**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**

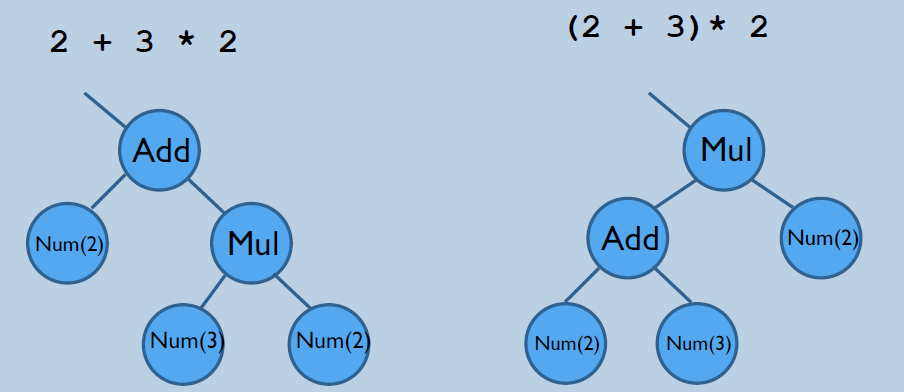
* Interpretador para CALC: implementação usando uma linguagem OO (Java) a função de avaliação é definida “em pedaços”, um caso para cada construtor do AST
* Compilador para CALC: implementação usando uma linguagem OO (Java) a função de compilação é definida “em pedaços”, um caso para cada construtor do AST, temos como alvo a JVM (máquina virtual Java)

# Syntax

Concreta – 2+1

Abstrata – add(num(2), num(1))

# Abstract syntax tree



# CALC interpreter

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

*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 VSM

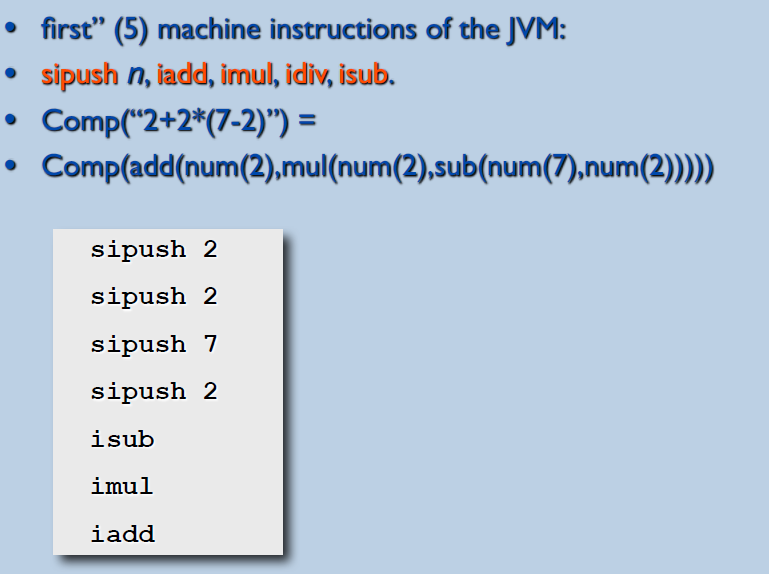
**Stack Machine:** todas as instruções consomem seus argumentos do topo da pilha e deixam um resultado no topo da pilha

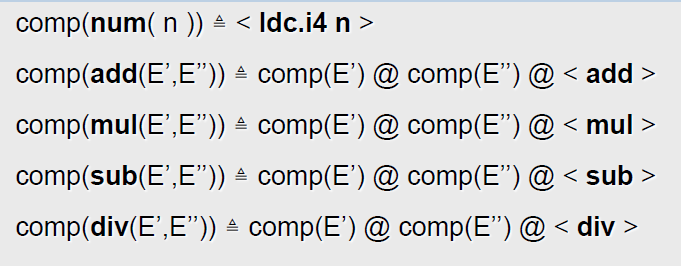
Instruções da JVM:

* 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

# CALC compiler

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





# Naming

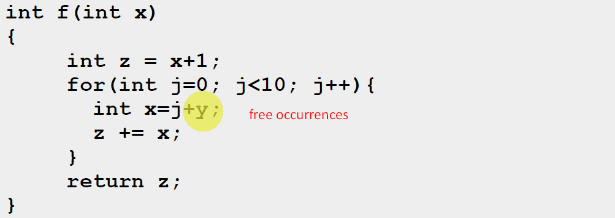
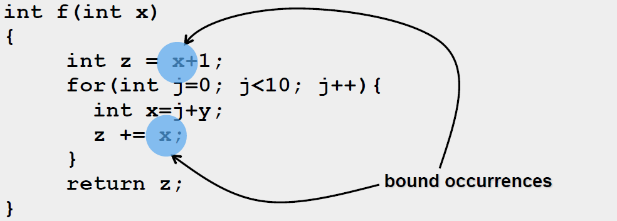
Os nomes são a primeira ferramenta que se usa para introduzir a abstração em uma linguagem de programação

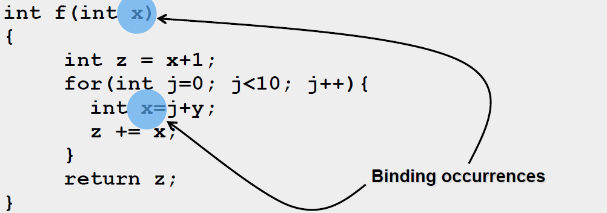
Fundamentalmente, o significado de um fragmento de programa com nomes é obtido substituindo cada nome pelo valor atribuído a ele em sua definição.

# Binding and Scope

A associação entre um identificador e o valor que ele denota é chamada de ligação.

* Uma ligação entre um identificador e o valor associado é sempre estabelecida em um contexto sintático bem definidoe é criada por uma construção do programa chamada declaração
* O contexto sintático qual a vinculação é estabelecida é chamado de escopo da declaração.





# Language CALCI

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

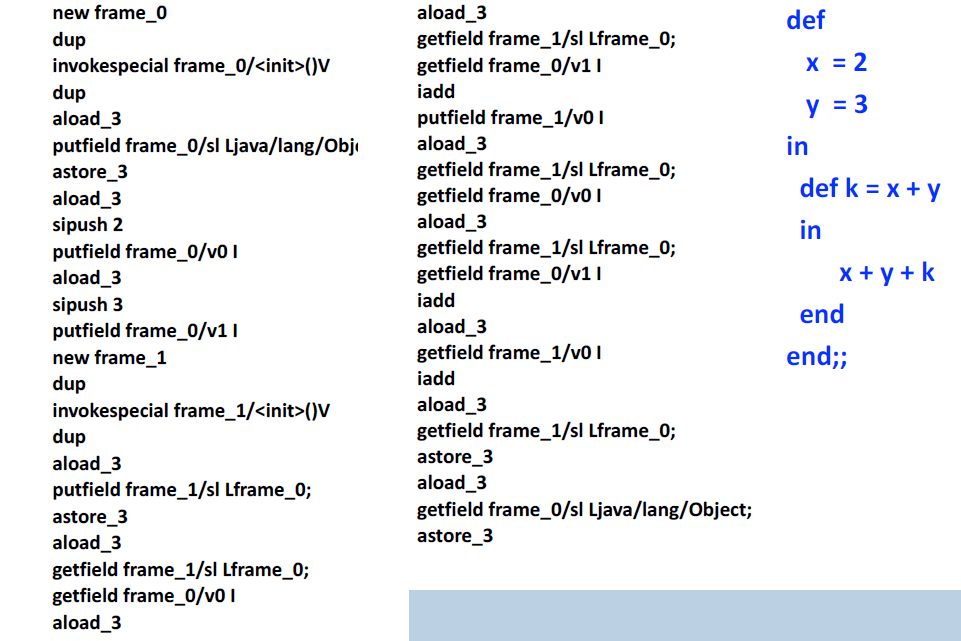
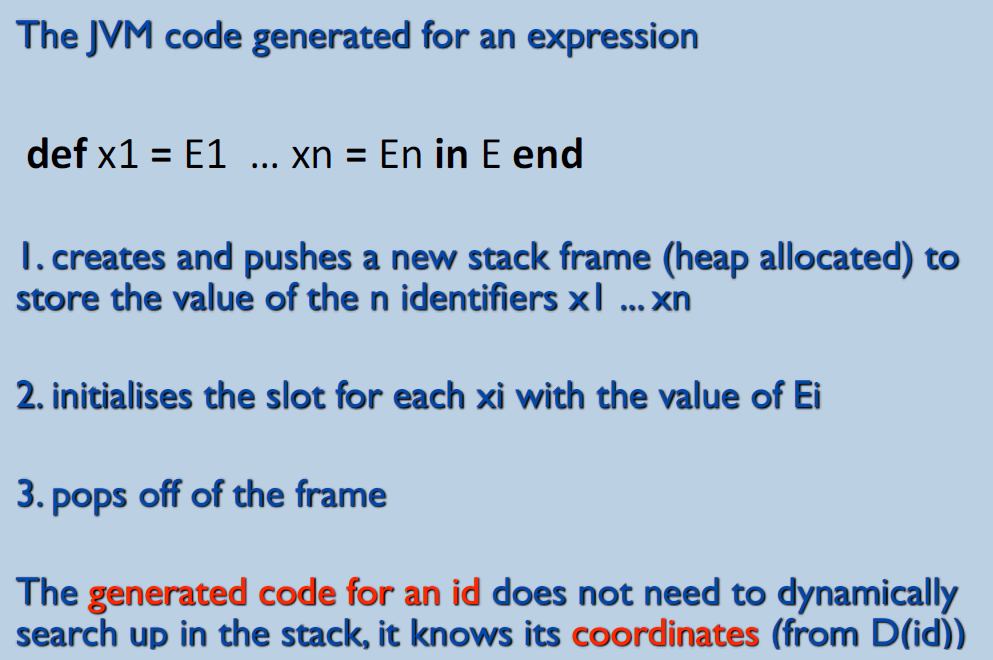
Na prática, é conveniente implementar ambientes usando uma estrutura de dados semelhante a uma pilha mutável chamada “spaghetti stack”.

• 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



# Project code

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

public interface ASTNode {

**PARSER (jasmin - .jj)**

PARSER\_BEGIN(Parser)

/\*\* ID lister. \*/

public class Parser {

}

PARSER\_END(Parser)

SKIP:

{

" "

| "\t"

| "\r"

}

TOKEN:

{

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

|

< Num: (["0"-"9"]) + >

|

< PLUS : "+" >

|

< MINUS : "-">

|

< MULT : "\*">

|

< DIV : "/">

|

< LPAR : "(" >

|

< RPAR : ")" >

|

< EL: "\n" >

}

ASTNode Start():

{ ASTNode t; }

{

t = Exp() <EL>

{ return t; }

}

ASTNode Exp():

{ Token op;

ASTNode t1, t2; }

{

t1=Term() ( ( op=<PLUS> | op=<MINUS> ) t2=Term()

{ if (op.kind == PLUS)

t1 = new ASTPlus(t1,t2);

else t1 = new ASTSub(t1,t2);

}

) \*

{ return t1; }

}

ASTNode Term():

{Token op;

ASTNode t1, t2;}

{

t1 = Fact() ( ( op=<MULT> | op=<DIV> ) t2 = Term()

{ if (op.kind == MULT)

t1 = new ASTMult(t1,t2);

else t1 = new ASTDiv(t1,t2);

}

)?

{ return t1; }

}

ASTNode Fact():

{ Token n;

ASTNode t;}

{

(

n=<Num> {t = new ASTNum(Integer.parseInt(n.image));}

| <LPAR> t=Exp() <RPAR>

| <MINUS> t = Fact() { t=new ASTNeg(t); }

)

{ return t; }

}

int eval();

void compile(CodeBlock c);

}

class CodeBlock {

private static final String PREAMBULE = “ . . . “;

private static final String POS = “ . . . “;

List<String> code;

public CodeBlock() {

code = new LinkedList<String>();

}

void emit(String opcode) {

code.add(opcode);

}

void dump(PrintStream f) {

f.println(PREAMBULE);

for (String line : code) {

f.println(line);

System.out.println(line);

}

f.println(POS);

}

}

public class ASTPlus implements ASTNode {

ASTNode lhs, rhs;

public int eval() {

int v1 = lhs.eval();

int v2 = rhs.eval();

return v1 + v2;

}

public ASTPlus(ASTNode l, ASTNode r) {

lhs = l;

rhs = r;

}

@Override

public void compile(CodeBlock c) {

lhs.compile(c);

rhs.compile(c);

c.emit("iadd");

}

}

public class ICLCompiler {

public static void main(String args[]) {

Parser parser = new Parser(System.in);

CodeBlock code = new CodeBlock();

while (true) {

try {

ASTNode ast = parser.Start();

ast.compile(code);

code.dump(new PrintStream(new File("../ficheiro.txt")));

} catch (Exception e) {

System.out.println("Syntax Error!");

parser.ReInit(System.in);

}

}

}

}

public class ICLInterpreter {

public static void main(String args[]) {

Parser parser = new Parser(System.in);

ASTNode exp;

while (true) {

try {

exp = parser.Start();

System.out.println(exp.eval());

} catch (Exception e) {

System.out.println("Syntax Error!");

parser.ReInit(System.in);

}

}

}

}