

# Interpretação e Compilação de Linguagens de Programação

Mestrado Integrado em Engenharia Informática  
Departamento de Informática  
Faculdade de Ciências e Tecnologia  
Universidade Nova de Lisboa

2016-2017

João Costa Seco (joao.seco@di.fct.unl.pt)

Lecture 05(a)

Linguagens Imperativas

Programas com Estado

# Unidade 5: Linguagens Imperativas

As expressões das linguagens consideradas até agora denotam sempre **valores puros**. Até agora os identificadores denotam sempre o mesmo valor ao longo da execução de um programa. No entanto, o paradigma de programação dominante é o paradigma imperativo, caracterizado pela mutação de estado (C, Java).

- Modelo de memória (cell: set e get)
- Ambiente versus memória
- Aliasing
- L-value e R-value
- Tempo de vida vs âmbito
- Manipulação de memória por apontadores, referências, etc
- Estrutura das linguagens imperativas. Família Algol vs família ML
- Sintaxe separada de comandos e expressões
- Zonas de memória (stack/heap)
- Representação interna de valores e objectos

a

# Modelo de Memória

- Memória:  
é um conjunto (potencialmente infinito) de células cujo conteúdo é mutável.
- Cada célula de memória tem um designador único (a referência da célula) e pode conter qualquer valor da linguagem.
- As referências são valores de um tipo de dados especial ref que só podem ser usados no contexto da memória a que dizem respeito.
- Operações primitivas sobre uma memória  $\mathcal{M}$

<b>new:</b>	$\mathcal{M} \times \text{void} \rightarrow \text{ref}$
<b>set:</b>	$\mathcal{M} \times \text{ref} \times \text{Value} \rightarrow \text{void}$
<b>get:</b>	$\mathcal{M} \times \text{ref} \rightarrow \text{Value}$
<b>free:</b>	$\mathcal{M} \times \text{ref} \rightarrow \text{void}$

# Modelo de Memória

- Operações sobre uma memória  $\mathcal{M}$

**new:**  $\mathcal{M} \times \text{void} \rightarrow \text{ref}$

Devolve uma referência para uma **nova** célula livre, e define-a como estando “em uso”.

**set:**  $\mathcal{M} \times \text{ref} \times \text{Value} \rightarrow \text{void}$

Altera o conteúdo da célula referida para o valor indicado. O valor “antigo” perde-se **irremediavelmente**.

**get:**  $\mathcal{M} \times \text{ref} \rightarrow \text{Value}$

Devolve o valor contido na célula referida.

**free:**  $\mathcal{M} \times \text{ref} \rightarrow \text{void}$

Define a célula referida como estando **livre**, devolvendo-a ao gestor de memória, para ser reciclada.

# Ambiente versus Memória

- Um **ambiente** indica a denotação de cada identificador declarado num programa e reflecte a estrutura estática do programa.
- A associação estabelecida no ambiente entre um identificador e o seu valor denotado é **fixa** e **imutável** dentro do âmbito respectivo.
- A **memória** agrega o conteúdo das variáveis de estado **mutáveis**, indicando o valor contido em cada localização (ou referência).
- Uma variável de estado é visível nos programas através de identificadores.
- A associação entre o identificador de uma variável de estado e a sua localização de memória é **imutável** e é mantida pelo ambiente.

# Ambiente versus Memória

## Ambiente

Identificador	Valor
PI	3,14
x	loc <sub>0</sub>
k	loc <sub>1</sub>
j	loc <sub>1</sub>
TEN	10

## Memória

Localização	Valor
loc <sub>0</sub>	25
loc <sub>1</sub>	12
loc <sub>2</sub>	loc <sub>1</sub>
...	...
loc	0

# Ambiente versus Memória

## Ambiente

Identificador	Valor
PI	3,14
x	0x00FF
k	0x0100
j	0x0100
TEN	10

## Memória

Endereço	Valor
0x00FF	25
0x0100	12
0x0102	0x0100
...	...
0xFFFF	0



# Propriedades do modelo de memória

## Ambiente

Identificador	Valor
PI	3,14
x	loc <sub>0</sub>
k	loc <sub>1</sub>
j	loc <sub>1</sub>
TEN	10

## Memória

Localização	Valor
loc <sub>0</sub>	25
loc <sub>1</sub>	12
loc <sub>2</sub>	loc <sub>1</sub>
...	...
loc	0

Uma mesma célula de memória pode ser referida por vários identificadores distintos (**aliasing**).

# Aliasing

- Dois identificadores diferentes que referem a mesma localização de memória.

```
class A {  
    int x;  
    boolean equals(A a) { return x == a.x}  
}  
A a = new A(); a.equals(a);
```

```
int x = 0;  
void f(int* y) { *y = x+1; }  
...  
f(&x);  
// x = ?
```

# Propriedades do modelo de memória

## Ambiente

Identificador	Valor
PI	3,14
x	loc <sub>0</sub>
k	loc <sub>1</sub>
j	loc <sub>1</sub>
TEN	10

## Memória

Localização	Valor
loc <sub>0</sub>	25
loc <sub>1</sub>	12
loc <sub>2</sub>	loc <sub>1</sub>
...	...
loc	0

Uma célula pode conter uma referência para outra célula, permitindo a construção de estruturas de dados dinâmicas.

# Operações Imperativas nas linguagens

- Reserva e inicialização de uma célula nova dada uma expressão E qualquer

**var(E)**

- Pode ser encontrada de diversas formas:

```
{  
  int a;  
  MyClass m;  
  ...  
}
```

em Java tem um significado  
em C++ tem outro,  
qual a diferença?

```
new int[10];
```

```
malloc(sizeof(int));
```

```
new MyClass();
```

# Operações Imperativas nas linguagens

- Afectação de um valor a uma variável dadas as expressões E e F

**E := F**

A expressão E denota uma referência para uma célula, F é uma expressão qualquer

- Pode ser encontrada de diversas formas:

**a = 1**

**i := 2**

**b[x+2][b[x-2]] = 2**

**\*(p+2) = y**

**myTable(i,j) = myTable(j,i)**

**Readln(MyLine);**

# Operações Imperativas nas linguagens

- Desreferenciação de uma célula de memória dada uma expressão  $E$  que denota uma referência para uma célula.

**!E**

- Pode ser encontrada de diversas formas:

**$i := !i + 1$  (linguagem ML)**

**$*p$  (linguagem C)**

**$i = i + 1$  (linguagem C)**

**$i++$  (linguagem C)**

# Operações Imperativas nas linguagens

- Desreferenciação de uma célula de memória dada uma expressão  $E$  que denota uma referência para uma célula.

**!E**

- Pode ser encontrada de diversas formas:

$i := !i + 1$  (linguagem ML)

$*p$  (linguagem C)

$\textcircled{i} = i + 1$  (linguagem C)

$\textcircled{i}++$  referências (linguagem C)

# Operações Imperativas nas linguagens

- Desreferenciação de uma célula de memória dada uma expressão E que denota uma referência para uma célula.

**!E**

- Pode ser encontrada de diversas formas:

**i := !i + 1** (linguagem ML)

**\*p** (linguagem C)

**i =  + 1** (linguagem C)

**i++** **valor** (linguagem C)



# L-Value e R-Value

- Se uma expressão E tem por valor uma referência, a maior parte das linguagens de programação interpreta E de forma **dependente do contexto**

**E := 2**

- (**Left-Value**) À “esquerda” do símbolo de afectação, denota o seu valor efectivo (que é uma referência)

**E := E + 1**

- (**Right-Value**) À “direita” do símbolo de afectação, denota o **conteúdo** da célula referida, evitando-se escrever a desreferenciação explícita

**E := !E + 1**

# L-Value e R-Value

- Se uma expressão E tem por valor uma referência, a maior parte das linguagens de programação interpreta E de forma **dependente do contexto**.

$$A[A[2]] := A[2] + 1$$

- A terminologia “L-Value” e “R-Value” não é muito feliz. Por exemplo, na expressão acima as duas subexpressões da forma  $A[2]$ , uma à esquerda e outra à direita, são ambas desreferenciadas implicitamente.

# Desreferenciação

- A operação de desreferenciação !E torna a interpretação dos programas mais precisa e evita qualquer ambiguidade.

$$A[!A[2]] := !A[2] + 1$$

- Por outro lado, pode argumentar-se que torna os programas mais difíceis de ler.

A desreferenciação implícita pode ser vista como uma operação de **coerção** (conversão ou cast).



# Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

<b>var( E )</b>	instanciação
<b>free( E )</b>	libertação
<b>E := E</b>	afecção
<b>! E</b>	desreferenciação

```
{  
/* linguagem C */  
  
  const int k = 2;  
  int a = k;  
  int b = a + 2;  
  ...  
  b = a * b  
  ...  
}
```

```
decl  
  k = 2  
  a = var(k)  
  b = var(!a+2)  
in  
  ...  
  b := !a * !b  
  ...  
  free(a);  
  free(b)  
end
```

# Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

**var( E )**

instanciação

**free( E )**

libertação

**E := E**

afecção

**! E**

desreferenciação

```
{  
/* linguagem C */  
  
    const int k = 2;  
    int a = k;  
    int b = a + 2;  
    ...  
    b = a * b  
    ...  
}
```

```
decl  
    k = 2  
    a = var(k)  
    b = var(!a+2)  
in  
    ...  
    b := !a * !b  
    ...  
    free(a);  
    free(b)  
end
```

libertação implícita (das células atribuídas aos ids a e b)

# Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

<b>var( E )</b>	instanciação
<b>free( E )</b>	libertação
<b>E := E</b>	afecção
<b>! E</b>	desreferenciação

```
{  
/* linguagem C */  
  
  int k = 2;  
  int *a = &k;  
  ... *a = k+*a ...  
}
```

```
decl  
  k = var(2)  
  a = var(k)  
in  
  ... !a := !k+!!a ...  
  free(k);  
  free(a)  
end
```

# Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

<b>var( E )</b>	instanciação
<b>free( E )</b>	libertação
<b>E := E</b>	afecção
<b>! E</b>	desreferenciação

```
{  
/* linguagem C */  
  
  int k = 2;  
  const int *a = &k;  
  int b = *a;  
  ... *a = k+b ...  
}
```

```
decl  
  k = var(2)  
  a = k  
  b = var(!a)  
in  
  ... a := !k+!b ...  
  free(k);  
  free(b)  
end
```



# Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

<b>var( E )</b>	instanciação
<b>free( E )</b>	libertação
<b>E := E</b>	afecção
<b>! E</b>	desreferenciação

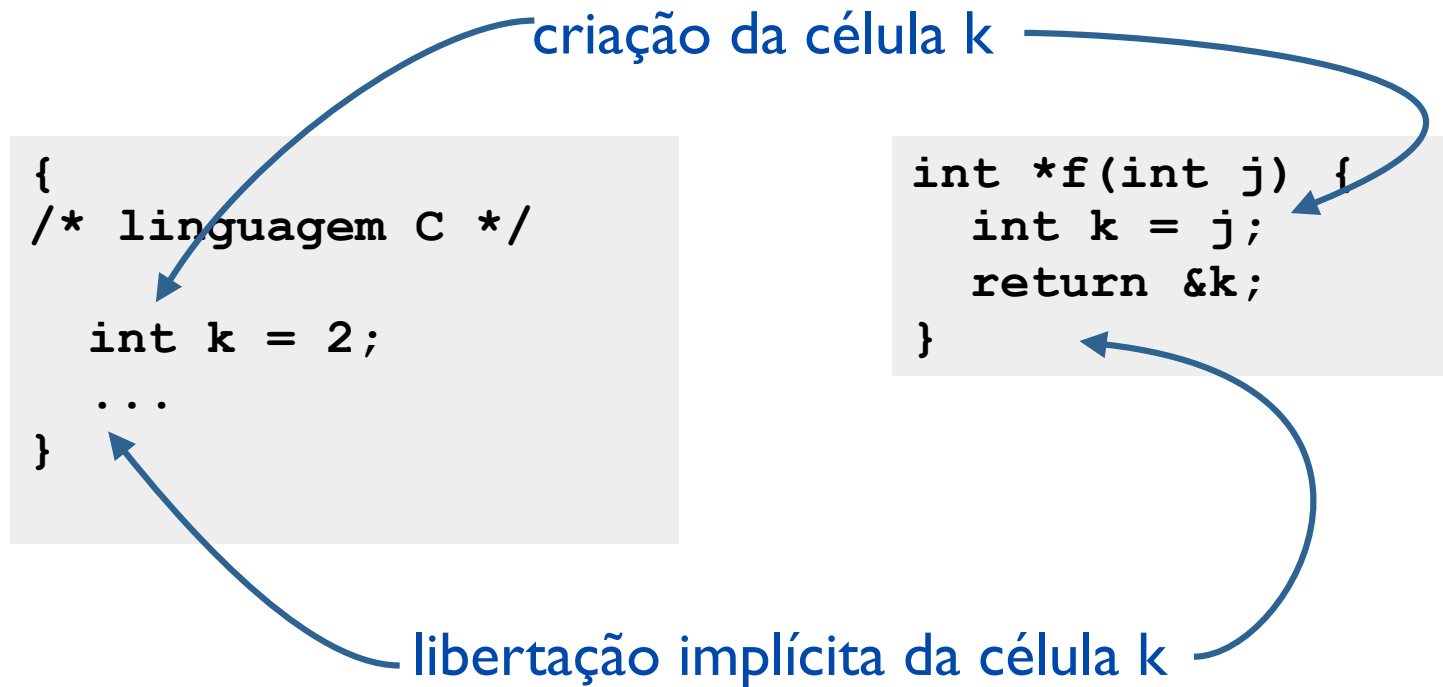
```
{  
/* linguagem C */  
  
  int k = 2;  
  int *a = &k;  
  ... k = k+*a ...  
}
```

```
decl  
  k = var(2)  
  a = var(k)  
in  
  ... k := !k+!!a ...  
  free(k);  
  free(a);  
end
```

# Tempo de Vida (de uma célula)

O **tempo de vida** de uma célula é o tempo que medeia entre a sua criação / reserva usando **var( )** e a sua libertação usando **free( )**.

- Em muitas situações, o tempo de vida da célula **coincide** com o âmbito do(s) seu(s) identificador.



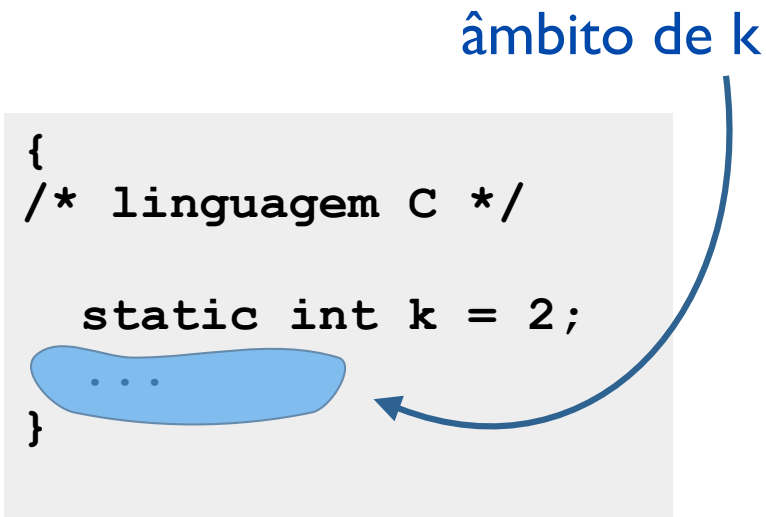
# Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **var( )** e a sua libertação usando **free( )**.

- Noutras situações, o tempo de vida da célula **extravasa** o âmbito do(s) seu(s) identificador.

âmbito de k

```
{  
/* linguagem C */  
  
static int k = 2;  
...  
}
```

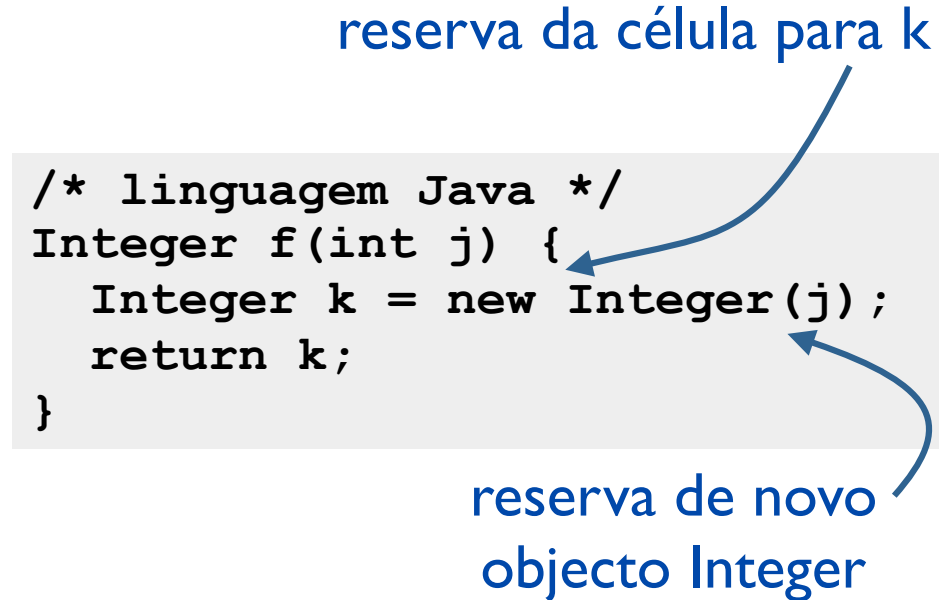


O tempo de vida da célula associada a k é o tempo do programa

reserva da célula para k

```
/* linguagem Java */  
Integer f(int j) {  
    Integer k = new Integer(j);  
    return k;  
}
```

reserva de novo objecto Integer



há libertação implícita da célula de k mas o objecto sobrevive ao bloco!

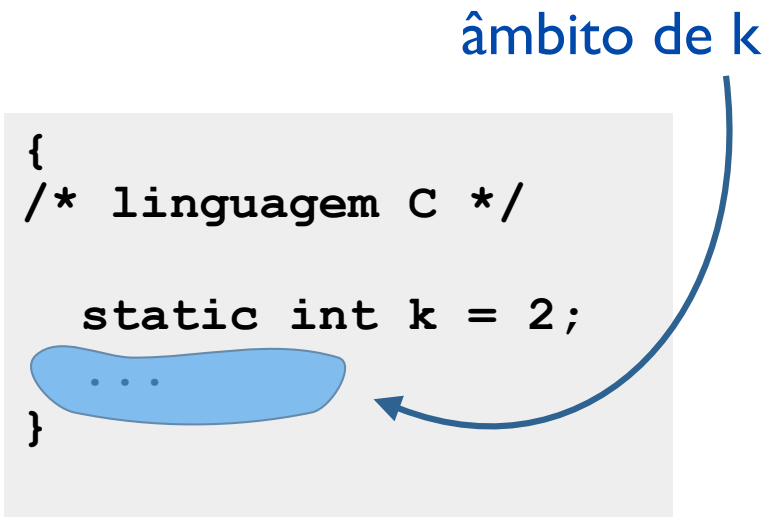
# Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **var( )** e a sua libertação usando **free( )**.

- Noutras situações, o tempo de vida da célula **extravasa** o âmbito do(s) seu(s) identificador.

âmbito de k

```
{  
/* linguagem C */  
  
static int k = 2;  
...  
}
```

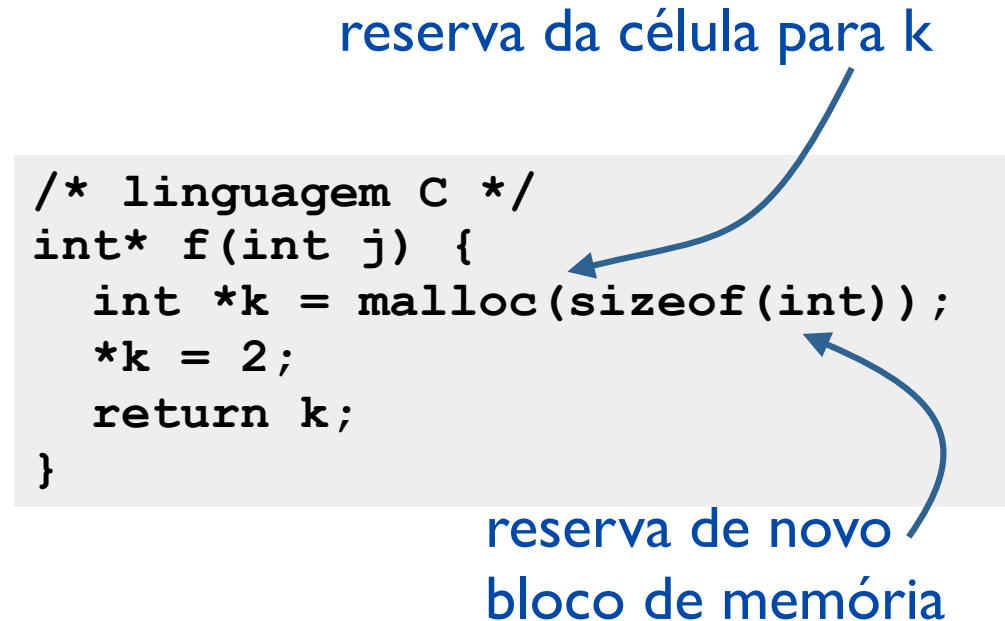


O tempo de vida da célula associada a k é o tempo do programa

reserva da célula para k

```
/* linguagem C */  
int* f(int j) {  
    int *k = malloc(sizeof(int));  
    *k = 2;  
    return k;  
}
```

reserva de novo bloco de memória



há libertação implícita da célula de k mas a memória reservada perdura.

# Linguagens Imperativas

## Linguagens da família do **ALGOL** (Pascal, C, ...)

Assumem como princípio de desenho uma separação muito clara, logo ao nível sintático, entre **expressões** e **comandos**

- Expressões:

Denotam valores puros (inteiros, booleanos, funções)

A avaliação de expressões não deve ter efeitos (laterais)

- Comandos:

Denotam efeitos (na memória)

Um comando é executado pelo efeito que produz na memória: representa uma **acção**.

# Uma linguagem de tipo ALGOL

- Definida com base em duas categorias sintáticas: Expressões (EXP) e comandos (COM):

<b>num:</b>	$\text{Integer} \rightarrow \text{EXP}$
<b>bool:</b>	$\text{Boolean} \rightarrow \text{EXP}$
<b>id:</b>	$\text{String} \rightarrow \text{EXP}$
<b>add:</b>	$\text{EXP} \times \text{EXP} \rightarrow \text{EXP}$
<b>and:</b>	$\text{EXP} \times \text{EXP} \rightarrow \text{EXP}$
<b>if:</b>	$\text{EXP} \times \text{COM} \times \text{COM} \rightarrow \text{COM}$
<b>while:</b>	$\text{EXP} \times \text{COM} \rightarrow \text{COM}$
<b>assign:</b>	$\text{EXP} \times \text{EXP} \rightarrow \text{COM}$
<b>seq:</b>	$\text{COM} \times \text{COM} \rightarrow \text{COM}$
<b>var:</b>	$\text{String} \times \text{EXP} \times \text{COM} \rightarrow \text{COM}$
<b>const:</b>	$\text{String} \times \text{EXP} \times \text{COM} \rightarrow \text{COM}$

# Linguagens Imperativas

## Linguagens da família do ML

Todas as construções pertencem a uma única categoria sintática, de **expressões**.

- Nestas linguagens, qualquer expressão pode potencialmente produzir um efeito lateral...
- Por exemplo, em OCAML a afectação  $x := E$  é uma expressão (de tipo **unit** (a.k.a. **void**)).
- N.B. Existem linguagens que combinam conceitos! Por exemplo, a linguagem C, contém expressões e comandos: a afectação  $(x = y)$  é uma expressão, e expressões podem produzir efeitos  $(i++)$ .

# Uma linguagem tipo ML (microML)

- Consideramos uma só categoria sintáctica para expressões (EXP):

<b>num:</b>	$\text{Integer} \rightarrow \text{EXP}$	
<b>bool:</b>	$\text{Boolean} \rightarrow \text{EXP}$	
<b>id:</b>	$\text{String} \rightarrow \text{EXP}$	
<b>add:</b>	$\text{EXP} \times \text{EXP} \rightarrow \text{EXP}$	
<b>var:</b>	$\text{EXP} \rightarrow \text{EXP}$	
<b>deref:</b>	$\text{EXP} \rightarrow \text{EXP}$	( !x )
<b>if:</b>	$\text{EXP} \times \text{EXP} \times \text{EXP} \rightarrow \text{EXP}$	
<b>while:</b>	$\text{EXP} \times \text{EXP} \rightarrow \text{EXP}$	
<b>assign:</b>	$\text{EXP} \times \text{EXP} \rightarrow \text{EXP}$	( x := y + z )
<b>seq:</b>	$\text{EXP} \times \text{EXP} \rightarrow \text{EXP}$	( S1 ; S2 )
<b>decl:</b>	$\text{String} \times \text{EXP} \times \text{EXP} \rightarrow \text{EXP}$	



# Semântica de microML

A semântica de uma linguagem imperativa pode ser caracterizada por uma função  $I$  que dá uma denotação a todos os programas abertos dado um ambiente e uma memória.

$$I : P \times ENV \times MEM \rightarrow VAL \times MEM$$

$P$  = Fragmentos de programa (abertos)

$ENV$  = Ambientes (funções  $ID \rightarrow VAL$ )

$MEM$  = Memórias

$VAL$  = Valores (Denotações)

O conjunto das denotações possíveis:

$$Val = Boolean \cup Integer \cup Ref$$

Esta função traduz a intuição que, em geral, um fragmento de programa  $P$  produz um **valor** e gera um **efeito** (na memória).

# Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$\text{eval} : \text{microML} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$

$\text{eval}(\text{add}(E1, E2), \text{env}, m0) \triangleq [ (v1, m1) = \text{eval}(E1, \text{env}, m0);$   
 $(v2, m2) = \text{eval}(E2, \text{env}, m1);$   
 $(v1 + v2, m2) ]$

$\text{eval}(\text{and}(E1, E2), \text{env}, m0) \triangleq [ (v1, m1) = \text{eval}(E1, \text{env}, m0);$   
 $(v2, m2) = \text{eval}(E2, \text{env}, m1);$   
 $(v1 \& v2, m2) ]$

# Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$\text{eval} : \text{microML} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$

$\text{eval}(\text{var}(E), \text{env}, m_0) \triangleq [ (v_1, m_1) = \text{eval}(E, \text{env}, m_0);$

$(\text{ref}, m_2) = m_1.\text{new}(v_1);$

$(\text{ref}, m_2) ]$

$\text{eval}(\text{deref}(E), \text{env}, m_0) \triangleq [ (\text{ref}, m_1) = \text{eval}(E, \text{env}, m_0);$

$(m_1.\text{get}(\text{ref}), m_1) ]$

$\text{eval}(\text{assign}(E_1, E_2), \text{env}, m_0) \triangleq [(v_1, m_1) = \text{eval}(E_1, \text{env}, m_0);$

$(v_2, m_2) = \text{eval}(E_2, \text{env}, m_1);$

$m_3 = m_2.\text{set}(v_1, v_2);$

$(v_2, m_3) ]$

# Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$\text{eval} : \text{microML} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

```
eval( seq(E1, E2) , env , m0)  $\triangleq$  [(v1 , m1) = eval( E1, env, m0);  

                (v2 , m2) = eval( E2, env, m1);  

                (v2 , m2) ]
```

```
eval( if(E1, E2, E3) , env , m0)  $\triangleq$   
[(v1 , m1) = eval( E1, env, m0);  
  if (v1 = T) then (v2 , m2) = eval( E2, env, m1);  
    else (v2 , m2) = eval( E3, env, m1);  
(v2 , m2) ]
```

# Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$\text{eval} : \text{microML} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$

$\text{eval}(\text{ while}(E1, E2), \text{env}, m0) \triangleq$

$[(v1, m1) = \text{eval}(E1, \text{env}, m0);$

$\text{if } (v1 = T) \text{ then } [(v2, m2) = \text{eval}(E2, \text{env}, m1);$

$(v, m1) = \text{eval}(\text{ while}(E1, E2), m2) ]$

$\text{else } (F, m1) ]$

**iteração interpretada em  
termos de recursão.**

# Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$\text{eval} : \text{microML} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$

```
eval( decl(s, EI, EB) , env , m0)  $\triangleq$   
    [(v1 , m1) = eval( EI, env, m0 );  
     env = env.BeginScope();  
     env.Assoc(s, v1);  
     (v2 , m2) = eval(EB, env, m1);  
     env = env.EndScope();  
     (v2 , m2) ]
```

# Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$\text{eval} : \text{microML} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$

```
decl a = var(2) in
decl b = var(!a) in
decl c = a in
(
  a := !b + 2;
  c := !c + 2
)
```

**a e c são aliases (sinónimos), ou seja, referem a mesma célula de memória.**





# Interpretação e Compilação de Linguagens de Programação

Mestrado Integrado em Engenharia Informática  
Departamento de Informática  
Faculdade de Ciências e Tecnologia  
Universidade Nova de Lisboa

2016-2017

João Costa Seco (joao.seco@di.fct.unl.pt)

# Lecture 05(b)

## Sistemas de tipos

# Unidade 6: Sistemas de tipos

Os sistemas de tipos são ferramentas de análise estática que garantem boas propriedades de programas concretos. A análise estática é uma forma de interpretação “abstracta” de programas. A principal característica da análise estática é que termina sempre, mesmo para programas que não terminam.

A forma mais comum de análise estática é a verificação de tipos (**type checking**). Os sistemas de tipos garantem a ausência de certos tipos de erros durante a execução.

- Erros de execução
- Interpretação abstracta
- Sistemas de tipos
- Consistência de um sistema de tipos
- Detecção de erros de execução

# Erros de execução...

- O que é que pode correr mal na execução de um programa?

```
decl
  a = newvar(0)
  b = newvar(2)
  c = newvar(a > b)
in
  if c then
    !a := !a + 1;
    c := 1 < !c
  end
```

# Erros de execução...

- O que é que pode correr mal na execução de um programa?

```
decl
  a = newvar(0)
  b = newvar(2)
in
  decl
    c = newvar(!a > !b)
  in
    if !c then
      a := !a + 1;
      c := 1 < !a
    end
  end
end
```

# Erros de execução...

- O que é que pode correr mal na execução de um programa?

```
eval( add(E1, E2) , env , m0)  $\triangleq$  [ (v1 , m1) = eval( E1, env, m0);  
                                         (v2 , m2) = eval( E2, env, m1);  
                                         (v1 + v2 , m2) ]
```

```
eval( deref(E) , env , m0)  $\triangleq$  [ (ref , m1) = eval( E, env, m0);  
                                   (m1.get(ref) , m1) ]
```

```
eval( assign(E1, E2) , env , m0)  $\triangleq$  [(v1 , m1) = eval( E1, env, m0);
```

É impossível em tempo de compilação calcular todos os valores possíveis e assim evitar os erros de execução. Os domínios são infinitos...

```
(v2 , m2) = eval( E2, env, m1);  
m3 = m2.set(v1, v2);  
(v1 , m3) ]
```

Podemos interpretar os programas aglomerando conjuntos infinitos de valores num único representante, um “tipo”.

# Funções Semânticas

- Algoritmo **eval** para calcular o valor de uma expressão qualquer da linguagem microML:

$$\text{eval} : \text{microML} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

- Algoritmo **comp** para gerar um programa na linguagem CIL equivalente a uma dada expressão da linguagem microML.

$$\text{comp} : \text{microML} \times \text{ENV} \rightarrow \text{CodeSeq}$$

# Funções Semânticas (mais uma)

- Algoritmo **eval** para calcular o valor de uma expressão qualquer da linguagem microML:

$$\text{eval} : \text{microML} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

- Algoritmo **comp** para gerar um programa na linguagem CIL equivalente a uma dada expressão da linguagem microML.

$$\text{comp} : \text{microML} \times \text{ENV} \rightarrow \text{CodeSeq}$$

- Algoritmo **typecheck** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$\text{typecheck} : \text{microML} \times \text{ENV} \rightarrow \text{TYPE}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\}, \text{none} \}$$



# Tipos para microML

- Algoritmo **typecheck** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$\text{typechk} : \text{microML} \times \text{ENV} \rightarrow \text{TYPE}$$
$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$
$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\}, \text{none} \}$$

- A função **typecheck** pode ser vista como um interpretador que avalia um programa de acordo com uma semântica especial, mais abstracta, (em que os “valores” são “tipos”).
- Nesta semântica, as operações da linguagem estão definidas com tipos como operandos e tipos como resultado.

# Tipos para microML

- Algoritmo **typecheck** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$\text{typecheck} : \text{microML} \times \text{ENV} \rightarrow \text{TYPE}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\}, \text{none} \}$$

**int**: é o tipo dos valores inteiros.

**bool**: é o tipo dos valores booleanos.

**ref** $\{\mathcal{T}\}$ : é o tipo das referências para células que só podem conter valores de tipo  $\mathcal{T}$ .

**Exemplo**: **ref** $\{\text{ref}\{\text{int}\}\}$  é o tipo das referências para células que só podem conter referências para (células) com valores inteiros.

**none**: é o “tipo” dos programas para os quais as funções semânticas **eval** e **comp** estão indefinidas.

# Tipificação de microML

- Algoritmo **typecheck** para calcular o tipo de uma expressão qualquer da linguagem microML:

**typecheck** : microML  $\times$  ENV  $\rightarrow$  TYPE

Se **E** for da forma **add**(E1, E2) e

se E1 for do tipo **int** num dado ambiente *env* e

se E2 for do tipo **int** num dado ambiente *env*

então o tipo de **E** é **int**

e se uma destas condições falhar?

então a execução da expressão não estaria definida...

senão, o tipo de **E** é **none**

# Tipificação de microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

**typecheck** : microML  $\times$  ENV  $\rightarrow$  TYPE

```
typecheck( add(E1, E2) , env )  $\triangleq$  [ t1 = typecheck ( E1, env )  
                                         t2 = typecheck ( E2, env )  
                                         if ( t1 == int ) and ( t2 == int )  
                                         then int  
                                         else none ]
```

# Tipificação de microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

**typecheck** : microML  $\times$  ENV  $\rightarrow$  TYPE

Todas as expressões num(*n*) denotam valores inteiros... o representante dos inteiros é **int**

**typecheck**( num(*n*) , *env*)  $\triangleq$  **int** ;

**typecheck**( id(*s*) , *env*)  $\triangleq$  *env*.Find(*s*) ;

**typecheck**( true , *env*)  $\triangleq$  **bool** ;

**typecheck**( false , *env*)  $\triangleq$  **bool** ;

# Tipificação de microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

**typecheck** : microML  $\times$  ENV  $\rightarrow$  TYPE

```
typecheck( and(E1, E2) , env )  $\triangleq$  [ t1 = typecheck ( E1, env )  
                                         t2 = typecheck ( E2, env )  
                                         if ( t1 == bool ) and ( t2 == bool )  
                                         then bool  
                                         else none ]
```

# Tipificação de microML

- Algoritmo **typecheck** para calcular o tipo de uma expressão qualquer da linguagem microML:

**typechk** : microML  $\times$  ENV  $\rightarrow$  TYPE

```
typecheck( assign(E1, E2) , env )  $\triangleq$   
    [ t1 = typecheck( E1, env );  
      t2 = typecheck( E2, env );  
      if ( t1 == ref{t2} )  
        then t2;  
      else none ; ]
```

# Tipificação de microML

- Algoritmo **typecheck** para calcular o tipo de uma expressão qualquer da linguagem microML:

**typecheck** : microML  $\times$  ENV  $\rightarrow$  TYPE

```
typecheck( if(E1, E2, E3) , env )  $\triangleq$   
  [ t1 = typecheck( E1, env );  
    if ( t1 != bool ) then none  
    else [ t2 = typecheck( E2, env );  
           t3 = typecheck( E3, env );  
           if ( t2 == none ) or ( t3 == none ) or ( t2 != t3 )  
           then none  
           else t2 ] ]
```



# Tipificação de microML

- Compare com o caso **if** do algoritmo **eval**.

Os “ramos” E2 e E3 são **ambos** analisados.  
Impõe-se (como restrição)  $t2 == t3$  . [Porquê?]

```
typecheck( if(E1, E2, E3) , env )  $\triangleq$   
  [ t1 = typecheck( E1, env );  
    if ( t1 != bool ) then none  
    else [ t2 = typecheck( E2, env );  
           t3 = typecheck( E3, env );  
           if ( t2 == none ) or ( t3 == none ) or ( t2 != t3 )  
           then none  
           else t2 ] ]
```

# Tipificação de microML

- Algoritmo **typecheck** para calcular o tipo de uma expressão qualquer da linguagem microML:

**typecheck** : microML  $\times$  ENV  $\rightarrow$  TYPE

```
typecheck( while(E1, E2) , env )  $\triangleq$   
  [ t1 = typecheck( E1, env );  
    if ( t1 != bool ) then none  
    else [ t2 = typecheck( E2, env );  
           if ( t2 == none ) then none  
           else bool ] ]
```

# Tipificação de microML

- Compare com o caso **while** do algoritmo **eval**.

O “ramo” E2 é sempre analisado exactamente **uma** vez.  
Impõe-se como tipo o tipo **bool**. [Porquê?]

```
typecheck( while(E1, E2) , env )  $\triangleq$   
  [ t1 = typecheck( E1, env );  
    if ( t1 != bool ) then none  
    else [ t2 = typecheck( E2, env );  
           if ( t2 == none ) then none  
           else bool ] ]
```

# Tipificação de microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

**typecheck** : microML  $\times$  ENV  $\rightarrow$  TYPE

```
typecheck( decl(s, E1, E2) , env )  $\triangleq$   
    [ t1 = typecheck( E1, env);  
      envlocal = env.BeginScope();  
      envlocal.Assoc(s, t1);  
      t2 = typecheck(E2, envlocal);  
      env = envlocal.EndScope(); t2 ]
```

# Correcção da Tipificação

- Podemos verificar que, para todo o programa microML  $P$ , e ambiente *env* que cubra todos os identificadores livres de  $P$ , a operação

**typecheck**( $P$ , *env*)

está bem definida e termina sempre [Porquê?]

- Podemos também demonstrar o seguinte teorema, que relaciona a tipificação com a avaliação de programas microML. [Como?]

**Teorema:** Para todo o programa microML  $P$  e tipo  $\mathcal{T}$ ,

Se **typchk**( $P, \emptyset$ ) =  $\mathcal{T}$  e **eval**( $P, \emptyset, \emptyset$ ) =  $v$  então  $v \in \mathcal{T}$ .

# Correcção da Tipificação

**Teorema:** Para todo o programa `microML`  $P$  e tipo  $\mathcal{T}$ ,  
Se **typchk**( $P, \emptyset$ ) =  $\mathcal{T}$  e **eval**( $P, \emptyset, \emptyset$ ) =  $v$  então  $v \in \mathcal{T}$ .

Em particular,  
se **typchk**( $P, \emptyset$ )  $\neq$  **none** então **eval**( $P, \emptyset, \emptyset$ )  $\neq$  **error**.

Ou seja:

ou  $P$  não termina,

ou  $P$  termina e **não incorre** em erros de execução.

*“Well-typed programs don’t go wrong”* (Robin Milner)

# Correcção da Tipificação

**Teorema:** Para todo o programa microML  $P$  e tipo  $\tau$ ,  
Se **typecheck**( $P, \emptyset$ ) =  $\tau$  e **eval**( $P, \emptyset, \emptyset$ ) =  $v$  então  $v \in \tau$ .

Esta propriedade é em geral conhecida como a  
“consistência do sistema de tipos”. Garante que

*“Well-typed programs don’t go wrong”* (Robin Milner)

Existem muitos programas  $P$  tais que **eval**( $P, \emptyset, \emptyset$ ) =  $v$  e  $v \in \text{int}$  mas  
(infelizmente) **typecheck**( $P, \emptyset$ ) = error ! [Exemplo?]

# Robin Milner (1934-2010)

## ACM Turing Award (1991)

For three distinct and complete achievements:

- 1) LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;
- 2) ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;
- 3) CCS, a general theory of concurrency. In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.





1. O programa está bem tipificado? se não, apresente uma versão correcta.
2. Qual o ambiente de tipificação da expressão **!y** ?
3. Qual o resultado/efeito do programa?

```
decl
  x = 10
  y = var(0)
in
  decl
    z = var(y)
    w = var(false)
  in
    while w do
      w := ((!z := !!z + y + 1) < x)
    end; !y
  end
end
```