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

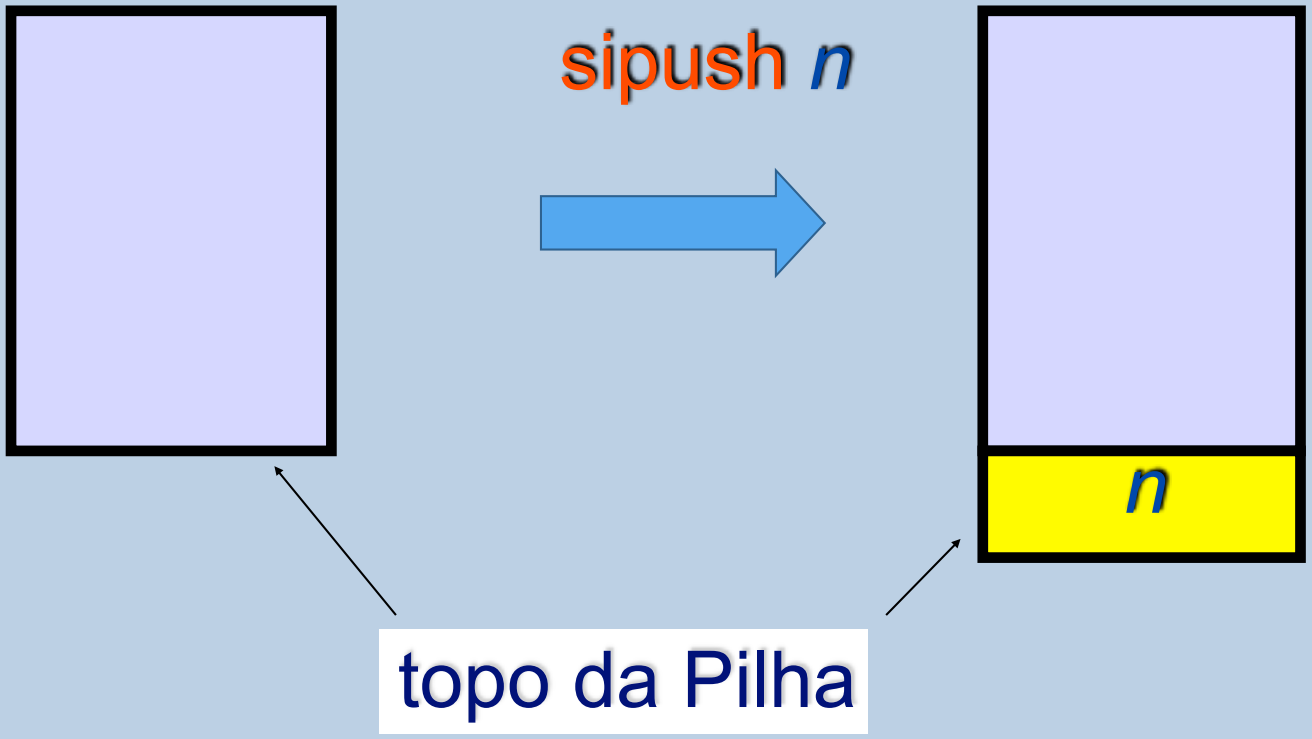
Java Virtual Machine (recap)

- 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

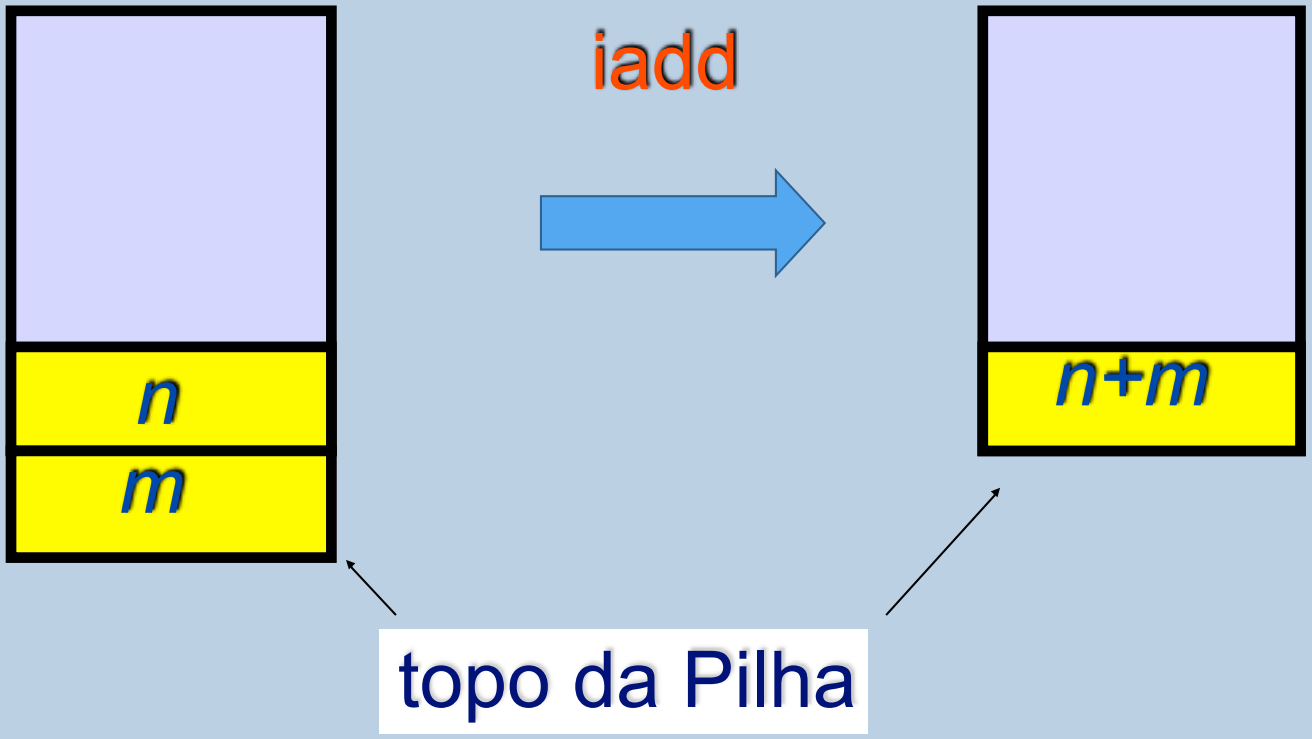
Java Virtual Machine

- "Primeiras" (5) instruções: sipush *n*, iadd, mul, idiv, isub.
- Load Constant (sipush *n*)



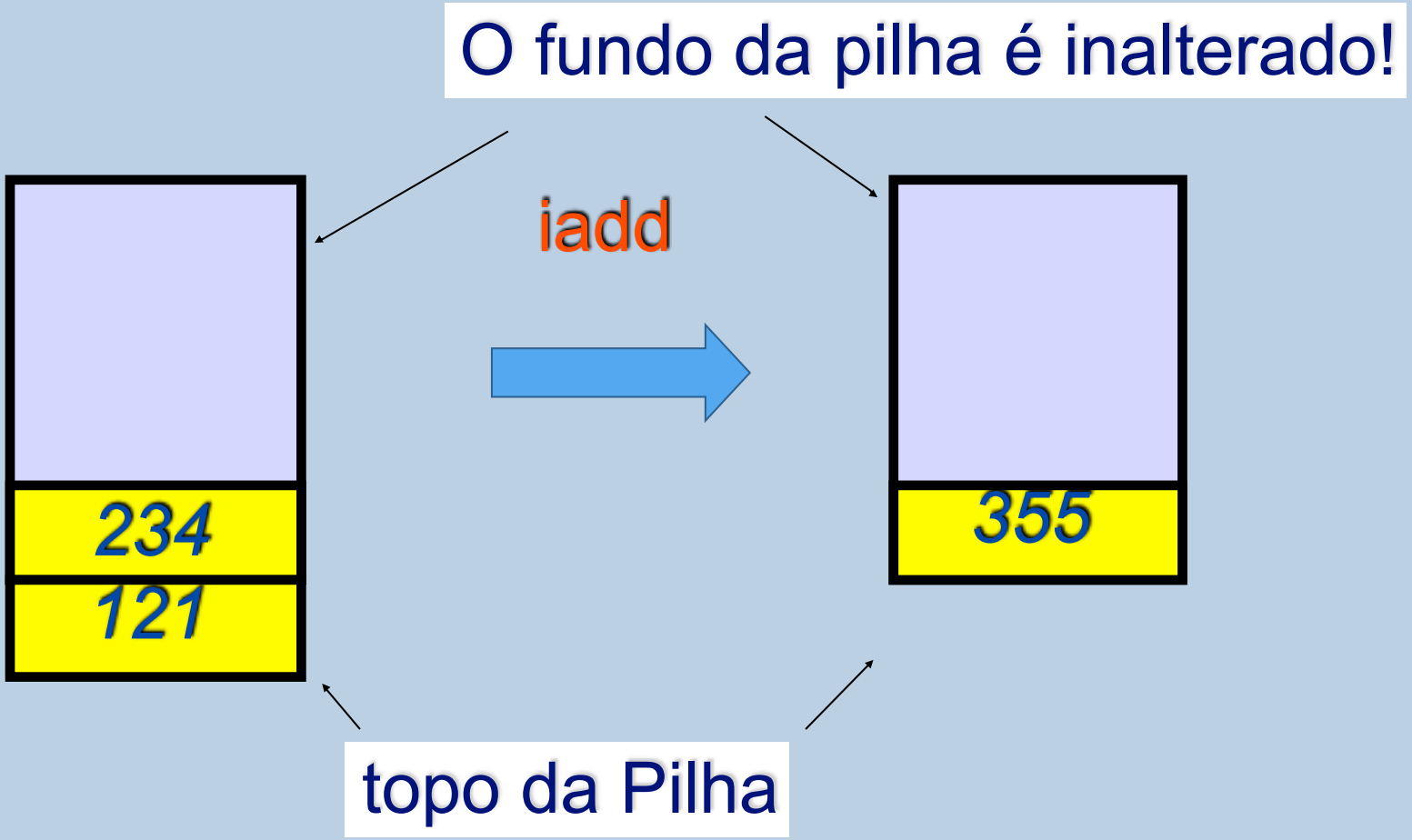
Java Virtual Machine

- "Primeiras" (5) instruções: sipush *n*, iadd, mul, idiv, isub.
- Add (iadd)



Java Virtual Machine

- "Primeiras" (5) instruções: sipush *n*, iadd, mul, idiv, isub.
- Add (iadd)



Compilador de CALC

- **Algoritmo comp(E) para traduzir uma expressão E qualquer de CALC numa sequência de instruções JVM**

$$comp : CALC \rightarrow CodeSeq$$

se E é da forma **num**(n):	comp(E) ≜ < **`sipush n`** >
se E é da forma **add**(E',E''):	s1 = comp(E'); s2 = comp(E'');
	comp(E) ≜ s1 @ s2 @ < i**`add`** >
se E é da forma **mul**(E',E''):	v1 = comp(E'); v2 = comp(E'');
	comp(E) ≜ s1 @ s2 @ < i**`mul`** >
se E é da forma **sub**(E',E''):	v1 = comp(E'); v2 = comp(E'');
	comp(E) ≜ s1 @ s2 @ < i**`sub`** >
se E é da forma **div**(E',E''):	v1 = comp(E'); v2 = comp(E'');
	comp(E) ≜ s1 @ s2 @ < i**`div`** >

Compilador de CALC

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALC numa sequência de instruções JVM

$$comp : CALC \rightarrow CodeSeq$$

comp(**num**(n)) ≜ < **sipush n** >

comp(**add**(E',E'')) ≜ comp(E') @ comp(E'') @ < i**add** >

comp(**mul**(E',E'')) ≜ comp(E') @ comp(E'') @ < i**mul** >

comp(**sub**(E',E'')) ≜ comp(E') @ comp(E'') @ < i**sub** >

comp(**div**(E',E'')) ≜ comp(E') @ comp(E'') @ < i**div** >

Correcção do Compilador

- Algoritmo comp(E) para traduzir uma expressão E qualquer da linguagem CALC numa sequência de instruções da linguagem JVM

$$comp : CALC \rightarrow CodeSeq$$

- **Propriedade de Correcção:** Quando a sequência de instruções 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 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

Compilação da Linguagem Core

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

D(x,type) = o tipo do identificador x (determinado pelo typechecker)

D(x,level) = o número de niveis de ambiente a percorrer até se atingir x

Se x estiver declarado no nível corrente (mais interno), D(x,level)=0.

Se x estiver declarado no nível a seguir, D(x,level)=1, etc.

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

Compilação da Linguagem Core

Compilação de expressões aritméticas (exemplos)

[[E1 + E2]]D =

    [[E1]]D

    [[E2]]D

    iadd

[[E1 * E2]]D =

    [[E1]]D

    [[E2]]D

    imul

Compilação da Linguagem Core

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

[[E1 > E2]]D =

```
    [[E1]]D
    [[E2]]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

    iand

[[E1 || E2]]D =

    [[E1]]D

    [[E2]]D

    ior

Compilação da Linguagem Core

Compilação de expressão condicional

[[E1 ? E2 : E3]]D =

[[E1]]D

ifeq L1

[[E2]]D

goto L2

L1: [[E3]]D

L2:

Compilação da Linguagem Core

Compilação de declarações e identificadores

Cada bloco de declarações

decl

x1=e1 ...

xn=en

in E **end**

vai (quando executado em runtime) necessitar de criar uma stack frame (no heap) para guardar o valor dos n identificadores x1 ... xn

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[[**decl** x1=e1 ... xn=en **in** E **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[[**decl** x1=e1 ... xn=en **in** E **end**]]D =

```
new frame_id
dup
invokespecial frame_id/<init>()V
dup
aload SL
putfield frame_id/sl Lcurrframetype
astore SL

aload SL
[[E1]]D+{x1|->t1, xn|->tn}
putfield frame_id/x1 Lt1

aload SL
[[E2]]D+{x1|->t1, xn|->tn}
putfield frame_id/x2 Lt2
```

(continued)

```
....

aload SL
[[En]]D+{x1|->t1, xn|->tn}
putfield frame_id/xn Ltn

[[E]]D+{x1|->t1, xn|->tn}

aload SL
getfield frame_id/sl Lcurrframetype
astore SL
```

Compilação da Linguagem Core

E[[**decl** x1=e1 ... xn=en **in** E **end**]]D =

```
new frame_id
dup
invokespecial frame_id/<init>()V
dup
aload SL
putfield frame_id/sl Lcurrframetype
astore SL

aload SL
[[E1]]D+{x1|->t1, xn|->tn}
putfield frame_id/x1 Lt1

aload SL
[[E2]]D+{x1|->t1, xn|->tn}
putfield frame_id/x2 Lt2
```

(continued)

```
....

aload SL
[[En]]D+{x1|->t1, xn|->tn}
putfield frame_id/xn Ltn

[[E]]D+{x1|->t1, xn|->tn}

aload SL
getfield frame_id/sl Lcurrframetype
astore SL
```

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,level)

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

Note-se também que Lxtype = D(x,type)

Compilação da Linguagem Core

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

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 desreferenciaçãpo

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 desreferenciaçãpo

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 desreferenciaçãpo

Caso em que tipo de E = ref T

E[[! E]]D =

[[E]]D
checkcast ref_class
getfield ref_class/v Ljava/lang/Object

Caso em que tipo de E2 = ref T

E[[E1 := E2]]D =

[[E1]]D
checkcast ref_class
[[E2]]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

[[~ 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:

Compilação da Linguagem Core

[[**(fun x1, …, xk => e)**]]D =

```
new closure_f0001

dup

aload SL

putfield closure_f0001/SL type ; // set environment field of the closure
```

Ideia geral

Para cada função deve ser gerada uma classe específica, que representa a closure associada. **Ver página seguinte, onde o esquema geral é apresentado.**

Cada closure é um objeto com um field onde é guardado o ambiente (aquele onde a função foi definida), e com um método call(..) que permite executar o corpo da função.

Compilação da Linguagem Core

```
.class              closure_f0001 // code schema for (fun x1,…,xk => e)
.implements         closure_interface_type_t002 ; // depends on the type of the function
.field              public SL  Lancestor_frame_id; .locals k+1
method             call(type1;type2;…)type
new f0001_frame ; // the function stack frame
dup
aload 0 ; get this
getfield closure_f0001/SL type // get the env stored in this closure
putfield f0001_frame/sl type // link to activation record
dup
aload 1 ; // first argument
putfield f0001_frame/loc_01 type
dup
….
aload k ; / kth argument
putfield f0001_frame/loc_0k type // all argument values are stored in frame
astore SL ; // save new SL. SL must be computed for every function scope (see note below)
[[ e ]]
return
.end
```

Compilação da Linguagem Core

[[**e (e1, e2, … ek)**]]D =

[[e]]

checkcast closure_interface_type_t002

[[e1]]

….

[[ek]]

invokeinterface closure_interface_type_t002/call(type1;type2;…)type

Ideia geral

Para cada que qualquer função (closure) possa ser chamada de forma uniforme, com base apenas no seu tipo (que é apenas o que é conhecido em tempo de compilação), define-se-para cada tipo de função (IIII)I uma interface própria.

Compilação da Linguagem Core

Para cada que qualquer função (closure) possa ser chamada de forma uniforme, com base apenas no seu tipo (que é apenas o que é conhecido em tempo de compilação), define-se-para cada tipo de função (IIII)I uma interface própria.

Por exemplo, para o tipo int x int -> int, seria a interface

```
.interface      closure_interface_type_INTxINT2INT
.method         call(II)I
.end
```