **módulos** ou *packages* também definem espaços de nomes separados
- ▶ Podemos usar **tabelas de símbolos separadas** para diferentes espaços
- ▶ Alternativa: usar uma tabela única e **distinguir os nomes por prefixos**

```
Prelude.lookup    -- no prelúdio
```

```
Data.Map.lookup  -- no módulo Data.Map
```

Operações fundamentais:

**inicializar** uma *tabela vazia*;

**inserir** dado o *identificador* e *informação* inserir uma nova entrada numa tabela;  
(se o identificador já ocorre, a informação deve alterada)

**procurar** dado o *identificador* devolve a *informação* associada (caso exista) ou um  
sinal de falha (i.e. uma excepção ou um resultado opcional)

Suportadas em vários tipos para *dicionários*. Exemplos:

- ▶ Map em Java
- ▶ Data.Map em Haskell
- ▶ dict em Python
- ▶ std::map em C++

Necessitamos de duas operações extra para âmbitos:

**abrir** iniciar num novo âmbito (i.e. à entrada de um bloco ou função)

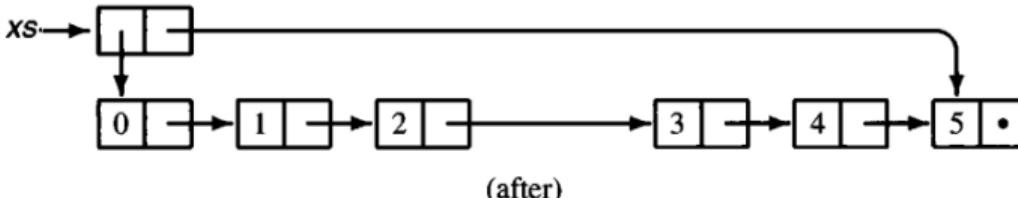
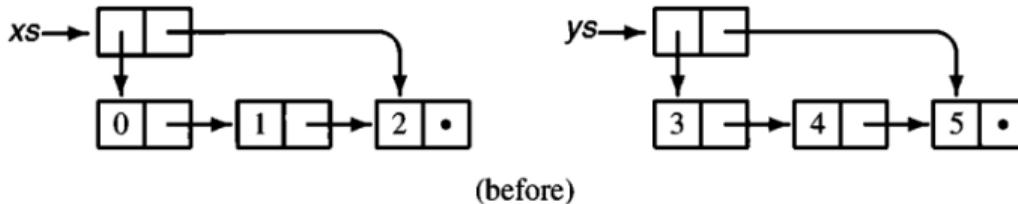
**fechar** terminar o âmbito atual *repondo todas as associações como estavam antes da abertura do âmbito*

Nota: **não é suficiente apagar entradas** ao fechar o âmbito (é necessário repor as associações anteriores).

- ▶ Podemos implementar usando diferentes estruturas concretas: listas ligadas, árvores de pesquisa, tabelas de partição (*hash tables*)
- ▶ Podemos usar estruturas **persistentes** ou **efémeras**:  
persistentes as operações preservam a versão anterior da estrutura  
efémeras as operações destroem a versão anterior da estrutura
- ▶ Estruturas dados em **linguagens funcionais** são normalmente persistentes enquanto em **linguagens imperativas** são normalmente efémeras

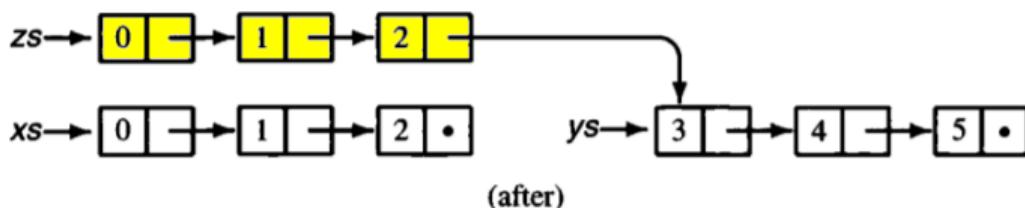
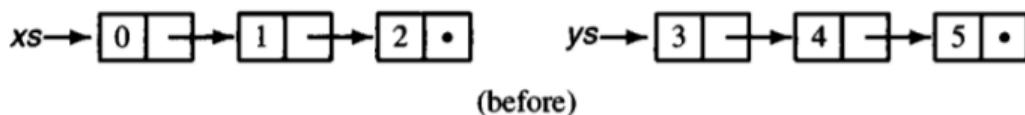
## Estrutura efémera (C, Java, etc.)

```
xs = new List(0,1,2);  
ys = new List(3,4,5);  
xs.append(ys);
```



## Estrutura persistente (Haskell, SML, etc.)

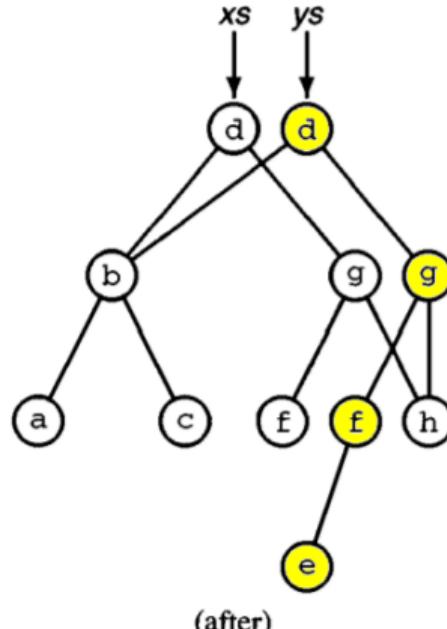
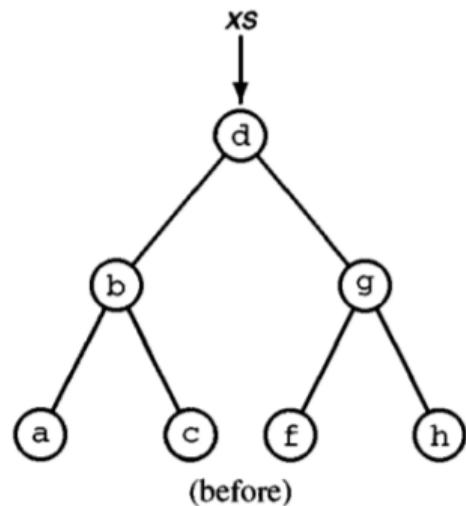
```
xs = [0,1,2]  
ys = [3,4,5]  
zs = xs ++ ys
```



Outro exemplo: conjuntos implementados como árvores *binárias de pesquisa*.

```
xs = Set.fromList ["a", "b", "c", "d", "f", "g"]
```

```
ys = Set.insert "e" xs
```



- ▶ Se a **tabela for persistente**: a implementação de âmbitos é trivial
  1. guardar uma referência para a tabela ao abrir o âmbito;
  2. restaurar a tabela ao sair do âmbito.<sup>1</sup>
- ▶ Se a **tabela for efémera**: necessitamos de trabalho extra  
(e.g. guardar valores anteriores das chaves inseridas e defazer as alterações)

---

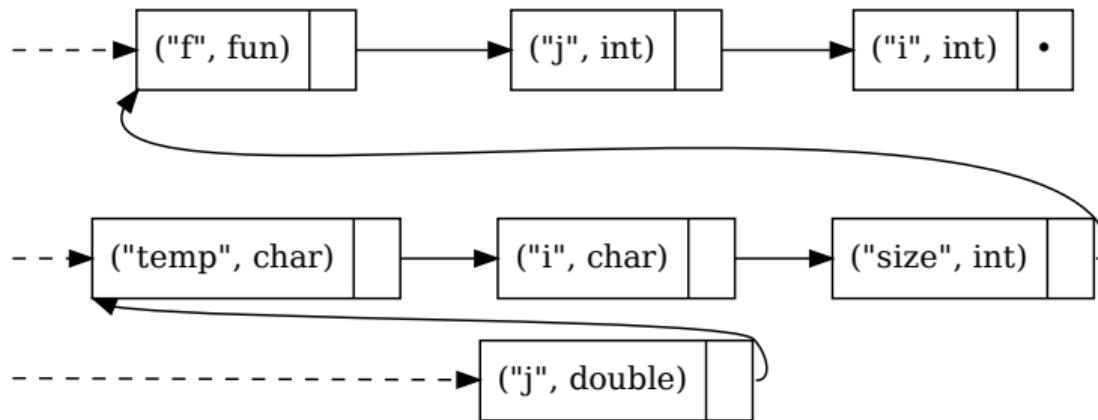
<sup>1</sup> Espaço em memória não utilizado será libertado pelo recoletor de lixo.

Uma lista de pares  $(identificador, info)$ :

- `inicializar` a lista vazia `[]`;
- `inserir` acrescentar  $(ident, info)$  ao início da lista;
- `procurar` do ínicio para o final da lista (i.e. `lookup` do prelúdio de Haskell);
- `abrir âmbito` lembrar a tabela atual;
- `fechar âmbito` voltar à tabela guardada.

## Exemplo: tabela para tipos

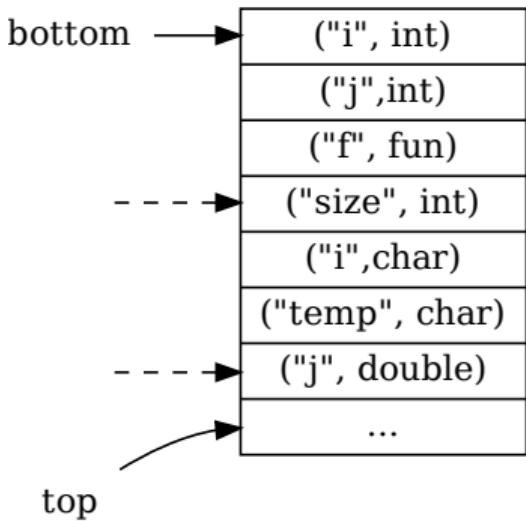
```
int i, j;  
int f(int size)  
{  char i, temp;  
  ...  
  { double j;  
  ...
```



Opera de forma similar às listas funcionais.

- inicializar** uma pilha vazia;
- inserir** empilhar (*name,info*)
- procurar** do topo para pilha para o início;
- abrir âmbito** guardar o atual topo da pilha;
- fechar âmbito** repor o topo da pilha guardado.

```
int i, j;  
int f(int size)  
{  char i, temp;  
    ...  
{  double j;  
    ...
```



- ▶ Tabelas implementadas como listas ou pilhas usam **pesquisa sequêncial**:  $O(n)$  para uma tabela com  $n$  entradas
- ▶ Não é muito importante num protótipo ou compilador experimental...
- ▶ Mas num compilador real,  $n$  pode ser bastante grande (número de identificadores do programa e bibliotecas)
- ▶ Alternativas mais eficientes: **árvores de pesquisa binária** ou **tabelas de partição**

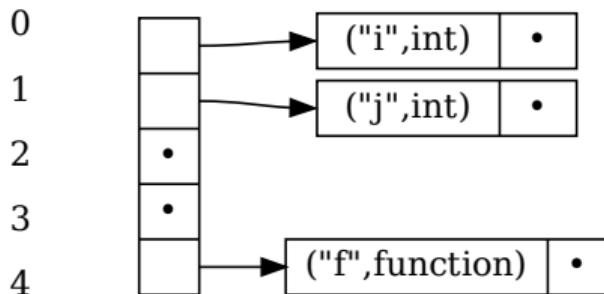
- ▶ Baseadas numa **ordem total sobre as chaves** ( $>$ ,  $<$ ,  $==$ )
- ▶ Árvores auto-equilibradas (AVL, Red-Black) garantem pesquisa e inserção em  $O(\log n)$  comparações no pior caso
- ▶ Implementações funcionais são usualmente **persistentes**:
  - ▶ e.g. `Data.Map` em Haskell
  - ▶ OK para tabelas de símbolos (mas as *tabelas de partição* são uma melhor opção)
- ▶ Implementações imperativas são usualmente **efémeras**:
  - ▶ e.g. `java.util.TreeMap` em Java
  - ▶ não são a boa opção para tabelas de símbolos (por causa da dificuldade de tratamento de âmbitos)

- ▶ Ao pesquisar numa árvore binária necessitamos de comparar chaves
- ▶ Contudo: a **comparação de cadeias não é  $O(1)$**  (necessita de percorrer as sequências dos caracteres)
- ▶ Alternativa: pesquisa sem comparações usando uma **tabela de partição (hash tables)**
- ▶ Vamos ver uma implementação imperativa simples apropriada para tabelas de símbolos

- ▶ Um *array*  $b[0], \dots, b[N - 1]$  de listas ligadas (*buckets*)
- ▶ Uma função  $h$  de *hash* que converte uma chave (identificador) num *inteiro* no intervalo 0 a  $N - 1$  (correspondente a um dos *buckets*)
- ▶ Pesquisamos uma chave  $k$  no *bucket*  $b[h(k)]$
- ▶ Quando duas chaves diferentes têm o mesmo *hash*: resolver a colisão usando a lista ligada
- ▶ Escolher o número  $n$  de *buckets* e a função  $h$  tal que:
  1. a função seja rápida de calcular
  2. distribua as chaves pelos *buckets* de forma o mais uniforme possível
- ▶ Se a função de *hash* e número de *buckets* forem bem escolhidos as colisões são raras e a pesquisa é eficiente (na prática: tempo constante)

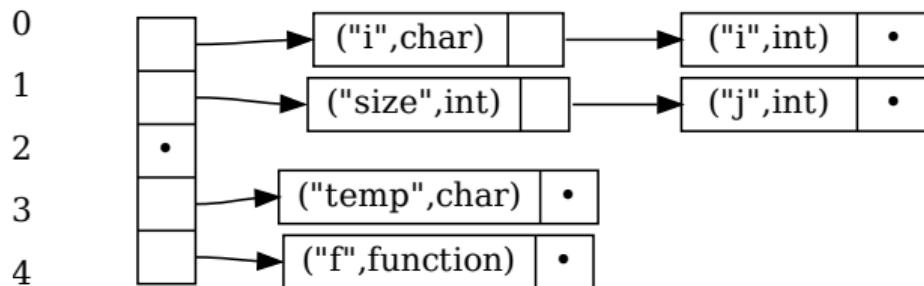
Exemplo:

- ▶ Uma tabela de partição com  $N = 5$  (número de *buckets*) e 3 entradas (*i, j, f*)
- ▶ Supondo que  $h(i) = 0$ ,  $h(j) = 1$  e  $h(f) = 4$
- ▶ Não há colisões: cada *bucket* têm uma só entrada



(Nota: na prática  $N$  será muito maior — na ordem das centenas ou milhares.)

- ▶ Inserimos novas entradas para `i`, `size` e `temp`
- ▶ Temos **colisões** na chave repetida (`i`) e (supondo) também numa diferente:  $h(\text{size}) = h(\text{j}) = 1$
- ▶ Resolvemos as colisões **colocando as novas entradas à cabeça**:



- ▶ Pesquisamos sempre do início para o fim de cada lista
- ▶ No final de um âmbito removemos todas as entradas introduzidas

Como escolher uma função de *hash* para identificadores?

$$x = x_0 x_1 \dots x_{k-1}$$

( $k$  é o comprimento da cadeia e  $x_i$  são os códigos dos caracteres).

- ▶ Muitas heurísticas diferentes dependendo de *trade-offs* (velocidade / colisões)
- ▶ Solução simples: somar os códigos multiplicados por potências de  $\alpha$

$$h(x) = (x_0 \alpha^{k-1} + x_1 \alpha^{k-2} + \dots + x_{k-1} \alpha^0) \bmod N$$

Escolhendo  $\alpha$  potência de 2: podemos implementar as multiplicações como *shifts*

- ▶ Numa implementação real: deve usar implementações de algoritmos mais robustos...

Exemplo de implementação em C ( $\alpha = 2^4 = 16$ ).

```
#define N ...

unsigned hash(char *ptr) {
    unsigned h = 0;
    while(*ptr) {
        h = (h<<4) + (unsigned)(*ptr++);
    }
    return (h % N);
}
```

- ▶ Existem implementações funcionais de tabelas de partição
- ▶ Em vez de *arrays* imperativos usam árvores *n*-árias (*hashed array mapped tries*)
- ▶ Em Haskell: Data.HashMap na biblioteca unordered-containers
- ▶ É uma estrutura persistente (tal como Data.Map)  
(facilita o tratamento de âmbitos)
- ▶ A função de *hash* é dada por uma instância da classe *Hashable*  
(não necessita de implementar)

- ▶ A eficiência não é crítica!
- ▶ Preferir uma solução simples que obviamente não tem erros a uma complicada em que os erros não são óbvios
- ▶ Implementar a tabela como um módulo com *interface* bem definida:
  - ▶ mais simples para garantir a correção das operações
  - ▶ pode mudar a implementação mais tarde (se tiver tempo)
- ▶ Em linguagem C: implemente usando uma pilha
- ▶ Em Haskell: implemente com Data.Map ou Data.HashMap (são igualmente simples)