Interpretação e Compilação (de Linguagens de Programação)

Unidade 3: Semântica Operacional

Os interpretadores e compiladores são algoritmos que aceitam programas sintaticamente válidos e produzem denotações (valores, efeitos, ou outros programas). Os interpretadores têm como resultado um valor ou efeito, os compiladores têm como resultado um programa numa linguagem de uma máquina.

Como podemos definir o processamento de programas?

- Definição composicional de Semântica operacional.
- Algoritmo interpretador de CALC: implementação usando uma linguagem orientada por objectos (Java)
- Algoritmo compilador de CALC: introdução à máquina virtual CLR

Semântica de CALC

- A função semântica I pode ser definida por um algoritmo que "sabe como interpretar" todos os programas sintacticamente correctos de uma linguagem, determinando o seu valor ou efeito

$$I : CALC \rightarrow Integer$$

CALC = conjunto dos programas válidos

Integer = conjunto dos significados (denotações)

Interpretadores

- Um interpretador para uma linguagem de programação é um algoritmo que atribui um valor ou efeito a cada programa legítimo da linguagem
- Os programas fornecidos como dados de entrada a um interpretador são representados por valores da sintaxe abstracta da linguagem a que pertencem
- Um algoritmo interpretador pode ser definido indutivamente na sintaxe abstracta da linguagem

Interpretadores

- Um interpretador para uma linguagem de programação é um algoritmo que atribui um valor ou efeito a cada programa legítimo da linguagem
- Os programas fornecidos como dados de entrada a um interpretador são representados por valores da sintaxe abstracta da linguagem a que pertencem
- Um algoritmo interpretador pode ser definido indutivamente na sintaxe abstracta da linguagem

Resumindo:

- Programas exprimem algoritmos que processam dados
- Programas também podem ser vistos como dados
- Um interpretador é um programa que processa programas

A Linguagem CALC (como tipo indutivo)

- Tipo de dados CALC com os construtores: num, add, mul, div, sub

num: Integer → CALC

add: CALC × CALC → CALC

mul: CALC × CALC → CALC

div: CALC × CALC → CALC

sub: CALC × CALC → CALC

A Linguagem CALC (como tipo indutivo)

- Tipo de dados CALC com os construtores: num, add, mul, div, sub

num: Integer → CALC

add: CALC × CALC → CALC

mul: CALC × CALC → CALC

div: CALC × CALC → CALC

sub: CALC × CALC → CALC

```
type calc =  Num of int
           | Add of calc * calc
           | Mul of calc * calc
           | Div of calc * calc
           | Sub of calc * calc
```

Interpretador de CALC

- **Algoritmo eval(E) para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:**

$$eval : CALC \rightarrow Integer$$

se E é da forma **num**(n):	eval(E) = n
se E é da forma **add**(E',E''):	v1 = eval(E'); v2 = eval(E'');
	eval(E) ≜ v1+v2
se E é da forma **mul**(E',E''):	v1 = eval(E'); v2 = eval(E'');
	eval(E) ≜ v1*v2
se E é da forma **sub**(E',E''):	v1 = eval(E'); v2 = eval(E'');
	eval(E) ≜ v1-v2
se E é da forma **div**(E',E''):	v1 = eval(E'); v2 = eval(E'');
	eval(E) ≜ v1/v2

Interpretador de CALC

- Algoritmo eval(E) para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

$$eval : CALC \rightarrow Integer$$

eval(num(n))	≜ n
eval(add(E',E''))	≜ eval(E') + eval(E'')
eval(mul(E',E''))	≜ eval(E') * eval(E'')
eval(sub(E',E''))	≜ eval(E') - eval(E'')
eval(div(E',E''))	≜ eval(E') / eval(E'')

[notação mais legível, usando pattern matching]

Interpretador de CALC

- Algoritmo eval(E) para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

$$eval : CALC \rightarrow Integer$$

eval(num(n)) ≜ n

```
let rec eval e = match e with
      Num(n) -> n
    | Add(e1,e2) -> (eval e1)+(eval e2)
    | Mul(e1,e2) -> (eval e1)*(eval e2)
    | Sub(e1,e2) -> (eval e1)-(eval e2)
    | Div(e1,e2) -> (eval e1)/(eval e2)
```

Interpretador de CALC

- Algoritmo eval(E) para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

$$eval : CALC \rightarrow Integer$$

- Note bem: a função de interpretação eval(-) é definida recursivamente na estrutura do seu argumento!
- Semântica composicional: o significado do todo só depende do significado das suas partes
- O mesmo não acontece nas linguagens "naturais":
  - time flies like an arrow
  - fruit flies like a banana

Semântica Operacional Estrutural

- Sintaxe (abstracta) definida por um tipo indutivo, apresentada por um conjunto de construtores;
- Semântica definida por um algoritmo interpretador, que atribui um significado (valor) a cada expressão da linguagem, calculando-o composicionalmente a partir do significado das suas subexpressões;
- A esta técnica de definição da semântica de uma linguagem de programação chama-se:

Semântica Operacional Estrutural

Implementação em Java

- Usando uma linguagem baseada em objectos, um tipo de dados indutivo pode ser representado por uma interface (que representa o tipo indutivo), e por um conjunto de classes (em que cada classe representa um construtor do tipo indutivo)
- A interface pode declarar uma ou mais operações sobre o tipo indutivo, por exemplo:

```
public interface CALC {

     int eval();

}
```

Ou seja, eval: CALC → Integer

Implementação em Java

- Cada classe representa um (e um só) dos construtores do tipo indutivo
- Cada classe fornece a implementação relativa ao construtor que representa, para cada operação definida sobre o tipo indutivo

```
public class Num implements CALC {
     private int value ;
     Num(int v) { value = v; }
     int eval() { return value; }
}
```

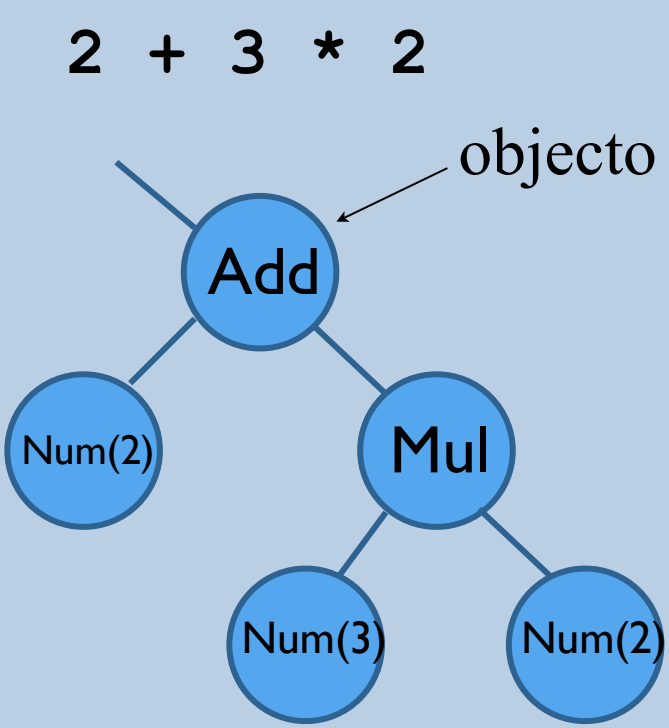
Implementação em Java

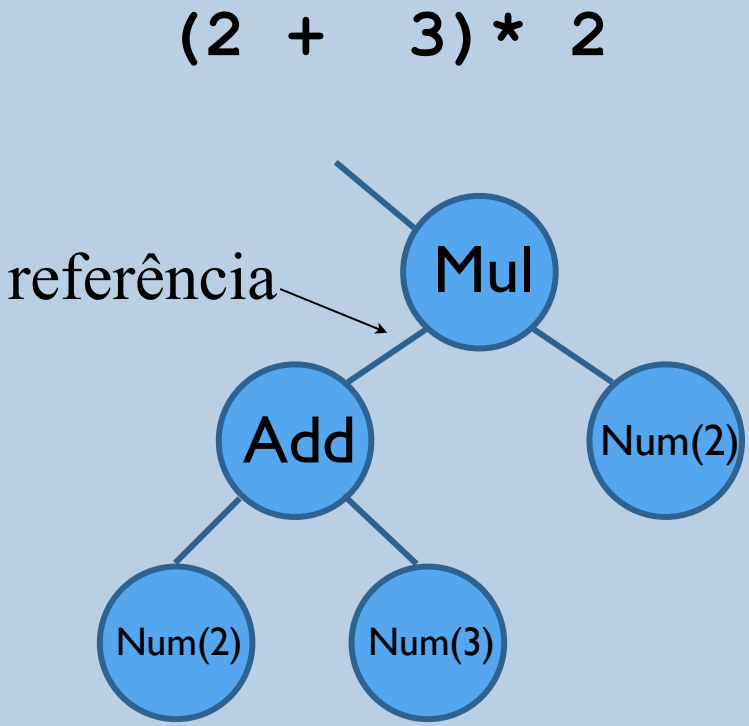
- Cada expressão é representada por uma árvore (n-ária) de objectos (uma AST);
- Cada construtor do tipo indutivo é aplicado chamando o construtor da classe respectiva:

```
CALC expr1 = new Add(new Num(2),new Num(3));
int result1 = expr1.eval();
CALC expr2 =
     new Add(new Sub(new Num(2),new Num(3)),new Num(3));
int result2 = expr2.eval();
```

Árvore Sintáctica Abstracta

- Representa a estrutura de uma frase da linguagem em termos dos seus constructores abstractos.
- Abstract Syntax Tree (AST)





Implementação em Java

- No nosso exemplo, temos as classes `Num`, `Add`, `Sub`, `Mul` e `Div`, implementando os construtores da linguagem num, add, sub, mul e div.
- A definição das operações fica "dispersa" pelas várias classes (é mais cómodo acrescentar novos construtores a uma linguagem do que novas operações)

```
public class Add implements CALC {
     private CALC lhs;
     private CALC rhs;
     Add(CALC l, CALC r) { lhs = l; rhs = r; }
     int eval() { return lhs.eval()+rhs.eval(); }
}
```

Compiladores

- Um compilador para uma linguagem de programação é um algoritmo que traduz cada programa legítimo da linguagem numa versão do mesmo program directamente executável por uma "máquina".
- A máquina pode ser física (Pentium) ou virtual (JVM, CLR)
- O destino do programa gerado pode ser outro compilador já existente [exemplo: tradutor de Java em C para ser processado pelo gcc]
- A tradução feita por um compilador preserva a semântica do programa.
- Um algoritmo compilador pode ser definido indutivamente na sintaxe abstracta da linguagem

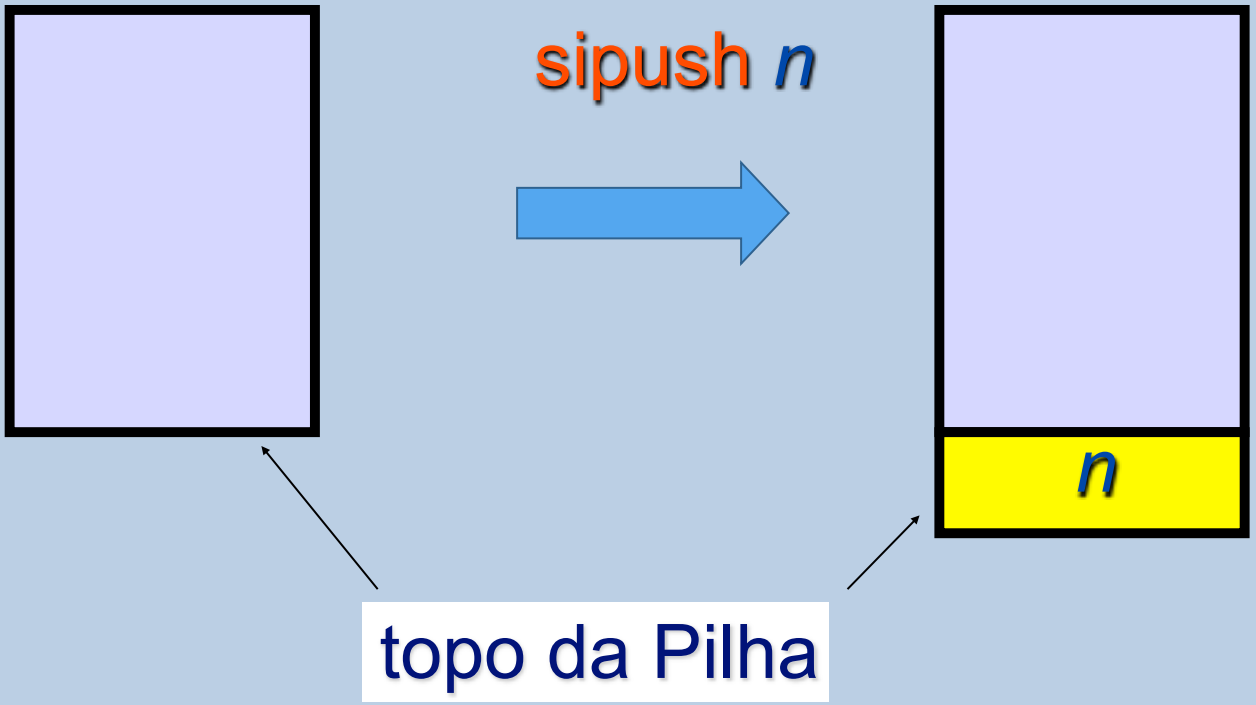
Java Virtual Machine

- É uma máquina de pilha: todas as instruções consomem argumentos do topo da Pilha, e deixam um resultado no topo da Pilha (*stack machine*)

- "Primeiras" (5) instruções da JVM:
  - sipush *n* : Carrega o valor n (short integer) no topo da pilha
  - iadd : Retira dois valores inteiros do topo da pilha e coloca na pilha a sua soma
  - imul : idem para a multiplicação
  - idiv : idem para a divisão
  - isub : idem para a subtracção

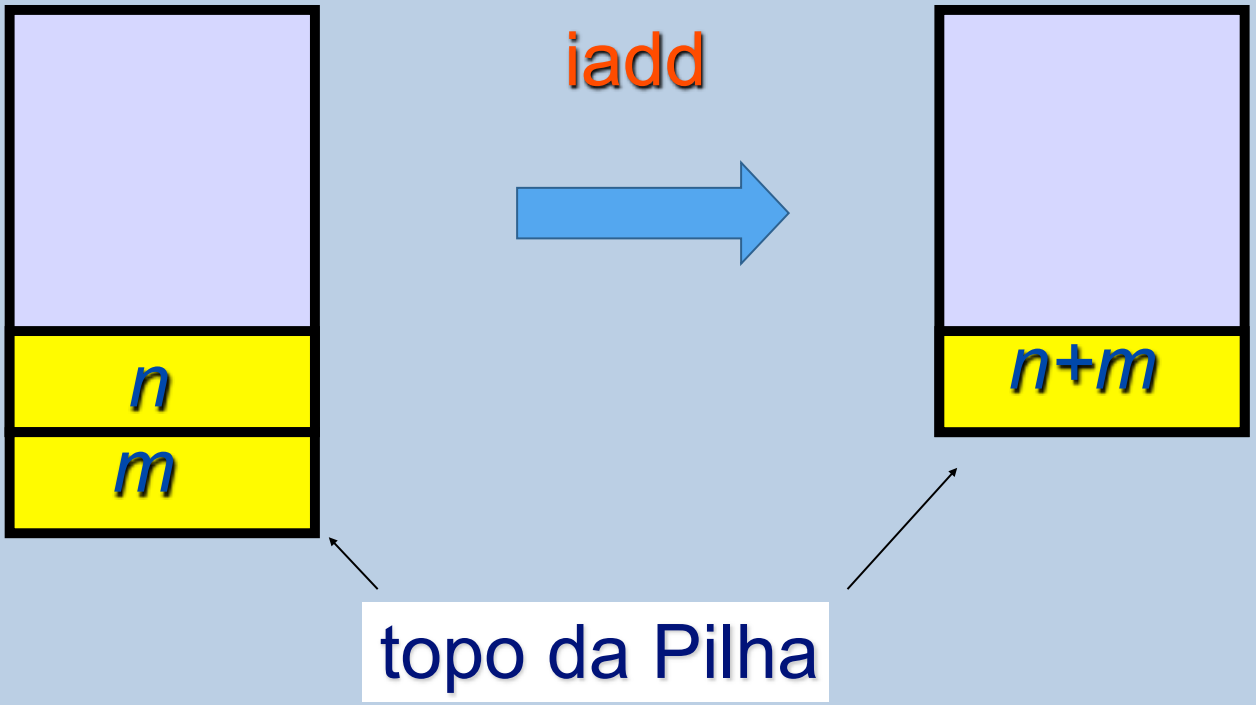
Java Virtual Machine

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



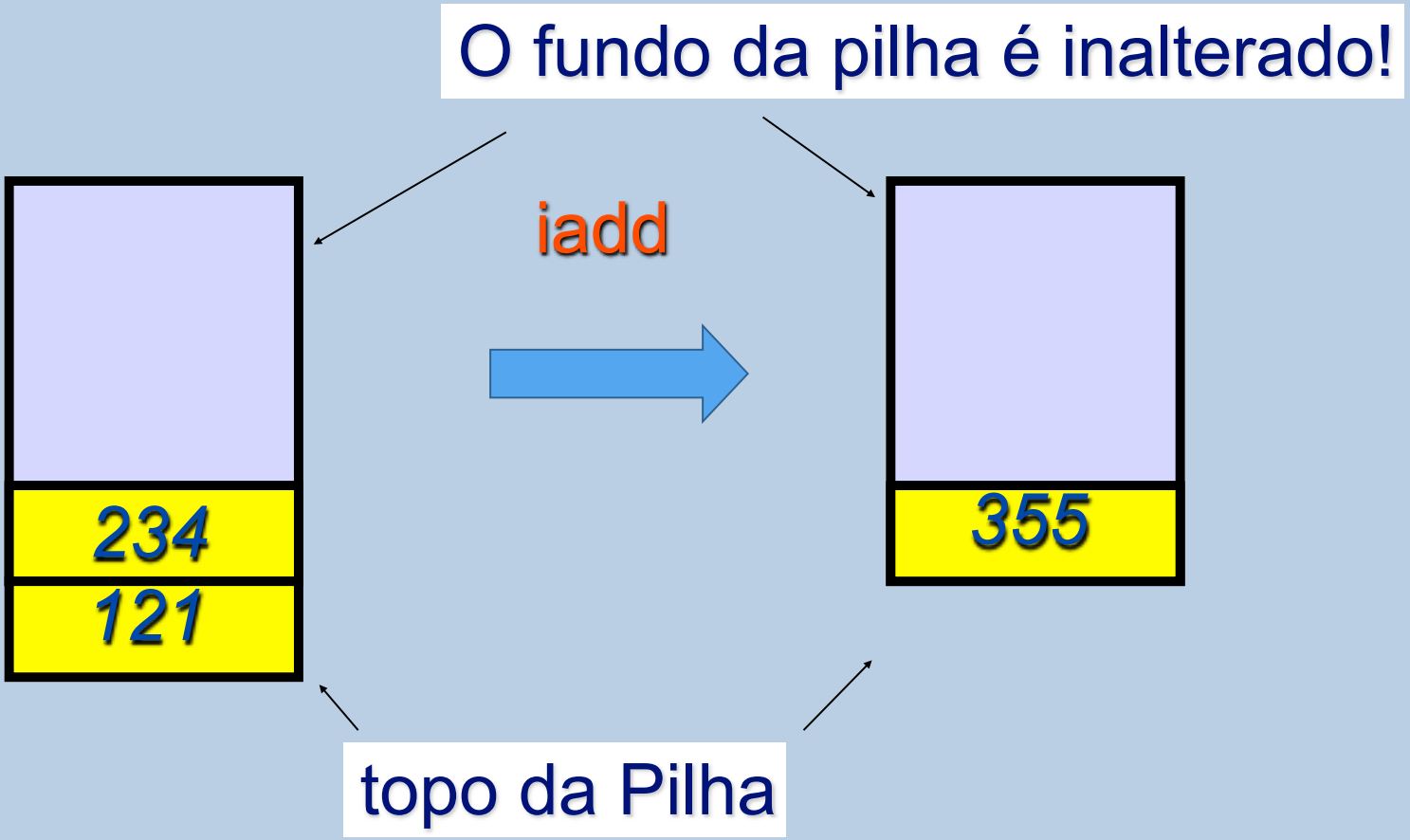
Java Virtual Machine

- "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 a pilha 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
```