**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

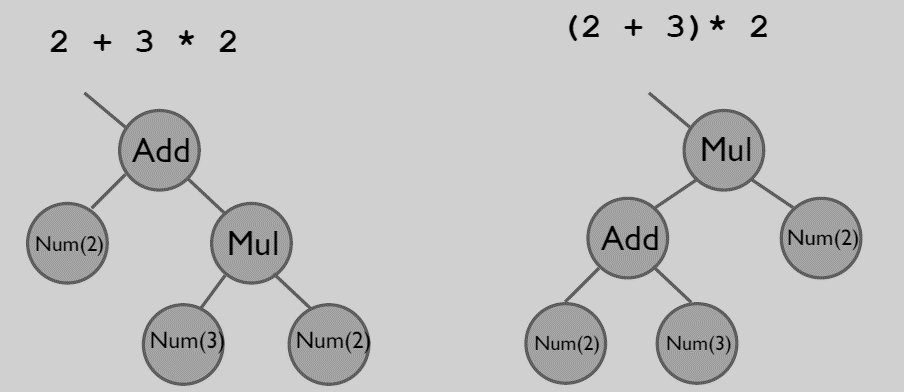
Concreta – 2+1

* 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

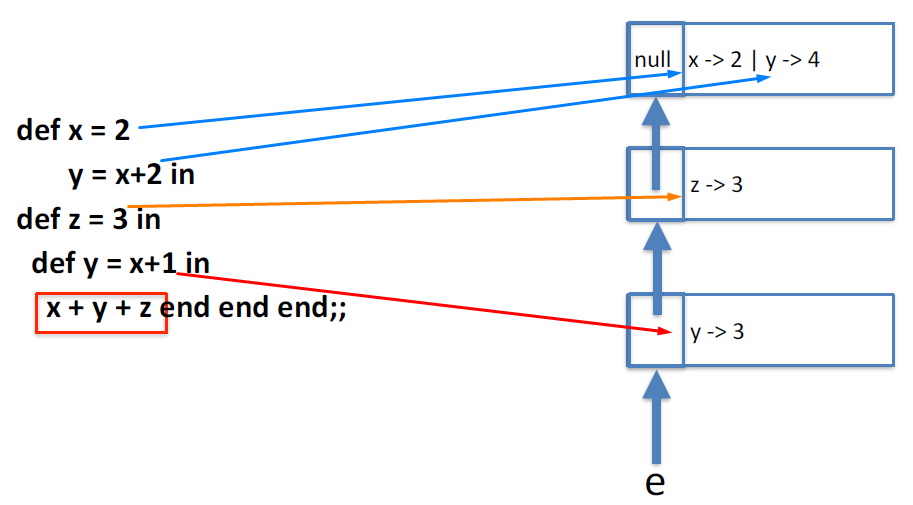
Instruções da JVM:

* 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 . . .

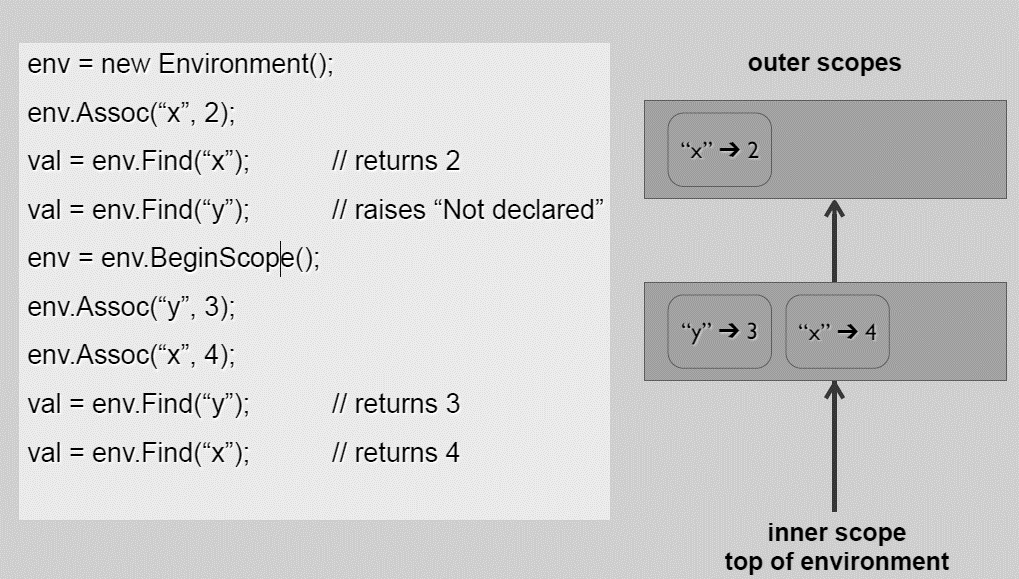
* **d**: a profundidade da stack de frames onde o identificador é declarado
* **s**: o slot na frame onde o valor é armazenado



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

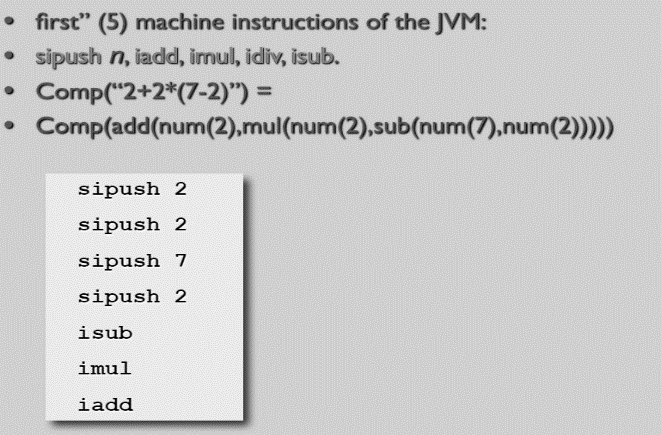
D(**y**) = (0,1)

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

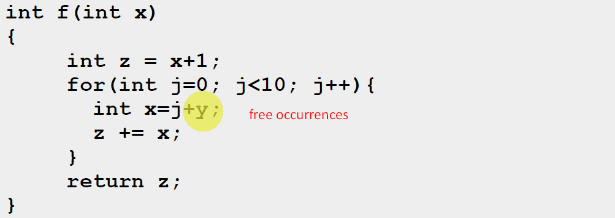
* O binding é defenido em contexto sintatico e criado por uma declaração (x = 1)

**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

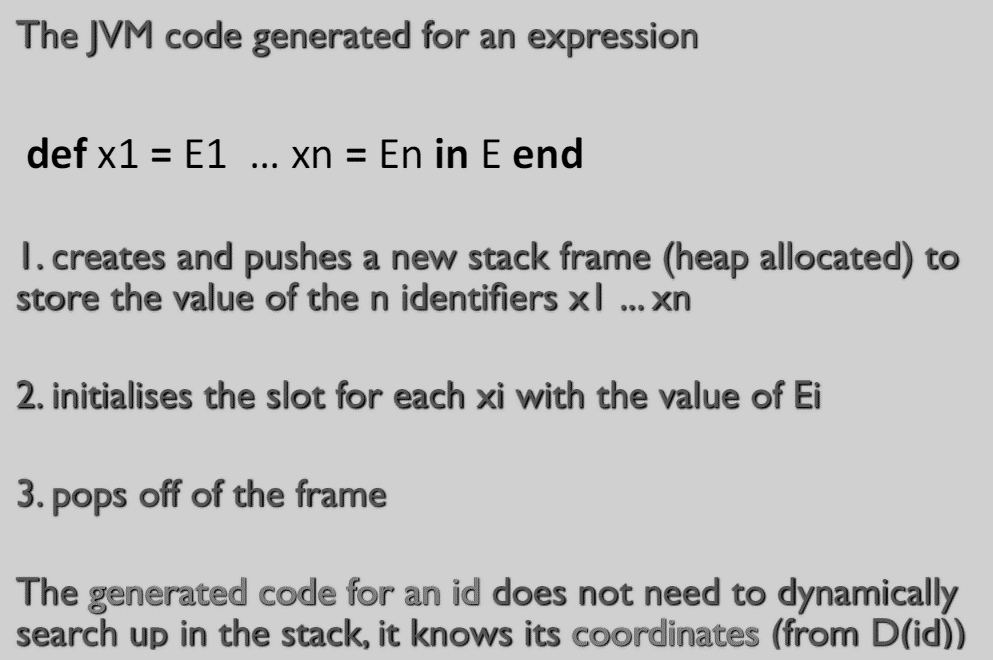
Na prática, é conveniente implementar ambientes usando uma estrutura de dados semelhante a uma pilha mutável chamada “spaghetti stack”.

• 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

**Fator -> (abstract syntax)**

**|** **num |** **true |** **false |** **id |** **string**

**| struct {** (**id=** EE)\* **} |** EE**.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

**| <fn>** <**id>**\* EE

**|** EE **(** EE\* **)**

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

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

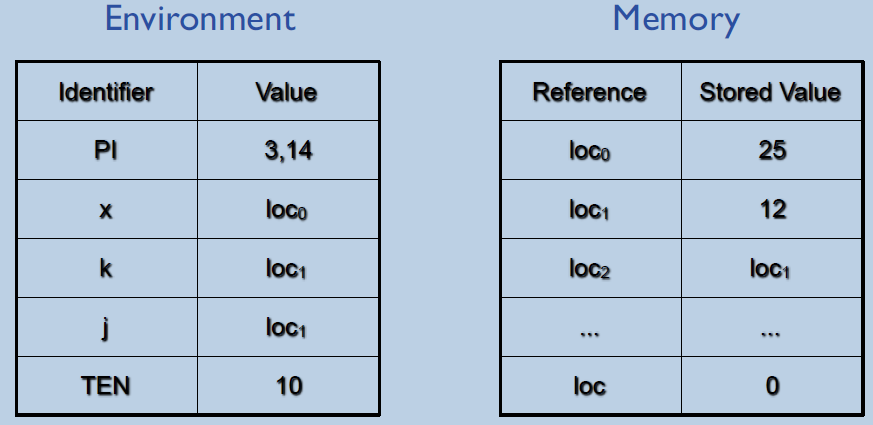
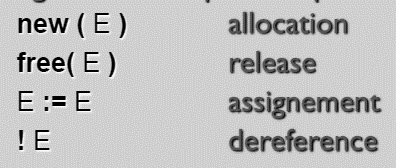
* E -> TE’
* E’ -> ε | + E
* T -> FT’
* T’ -> ε | \* T
* F -> num
* F -> ( E )
* F -> - F

EBNF (Extended BNF)

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

# Environment versus Memory

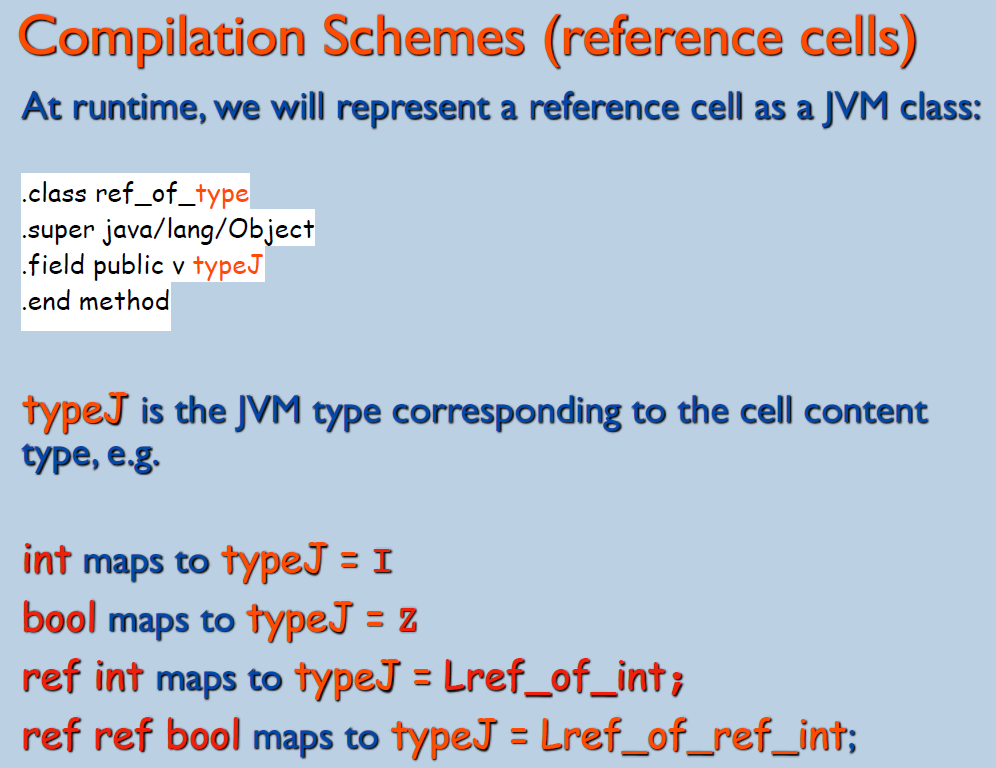
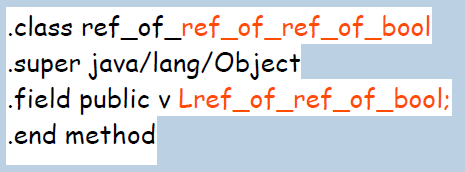
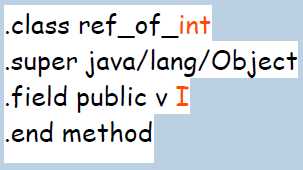
* O environment dá o valor associado a cada identifier declarado no programa e reflecte a estrutura estática de tal programa (nesting of scopes).
* O binding entre um identifier e o seu valor é fixa e imutável. O valor é bound apenas uma vez usando a operação assoc().
* A memória contém um conjunto de células mutáveis, cada célula é nomeada por um valor de referência e detém um valor.
* O valor armazenado numa referência pode ser alterado durante a execução, utilizando operações de atribuição (por exemplo, X := E)

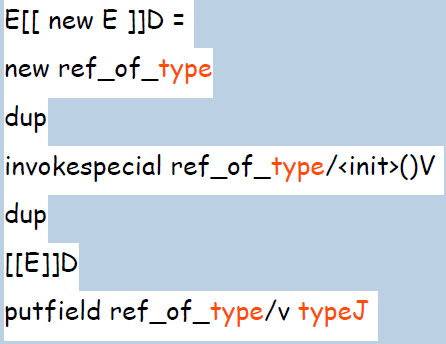
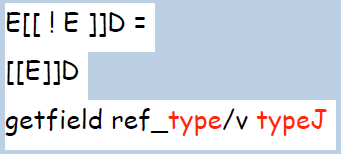
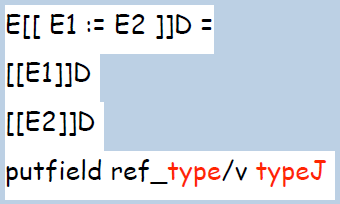
 

**Nota:** As referências são valores; a célula de memória pode ser bound a diferentes nomes (aliasing)

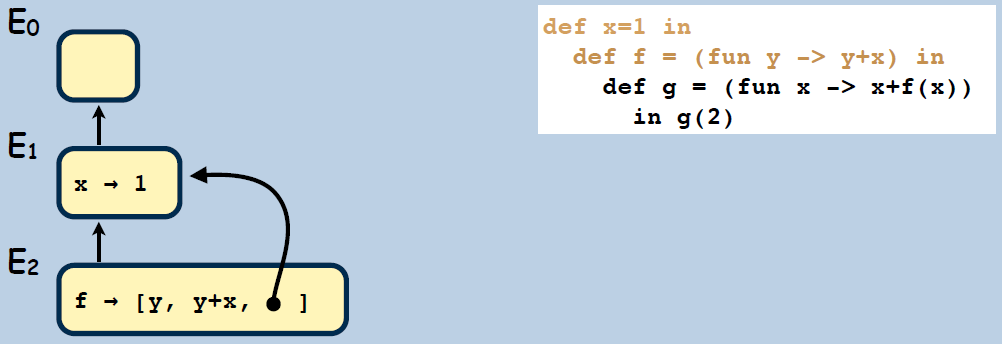
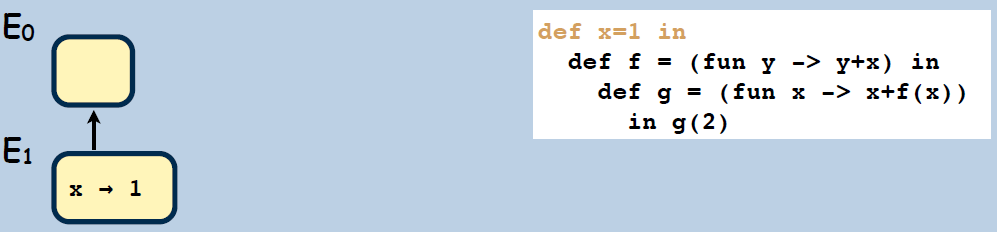
# Lifetime vs Scope

# A lifetime de uma memory cell é o tempo (durante a execução do programa) que intermedia a sua allocation new(...) e a sua libertação livre(...)

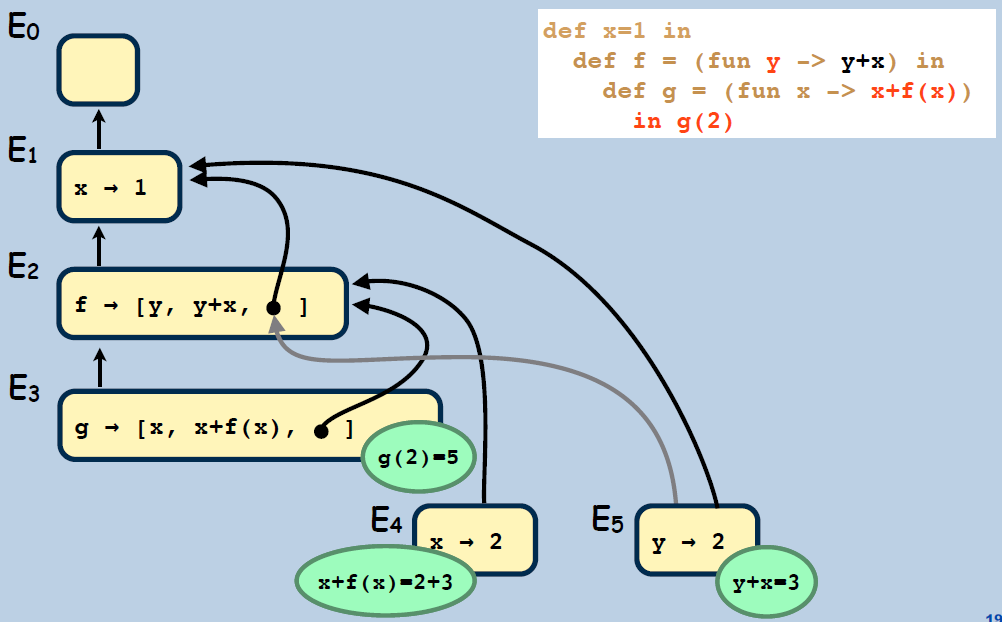
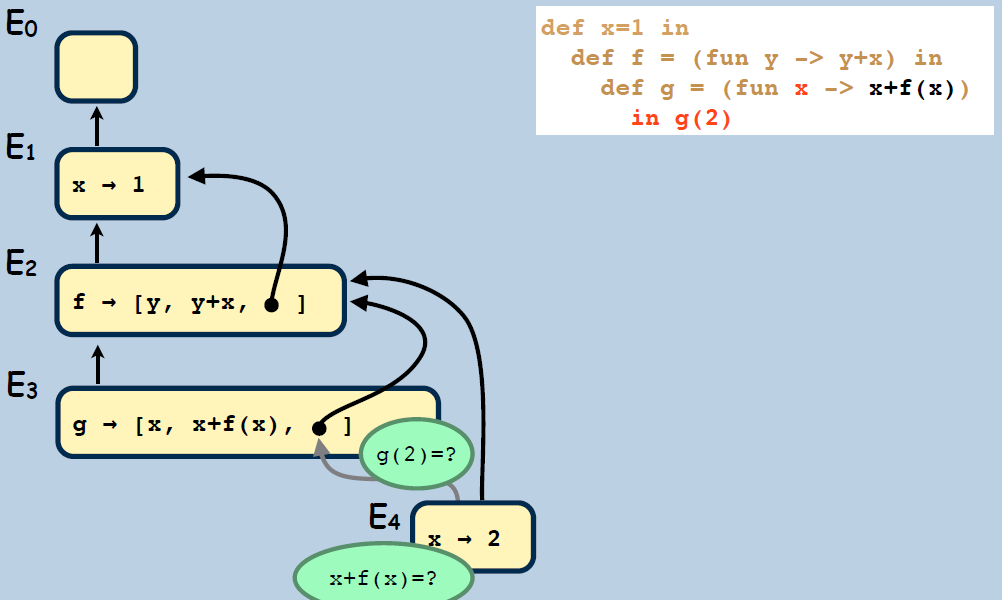
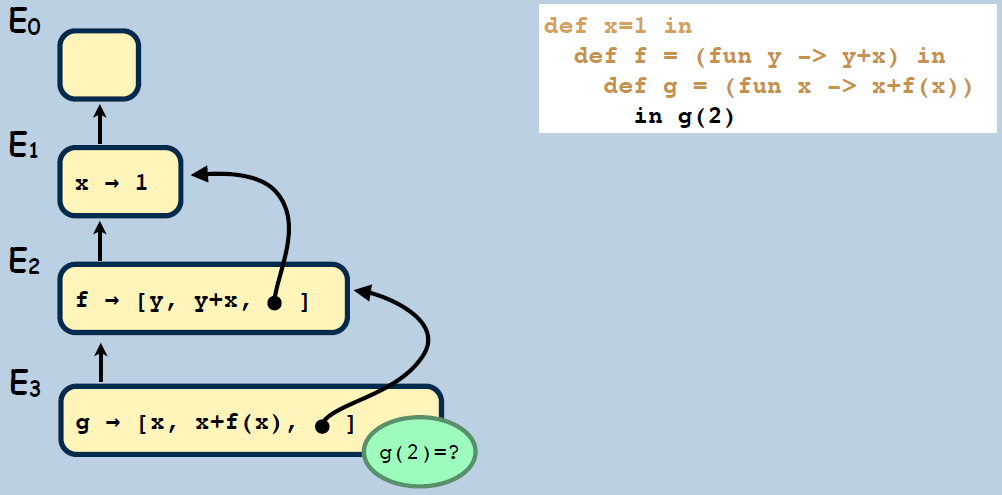
  

# Functions



2

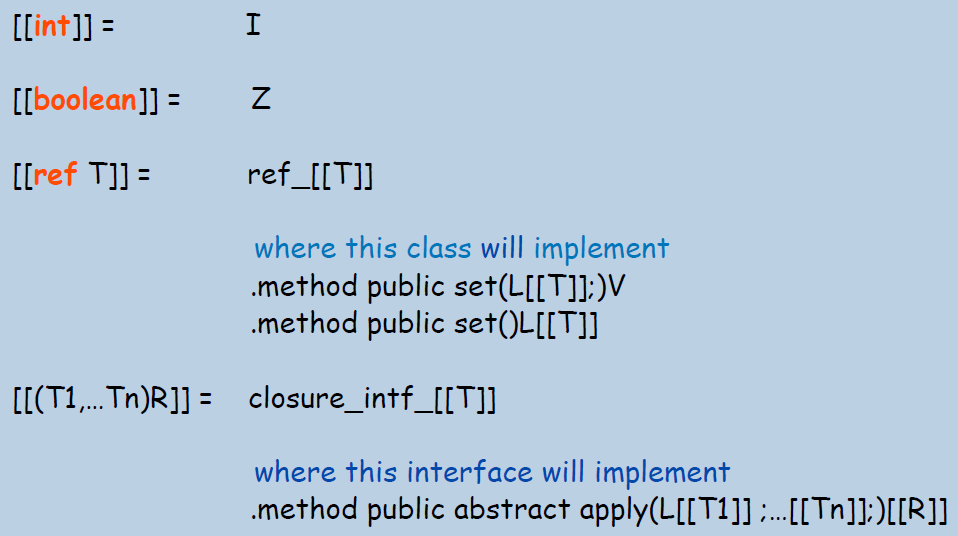
1



5

4

3

def

inc : (int)int = fun x:int -> x + 1 end

in

println inc(2+2)

end;;

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

.class public closure\_0

.super java/lang/Object

.implements closure\_interface\_int\_int

.field public sl Ljava/lang/Object;

.method public <init>()V

aload\_0

invokenonvirtual java/lang/Object/<init>()V

return

.end method

.class public frame\_1

.super java/lang/Object

.field public sl Ljava/lang/Object;

.field public v0 I

.method public <init>()V

aload\_0

invokenonvirtual java/lang/Object/<init>()V

return

.end method

# Project code

SKIP :

{

  " "

| "\t"

| "\r"

| "\n"

}

TOKEN :

{

  <COMMA : ",">

  |

  <FUN : "fn">

  |

  <MUT : "mut">

  |

  <TRUE : "true">

  |

  <FALSE : "false">

  |

  <IF : "if">

  |

  <THEN : "then">

  |

  <ELSE : "else">

  |

  <WHILE : "while">

  |

  <LET : "let">

  |

  <PRINTLN : "println">

  |

  <NEW : "new">

  |

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

  |

  <Num: (["0"-"9"]) + >

  |

  <PLUS : "+" >

  |

  <MINUS : "-">

  |

  <MULT : "\*">

  |

  < DIV : "/">

|

< LPAR : "(" >

  |

  < RPAR : ")" >

  |

  < LBRA : "{" >

  |

  < RBRA : "}" >

  |

  < RECEIVE : "=">

  |

  < SEQUENCE : ";">

  |

  < ASSIGN : ":=">

  |

  < EQUAL : "==">

  |

  < DIFFERENT : "~=">

  |

  < GREATER : ">">

  |

  < GREATEREQUAL : ">=">

  |

  < LESS : "<">

  |

  < LESSEQUAL : "<=">

  |

  < DEREFERENCER : "!">

  |

  < AND : "&&">

  |

  < OR : "||">

  |

  < NEG : "~">

  |

  < END: ";;" >

}

ASTNode Start():

{

  ASTNode t1;

}

{

   t1 = MainExp() <END>

   { return t1; }

}

ASTNode Sequence():

{ Token op; ASTNode t1, t2; }

{

t1=Assignment()( op=<SEQUENCE> t2=Assignment(){t1 = new ASTSequence(t1,t2);} )\*

{ return t1; }

}

ASTNode Assignment():

{ Token op; ASTNode t1, t2; }

{

t1=BooleanAdd()( op=<ASSIGN> t2=BooleanAdd(){t1 = new ASTAssign(t1,t2);} )\*

{ return t1; }

}

ASTNode BooleanAdd() :

{ Token op; ASTNode t1, t2; }

{

t1=BooleanMult()( op=<OR> t2=BooleanMult(){t1 = new ASTOr(t1,t2);} )\*

{ return t1; }

}

ASTNode BooleanMult():

{ Token op; ASTNode t1, t2; }

{

t1=Equal()( op=<AND> t2=Equal()

{

if(op.kind == AND)

t1 = new ASTAnd(t1,t2);

} )\*

{ return t1; }

}

ASTNode Equal():

{ Token op; ASTNode t1, t2; }

{

t1=Relops()( (op=<EQUAL> | op=<DIFFERENT>) t2=Relops()

{

if (op.kind == EQUAL)

t1 = new ASTEqual(t1,t2);

else if(op.kind == DIFFERENT)

t1 = new ASTDifferent(t1,t2);

} )\*

{ return t1; }

}

ASTNode Relops() :

{ Token op; ASTNode t1, t2; }

{

t1=Exp() ( ( op=<GREATER> | op=<GREATEREQUAL> | op=<LESS> | op=<LESSEQUAL>) t2=Exp(){

if (op.kind == GREATER)

t1 = new ASTGreater(t1,t2);

else if (op.kind == GREATEREQUAL)

t1 = new ASTGreaterOrEqual(t1,t2);

else if (op.kind == LESS)

t1 = new ASTLess(t1,t2);

else if (op.kind == LESSEQUAL)

t1 = new ASTLessOrEqual(t1,t2);

} )?

{ return t1; }

}

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 x;

ASTNode t1,t2,t3;

Map<String,ASTNode> map;

}

{

( x= <Num> { t1 = new ASTNum(Integer.parseInt(x.image)); }

| (x = <TRUE> | x = <FALSE>) {t1 = new ASTBool(Boolean.parseBoolean(x.image));}

| x = <Id> { t1 = new ASTId(x.image);}

| <PRINTLN> t1=Fact() {t1 = new ASTPrintln(t1);}

| <NEW> t1 = Fact() {t1 = new ASTNew(t1);}

| <NEG> t1=Fact() {t1 = new ASTNot(t1);}

| <DEREFERENCER> t1=Fact() {t1 = new ASTDeferencer(t1);}

| <LPAR> t1=MainExp() <RPAR>

| <MINUS> t1 = Fact() { t1=new ASTNeg(t1); }

| <LBRA> {map = new HashMap<>();}

(<LET> (<MUT>)? x=<Id> <RECEIVE> t1=Assignment() <SEQUENCE> { map.put(x.image,t1);})+

t1 = MainExp()

<RBRA>

{ t1 = new ASTDef(map,t1); }

| <IF> t1=MainExp() <LBRA> t2 = MainExp() <RBRA> <LBRA> t3 = MainExp() <RBRA>{t1 = new ASTIf(t1,t2,t3);}

| <WHILE> t1=MainExp() <LBRA> t2=MainExp() <RBRA> {t1 = new ASTWhile(t1,t2);})

| <FUN> {List<String> l = new LinkedList<>();} <ID> <LPAR>(

x = <ID> {}(<COMMA> <ID> )\* )?

<RPAR>t1=Assignment() <LBRA>

<RBRA>

{ return t1; }

}

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

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

public interface IValue {

String toString();

}

---------------------------------------------------------------------------

public class VCell implements IValue { //VInt e VBool são parcidos

private IValue v;

public VCell(IValue v) { this.v = v; } //ref

public IValue getVal() { return v; }

public void set(IValue v) { this.v = v; }

}

---------------------------------------------------------------------------

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

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

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) ))))