

Interpretação e Compilação (de Linguagens de Programação)

Unid. 7A : Esquemas de Compilação

Nesta edição da disciplina, vamos considerar a geração de código para a JVM.

Arquitectura da JVM, estruturas principais

Evaluation Stack (ES)

Guarda os argumentos e resultados intermédios das operações

Call Stack (CS)

Guarda os registos de ativação das funções (métodos)

Heap (H)

Memória gerida automaticamente (garbage collected)

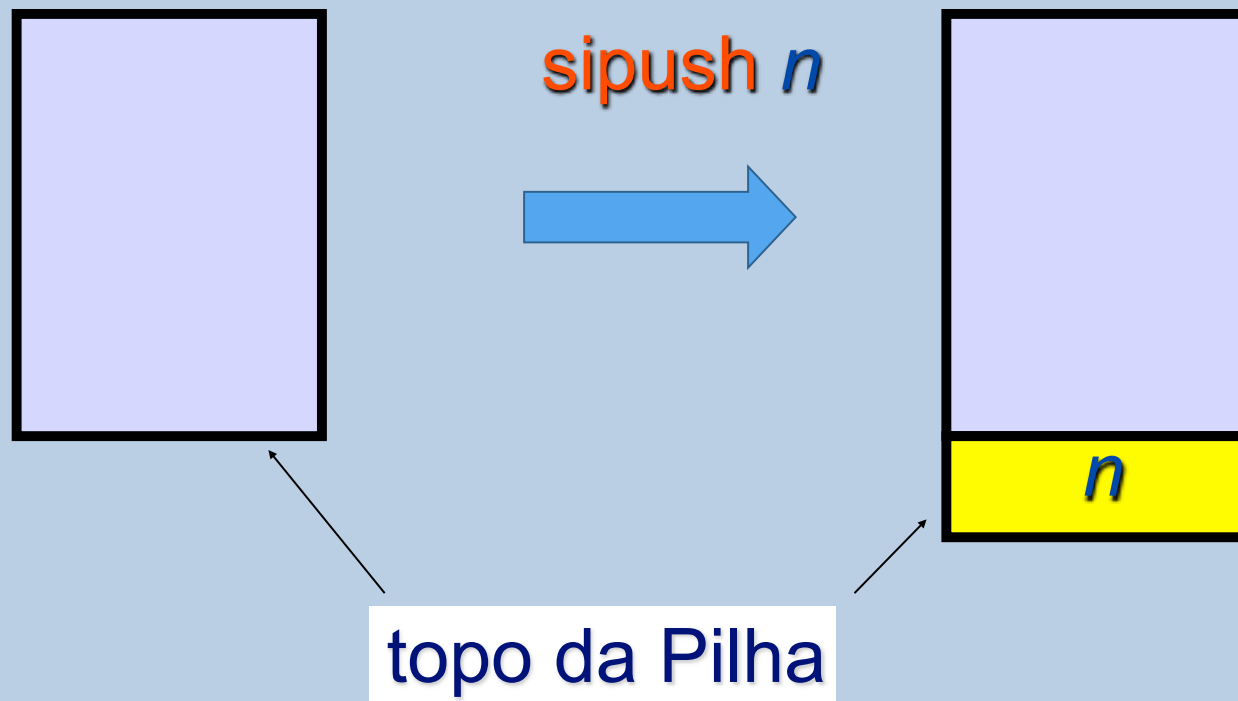
Guarda os objectos

Java Virtual Machine (recap)

- É uma máquina de pilha: todas as instruções consomem argumentos do topo da ES, e deixam um resultado no topo da ES
- “Primeiras” (5) instruções da JVM:
 - **sipush** *n* : Carrega o valor *n* (short integer) no topo do ES
 - **iadd** : Retira dois valores inteiros do topo do ES e coloca no ES a sua soma
 - **imul** : idem para a multiplicação
 - **idiv** : idem para a divisão
 - **isub** : idem para a subtracção

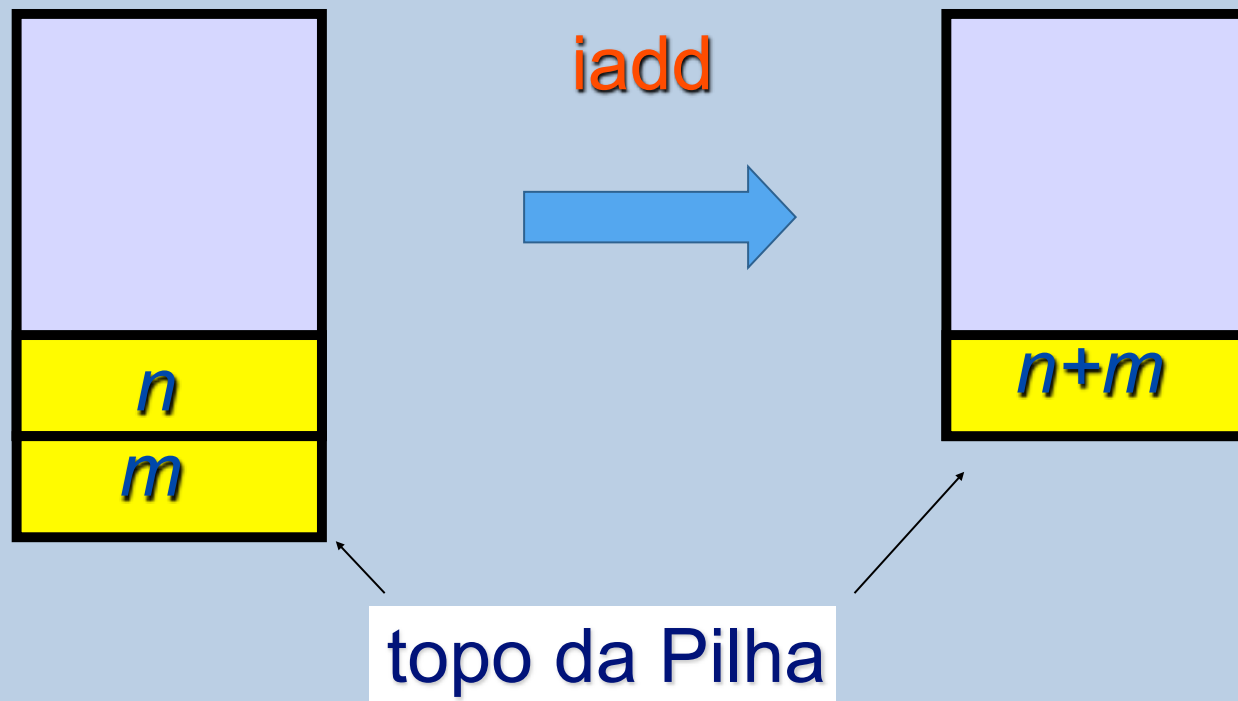
Java Virtual Machine

- “Primeiras” (5) instruções: `sipush n`, `iadd`, `mul`, `idiv`, `isub`.
- Load Constant (`sipush n`)



Java Virtual Machine

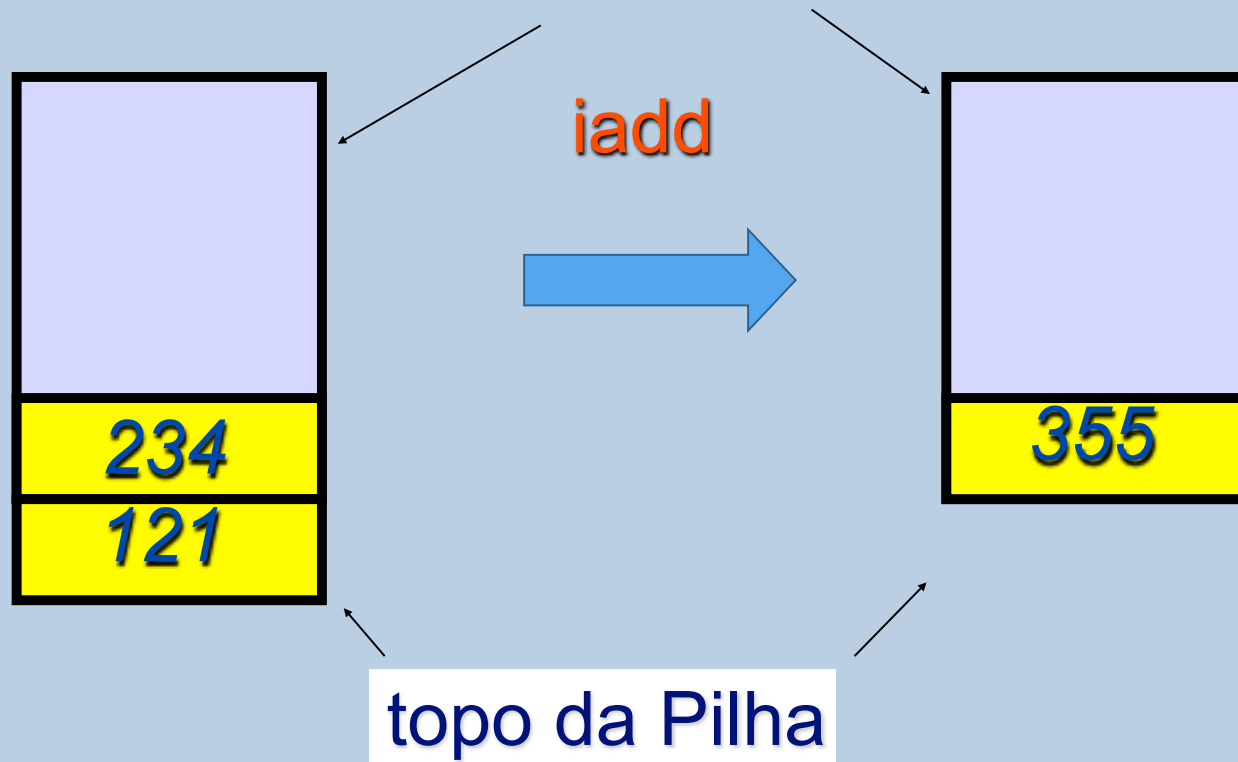
- “Primeiras” (5) instruções: `ipush n`, `iadd`, `mul`, `idiv`, `isub`.
- Add (`iadd`)



Java Virtual Machine

- “Primeiras” (5) instruções: `sipush n`, `iadd`, `mul`, `idiv`, `isub`.
- `Add (iadd)`

O fundo da pilha é inalterado!



Compilador de CALC

- Algoritmo $\text{comp}(E)$ para traduzir uma expressão E qualquer de CALC numa sequência de instruções JVM

$\text{comp} : \text{CALC} \rightarrow \text{CodeSeq}$

se E é da forma num (n):	$\text{comp}(E) \triangleq < \text{sipush } n >$
se E é da forma add (E', E''):	$s1 = \text{comp}(E'); s2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{iadd} >$
se E é da forma mul (E', E''):	$v1 = \text{comp}(E'); v2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{imul} >$
se E é da forma sub (E', E''):	$v1 = \text{comp}(E'); v2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{isub} >$
se E é da forma div (E', E''):	$v1 = \text{comp}(E'); v2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{idiv} >$

Compilador de CALC

- Algoritmo $\text{comp}(E)$ para traduzir uma expressão E qualquer de CALC numa sequência de instruções JVM

$\text{comp} : \text{CALC} \rightarrow \text{CodeSeq}$

$\text{comp}(\text{num}(n)) \triangleq < \text{sipush } n >$

$\text{comp}(\text{add}(E', E'')) \triangleq \text{comp}(E') @ \text{comp}(E'') @ < \text{iadd} >$

$\text{comp}(\text{mul}(E', E'')) \triangleq \text{comp}(E') @ \text{comp}(E'') @ < \text{imul} >$

$\text{comp}(\text{sub}(E', E'')) \triangleq \text{comp}(E') @ \text{comp}(E'') @ < \text{isub} >$

$\text{comp}(\text{div}(E', E'')) \triangleq \text{comp}(E') @ \text{comp}(E'') @ < \text{idiv} >$

Correcção do Compilador

- Algoritmo $\text{comp}(E)$ para traduzir uma expressão E qualquer da linguagem CALC numa sequência de instruções da linguagem JVM

$\text{comp} : \text{CALC} \rightarrow \text{CodeSeq}$

- **Propriedade de Correcção:** Quando a sequência de instruções $\text{comp}(E)$ é executada num estado da máquina virtual em que o ES está no estado p , quando termina deixa sempre a máquina no estado $\text{push}(v, p)$, em que v é o valor da expressão E .

Java Virtual Machine

- “Primeiras” (5) instruções: **sipush** *n*, **iadd**, **imul**, **idiv**, **isub**.
- **Comp**(“2+2*(7-2)”)
- **Comp**(**add**(**num**(2),**mul**(**num**(2),**sub**(**num**(7),**num**(2))))))

```
sipush 2  
sipush 2  
sipush 7  
sipush 2  
isub  
imul  
iadd
```

Java Virtual Machine

Toda a informação necessária sobre a JVM, incluindo em particular a lista de instruções, pode ser encontrada em:

<https://docs.oracle.com/javase/specs/jvms/se7/html/index.html>

The Java® Virtual Machine Specification

Java SE 7 Edition

Tim Lindholm

Frank Yellin

Gilad Bracha

Alex Buckley

2013-02-28

Compilação da Linguagem Core

Um ambiente de compilação D guarda para cada identificador x livre no programa a compilar a seguinte informação:

$D(x, \text{type})$ = o tipo do identificador x (determinado pelo typechecker)

$D(x, \text{level})$ = o número de níveis de ambiente a percorrer até se atingir x

Se x estiver declarado no nível corrente (mais interno), $D(x, \text{level})=0$.

Se x estiver declarado no nível a seguir, $D(x, \text{level})=1$, etc.

Nesta apresentação, dada uma expressão E da nossa linguagem e D um ambiente representamos por $[[E]]$ a sequência de instruções geradas para E no ambiente D .

Compilação da Linguagem Core

Compilação de expressões aritméticas (exemplos)

$[[E1 + E2]]\mathcal{D} =$

$[[E1]]\mathcal{D}$

$[[E2]]\mathcal{D}$

iadd

$[[E1 * E2]]\mathcal{D} =$

$[[E1]]\mathcal{D}$

$[[E2]]\mathcal{D}$

imul

Compilação da Linguagem Core

Compilação de expressões lógicas (representando booleanos como inteiros 1,0)

$[[E1 \succ E2]]\mathcal{D} =$

$[[E1]]\mathcal{D}$

$[[E2]]\mathcal{D}$

isub

ifgt L1

sipush 0

goto L2

L1: sipush 1

L2:

Compilação da Linguagem Core

Compilação de expressões lógicas (representando booleanos como inteiros 1,0)

`[[ E1 && E2 ]]D =`

`[[E1]]D`

`[[E2]]D`

`land`

`[[ E1 || E2 ]]D =`

`[[E1]]D`

`[[E2]]D`

`lor`

Compilação da Linguagem Core

Compilação de expressão condicional

$[[E1 \text{ ? } E2 : E3]]\mathcal{D} =$

$[[E1]]\mathcal{D}$

ifeq L1

$[[E2]]\mathcal{D}$

goto L2

L1: $[[E3]]\mathcal{D}$

L2:

Compilação da Linguagem Core

Compilação de declarações e identificadores

Cada bloco de declarações

decl

$x_1 = e_1 \dots$

$x_n = e_n$

in E end

vai (quando executado em runtime) necessitar de criar uma stack frame (no heap) para guardar o valor dos n identificadores $x_1 \dots x_n$

Assume-se que em cada momento é conhecida a localização SL (em variável local) do static link corrente, assim como o seu tipo JVM (Lcurrframetype)

O SL corrente contém uma referência para a stack frame, representada na JVM por um objeto no heap.

Compilação da Linguagem Core

Compilação de declarações e identificadores

Estrutura genérica de uma stack frame (usando syntax Jasmin)

```
.class          frame_id
.super          java/lang/Object
.field          public sl Lancestor_frame_id;
.field          public x_0 type
.field          public x_1 type;
..
.field          public x_1 type;
.end method
```

frame_id : será o nome da "classe" JVM que implementa a frame

ancestor_frame_id : será o nome da classe JVM que implementa a frame do nível lexical anterior (se existir), acessível através do campo sl desta frame

Compilação da Linguagem Core

Compilação de declarações e identificadores

Assume-se que em cada momento é conhecida a localização SL (em variável local) do static link corrente, assim como o seu tipo JVM (Lcurrframetype)

$E[[\text{decl } x_1=e_1 \dots x_n=e_n \text{ in } E \text{ end }]]D =$

new frame_id

dup

invokespecial frame_id/<init>()V

dup

aload SL

putfield frame_id/sl Lcurrframetype

astore SL

Compilação da Linguagem Core

$E[[\text{decl } x_1=e_1 \dots x_n=e_n \text{ in } E \text{ end }]]D =$ (continued)

new frame_id

....

dup

invokespecial frame_id/<init>()V

aload SL

dup

$[[E_n]]D+\{x_1 \rightarrow t_1, x_n \rightarrow t_n\}$

aload SL

putfield frame_id/xn Lt_n

putfield frame_id/sl Lcurrframetype

astore SL

$[[E]]D+\{x_1 \rightarrow t_1, x_n \rightarrow t_n\}$

aload SL

aload SL

$[[E_1]]D+\{x_1 \rightarrow t_1, x_n \rightarrow t_n\}$

getfield frame_id/sl Lcurrframetype

putfield frame_id/x1 Lt₁

astore SL

aload SL

$[[E_2]]D+\{x_1 \rightarrow t_1, x_n \rightarrow t_n\}$

putfield frame_id/x2 Lt₂

Compilação da Linguagem Core

$E[[\text{decl } x_1=e_1 \dots x_n=e_n \text{ in } E \text{ end }]]D =$ (continued)

new frame_id

....

dup

invokespecial frame_id/<init>()V

aload SL

dup

$[[E_n]]D+\{x_1 \rightarrow t_1, x_n \rightarrow t_n\}$

aload SL

putfield frame_id/xn Lt_n

putfield frame_id/sl Lcurrframetype

astore SL

$[[E]]D+\{x_1 \rightarrow t_1, x_n \rightarrow t_n\}$

aload SL

aload SL

$[[E_1]]D+\{x_1 \rightarrow t_1, x_n \rightarrow t_n\}$

getfield frame_id/sl Lcurrframetype

putfield frame_id/x1 Lt₁

astore SL

aload SL

$[[E_2]]D+\{x_1 \rightarrow t_1, x_n \rightarrow t_n\}$

putfield frame_id/x2 Lt₂

Compilação da Linguagem Core

Compilação de declarações e identificadores

O valor de um identificador tem que ser "procurado" na frame respetiva, tal como indicado pelo ambiente D

$E[[x]]D =$

aload SL

getfield frame_id/sl Lancestor_frame_id

...

getfield frame_id/sl Lancestor_frame_id

getfield frame_id/x Lxtype

O número de desreferenciações de **getfield sl** é dado por $D(x, \text{level})$

É necessário definir também os tipos das frames intermédias (guardadas em D)

Note-se também que $Lxtype = D(x, \text{type})$

Compilação da Linguagem Core

Compilação de referências (células de memória), afectação e desreferenciação

Estrutura genérica de uma referência (usando syntax Jasmin)

```
.class          ref_class
.super          java/lang/Object
.field          public v Lvalue_type;
.end method
```

value_type será o tipo do valor guardado no objecto referência

Na prática, e para já, precisamos apenas de três tipos de conteúdo

I (int)

Lref_class (ref T)

Compilação da Linguagem Core

Compilação de referências (células de memória), afectação e desreferenciação

Estrutura genérica de uma referência (usando syntax Jasmin)

```
.class          ref_int
.super          java/lang/Object
.field          public v I;
.end method
```

```
.class          ref_class
.super          java/lang/Object
.field          public v Ljava/lang/Object;
.end method
```


Compilação da Linguagem Core

Compilação de referências (células de memória), afectação e desreferenciação

Caso em que tipo de E = int (ou bool)

E[[new E]]D =

new ref_int

dup

invokespecial ref_int/<init>()V

dup

[[E]]D

putfield ref_int/v I

Compilação da Linguagem Core

Compilação de referências (células de memória), afectação e desreferenciação

Caso em que tipo de E = int (ou bool)

$E[[! E]]D =$

$[[E]]D$

checkcast ref_int

getfield ref_int/v I

Caso em que tipo de $E2$ = int (ou bool)

$E[[E1 := E2]]D =$

$[[E1]]D$

checkcast ref_int

$[[E2]]D$

putfield ref_int/v I

Compilação da Linguagem Core

Compilação de referências (células de memória), afectação e desreferenciação

Caso em que tipo de E = ref T

E[[new E]]D =

new ref_class

dup

invokespecial ref_class/<init>()V

dup

[[E]]D

putfield ref_class/v Ljava/lang/Object

Compilação da Linguagem Core

Compilação de referências (células de memória), afectação e desreferenciação

Caso em que tipo de E = ref T

$E[[! E]]\mathcal{D} =$

$[[E]]\mathcal{D}$

checkcast ref_class

getfield ref_class/v Ljava/lang/Object

Caso em que tipo de E2 = ref T

$E[[E1 := E2]]\mathcal{D} =$

$[[E1]]\mathcal{D}$

checkcast ref_class

$[[E2]]\mathcal{D}$

putfield ref_class/v Ljava/lang/Object

Compilação da Linguagem Core

Compilação de outros comandos, é bastante fácil

[[**while** E1 **do** E2 **end**]]D =

L1:

[[E1]]D

ifeq L2

[[E2]]D

goto L1

L2:

Com o método de avaliação em "curto circuito", pode simplificar-se um pouco a compilação de expressões condicionais e em particular no contexto de construções condicionais como while, if then else, etc...

Compilação da Linguagem Core

Avaliação em curto circuito (CC) de uma expressão lógica E

Ideia geral

o código gerado para E termina com um salto para TL se o valor de E é **true** e com um salto para FL se o valor de E é **false**.

A geração de código para E depende de duas labels TL e TF, que são dadas como parâmetros do algoritmo de compilação CC da expressão E

$[[E, TL, FL]]D$

$[[true, TL, FL]]D =$

goto TL

$[[false, TL, FL]]D =$

goto FL

Compilação da Linguagem Core

$[[\sim E2, TL, FL]]D =$

$[[E1, FL, TL]]D$

$[[E1 \&\& E2, TL, FL]]D =$

$[[E1, NEWLabel, FL]]D$

NEWLabel:

$[[E2, TL, FL]]D$

$[[E1 || E2, TL, FL]]D =$

$[[E1, TL, NEWLABEL]]D$

NEWLabel:

$[[E2, TL, FL]]D$

Compilação da Linguagem Core

$[[E1 > E2, TL, FL]]D =$

$[[E1]]D$

$[[E2]]D$

isub

ifgt TL

goto FL

Compilação da Linguagem Core

[[while E1 do E2 end]]D =

L0:

[[E1, L1, L2]]D

L1:

[[E2]]

goto L0

L2:

Compilação da Linguagem Core

[[if E1 then E2 else E3 end]]D =

[[E1, L1, L2]]D

L1: [[E2]]

goto LE

L2: [[E3]]

LE:

Podem ser feitas melhorias óbvias a este esquema de compilação curto circuito de forma a evitar geração de goto's inúteis, do tipo:

goto L

L: