**Interpretação e Compilação de Linguagens**

## Interpretador

Executa programa fonte diretamente

## Compilador

Produz um programa em linguagem alvo de baixo nivel. Este programa alvo implementa depois o programa fonte

**Exemplo - CALC language**

* Interpretador para CALC: implementação usando Java - a função de avaliação é definida “em pedaços”, um caso para cada construtor do AST
* Compilador para CALC: implementação usando Java - a função de compilação é definida “em pedaços”, um caso para cada construtor do AST, e temos como maquina alvo a JVM (máquina virtual Java). A denotação do programa fonte e do alvo é a mesma.

# Syntax

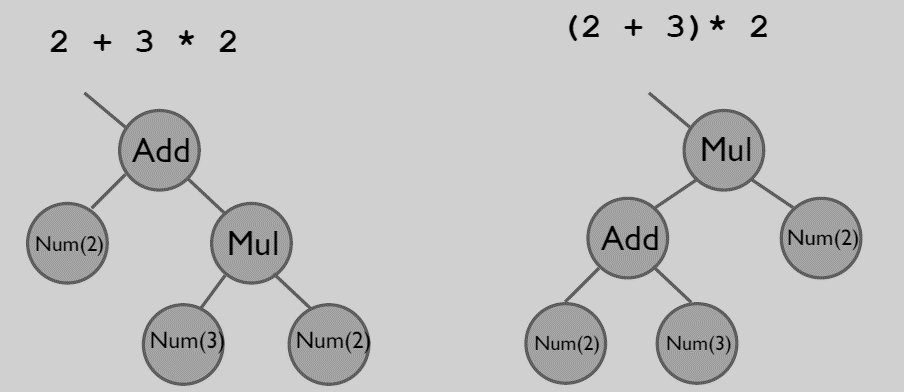
Concreta – 2+1

* define a forma como as expressões e os programas são efetivamente escritos em termos de formatação, sequências de caracteres (ascii/unicode), etc…

Abstrata – add(num(2), num(1))

* define a estrutura profunda de expressões e programas em termos de uma composição de construtores abstratos

# Abstract syntax tree



# CALC interpreter

Alg. **eval(***E***)** computa a notação de de uma qualquer *CALC expression*

O interpretador vai interpretar qual a denotação da expressão(add, mult, sub, ...).

*E* pode ser:

* Num(n)
* Add(E’,E’’)
* . . .

# Java implementation

* Cada expressão da linguagem é assim representada por uma árvore (n-ária) de objetos (o AST)
* Construtores do AST são os construtores do tipo de dados indutivo e implementados pelos construtores das classes AST

# Java virtual machine JVM

**Stack Machine:** todas as instruções consomem seus argumentos do topo da pilha e deixam um resultado no topo da pilha

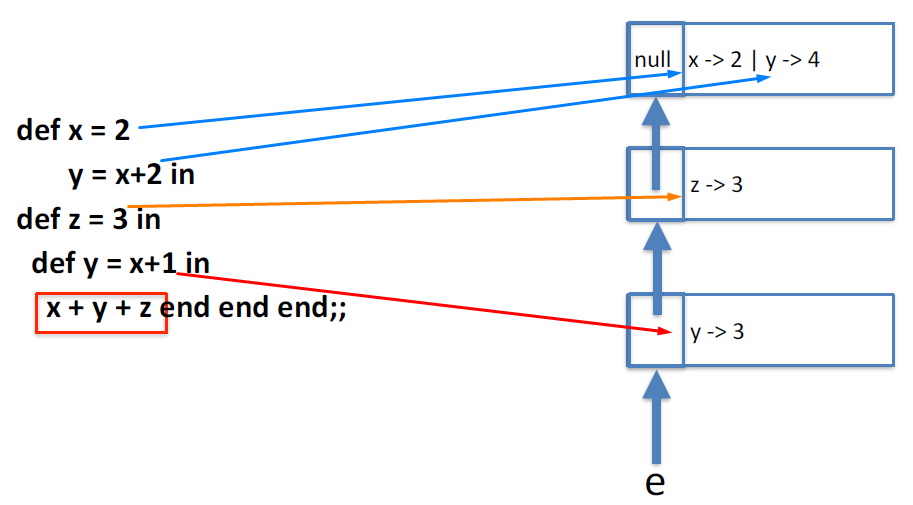
Instruções da JVM:

* sipush n : empurra o inteiro n no topo da pilha (tos)
* iadd : Desempilha dois valores inteiros de tos e empurra sua soma
* imul : da mesma forma para sua multiplicação
* idiv : da mesma forma para sua divisão
* isub : da mesma forma para sua subtração

O ambiente de compilação D mapeia cada nome livre do programa a ser compilado em suas coordenadas:

**D(x) = (d, s)** onde . . .

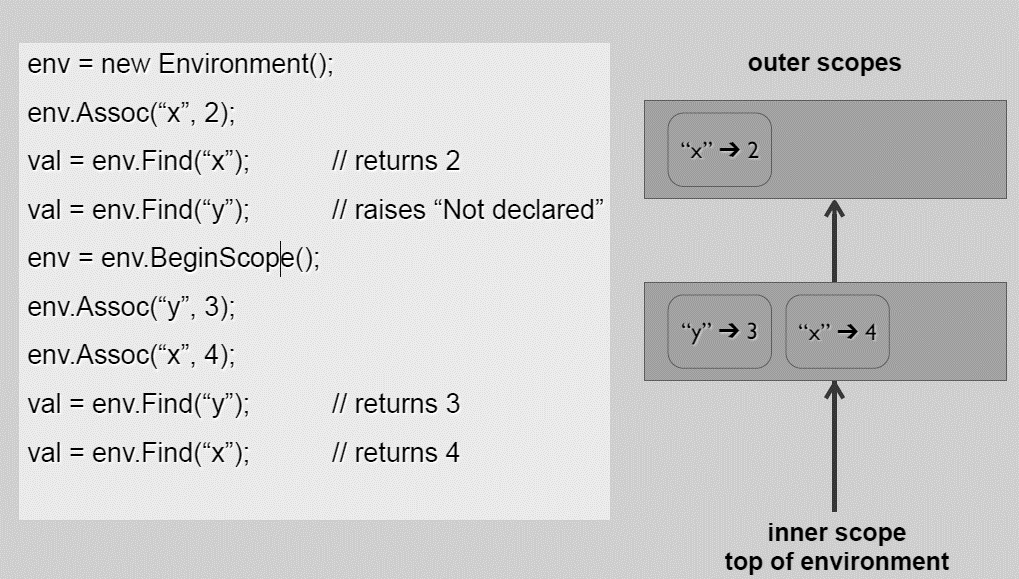
* **d**: a profundidade da stack de frames onde o identificador é declarado
* **s**: o slot na frame onde o valor é armazenado



D(**x**) = (0,0)

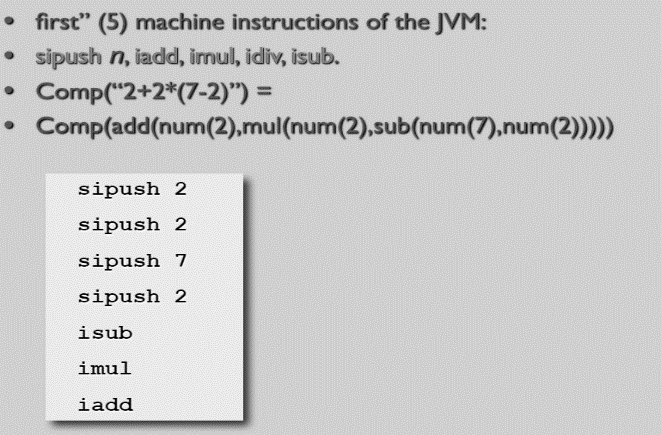
D(**y**) = (0,1)

D(**z**) = (1,0)



# CALC compiler

Algoritmo comp(*E*) que traduz a expressão CALC *E* em uma sequência de instruções JVM





# Naming

Os nomes são a primeira ferramenta que se usa para introduzir a abstração em uma linguagem de programação

Fundamentalmente, o significado de um fragmento de programa com nomes é obtido substituindo cada nome pelo valor atribuído a ele em sua definição.

# Binding and Scope

**Binding** 🡪 associação entre um intentificador e um valor

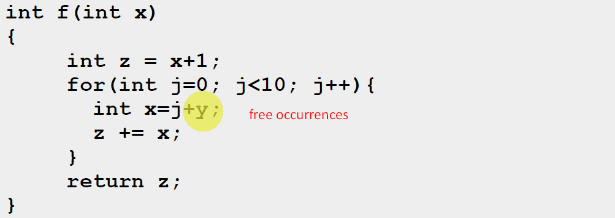
* O binding é defenido em contexto sintatico e criado por uma declaração (x = 1)

**Scope** 🡪 zona onde o binding/declaração e establecida

Para cada ocorrência vinculada há uma e apenas uma ocorrência vinculante (de uma ocorrência na declaração)

# 

Qualquer ocorrência de um identificador que não seja vinculativo nem vinculado é dito livre



# Language CALC

CALCI estende nossa linguagem de expressão básica CALC com declarações gerais def:



Um programa CALCI é uma expressão fechada de CALCI



# The Environment as an ADT

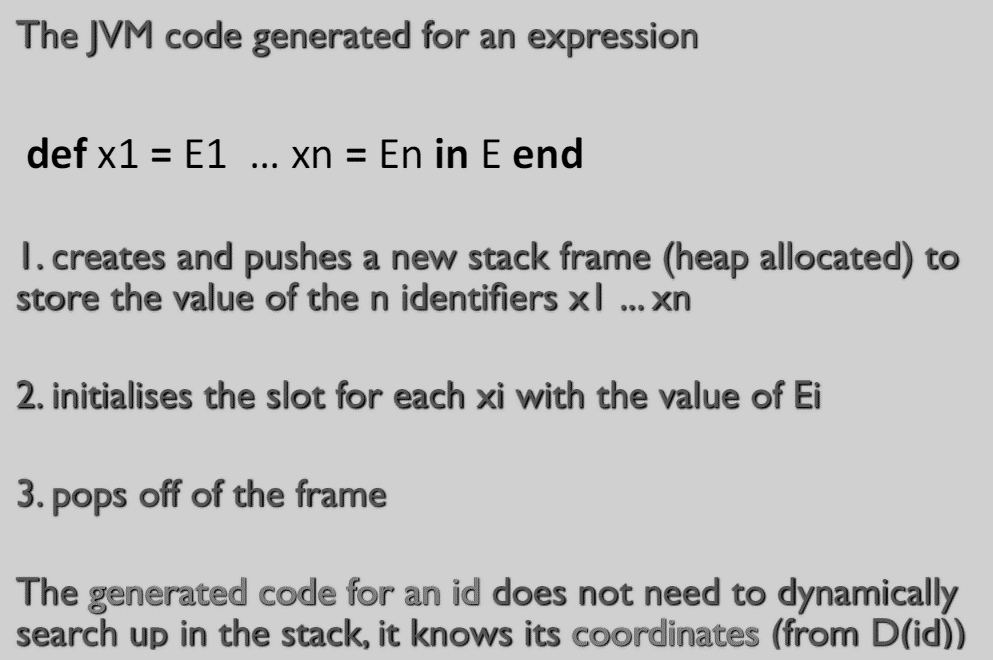
Na prática, é conveniente implementar ambientes usando uma estrutura de dados semelhante a uma pilha mutável chamada “spaghetti stack”.

• Um ambiente armazena todas as ligações relativas ao escopo atual e todos os escopos envolvendo em quadros.

• A partir de qualquer estado do ambiente pode-se criar um novo quadro “filho”, correspondente a um novo escopo aninhado.

• Cada quadro é vinculado ao quadro ancestral usando uma referência.

# Complilation of CALCI



-------------------------------------------- **in x + y + k**

aload\_3

getfield frame\_1/sl Lframe\_0;

getfield frame\_0/v0 I

aload\_3

getfield frame\_1/sl Lframe\_0;

getfield frame\_0/v1 I

iadd

aload\_3

getfield frame\_1/v0 I

iadd

-------------------------------------------- **end end;;**

aload\_3

getfield frame\_1/sl Lframe\_0;

astore\_3

aload\_3

getfield frame\_0/sl Ljava/lang/Object;

astore\_3

-------------------------------------------- **def**

new frame\_0

dup

invokespecial frame\_0/<init>()V

dup

aload\_3

putfield frame\_0/sl Ljava/lang/Object;

astore\_3

-------------------------------------------- **x = 2 y = 3**

aload\_3

sipush 2

putfield frame\_0/v0 I

aload\_3

sipush 3

putfield frame\_0/v1 I

-------------------------------------------- **in def**

new frame\_1

dup

invokespecial frame\_1/<init>()V

dup

aload\_3

putfield frame\_1/sl Lframe\_0;

astore\_3

-------------------------------------------- **k = x + y**

aload\_3

getfield frame\_1/sl Lframe\_0;

getfield frame\_0/v0 I

aload\_3

getfield frame\_1/sl Lframe\_0;

getfield frame\_0/v1 I

iadd

aload\_3

putfield frame\_1/v0 I

# Functions

# 

**Fator -> (abstract syntax)**

| num | true | false | id

| EE + EE | EE - EE

| EE \* EE | EE / EE | -EE | ( EE )

| EE == EE | EE > EE | EE >= EE |…

| EE && EE | EE || EE | ~ EE

| def (id = EE)+ in EE end

| new EE | EE := EE | ! EE

| if EE then EE else EE end

| while EE do EE end

| print EE

| EE ; EE

Alphabet = { num, +, -, \*, /, (, ) }

Grammar (non-ambiguous and LL(1)):

* E -> TE’
* E’ -> ε | + E
* T -> FT’
* T’ -> ε | \* T
* F -> num
* F -> ( E )
* F -> - F

EBNF (Extended BNF)

* E -> T [ ( + | - ) T ] \*
* T -> F [ ( \* | / ) F ] \*
* F -> num | ( E ) | - F | { ( let id = EE ; )+ EE }

# Project code

// ToDo

**-------------------------------------------------------------------------------------------------------------**

**-------------------------------------------------------------------------------------------------------------**

Token stringLiteral in javacc as a regular expression

--> <Id: “”” ([“a”-“z”, “A”-“A”, “0”-“9” ])\* “””>

Abstract syntax tree for expression: –(-(4-4\*2))

--> ASTNeg( ASTNeg( ASTSub( ASTNum(4) , ASTMult( ASTNum(4) , ASTNum(2) ))))