**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

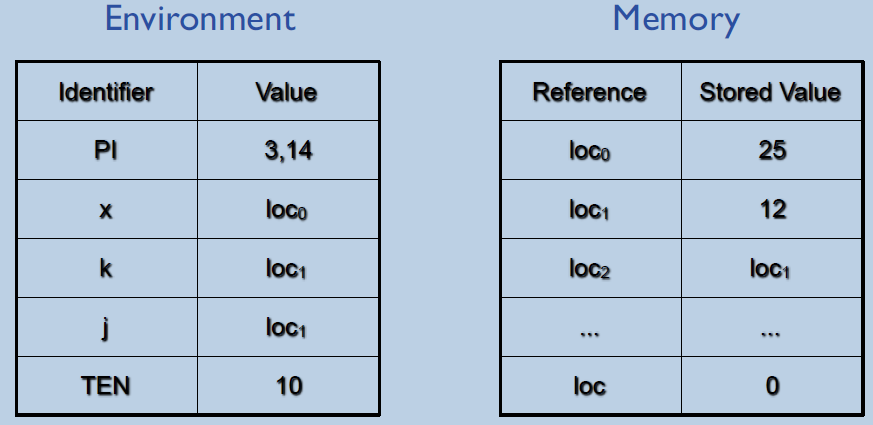
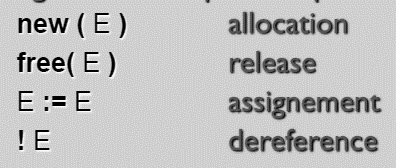
* define a forma como as expressões e os programas são efetivamente escritos em termos de formatação, sequências de caracteres (ascii/unicode), etc…

Abstrata – add(num(2), num(1))

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

# 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)

# Complilation of CALCI

**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 }

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

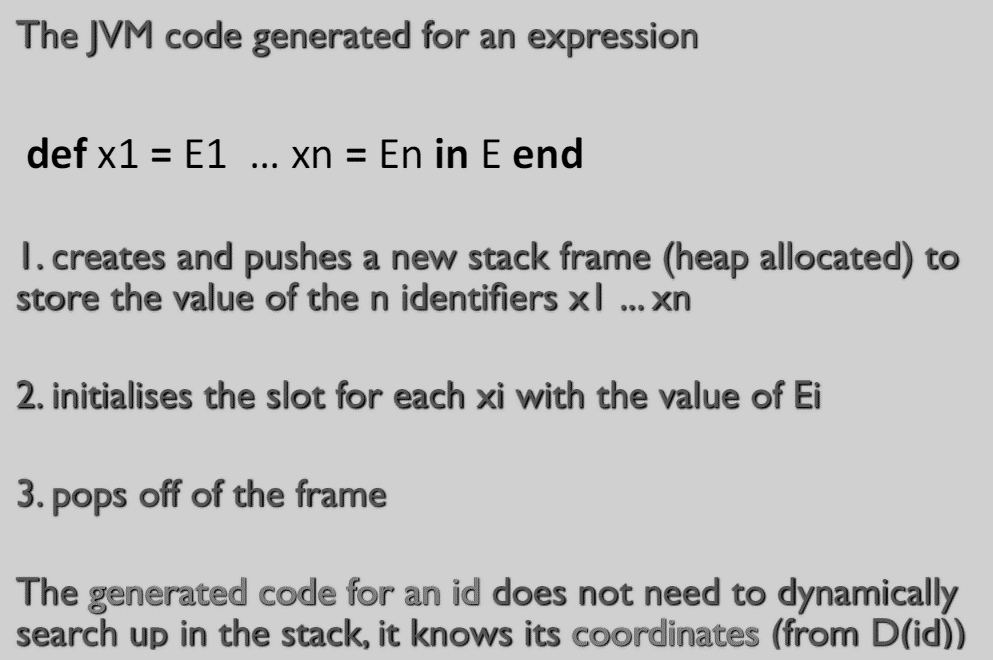
getfield frame\_1/sl Lframe\_0;

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

dup

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

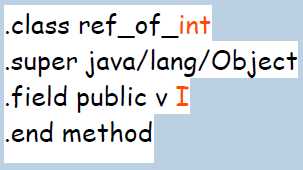
putfield frame\_1/v0 I

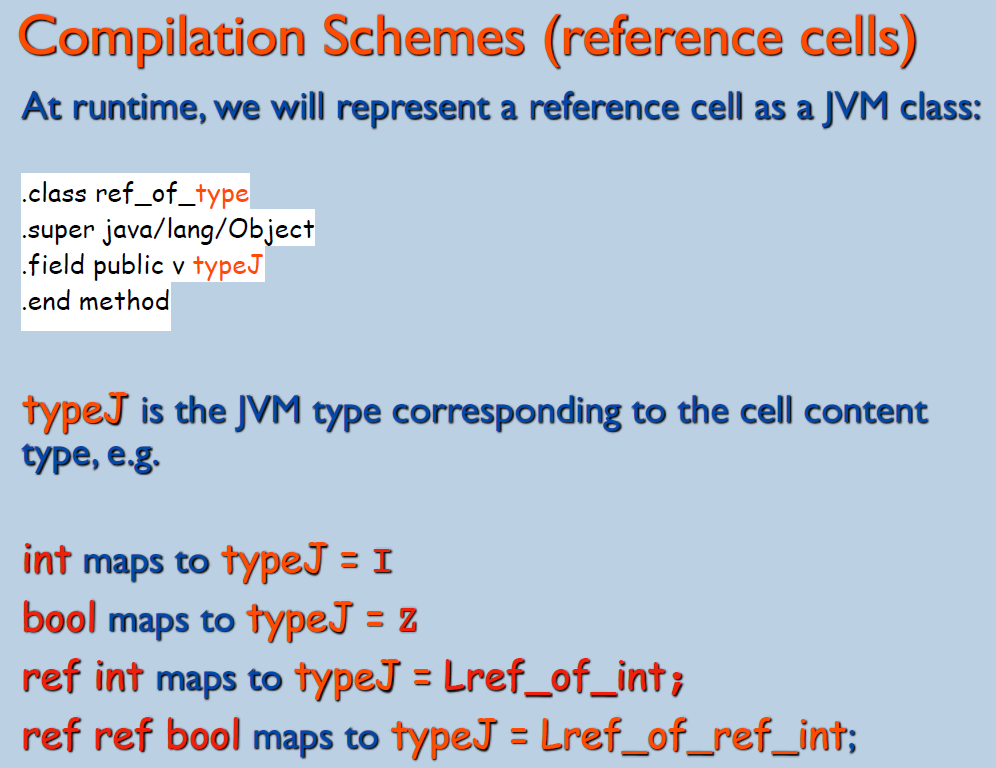
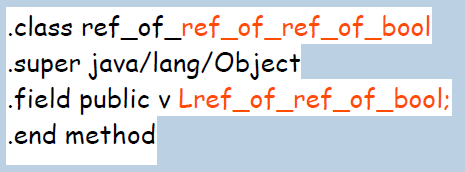
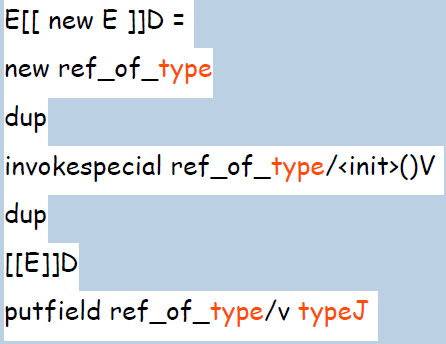
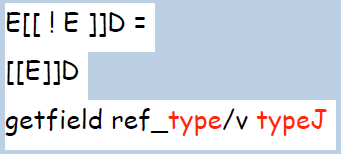
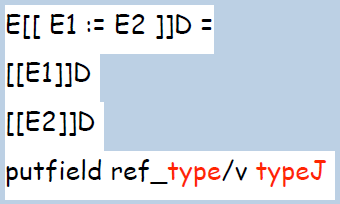
# Lifetime vs Scope

# 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(...)

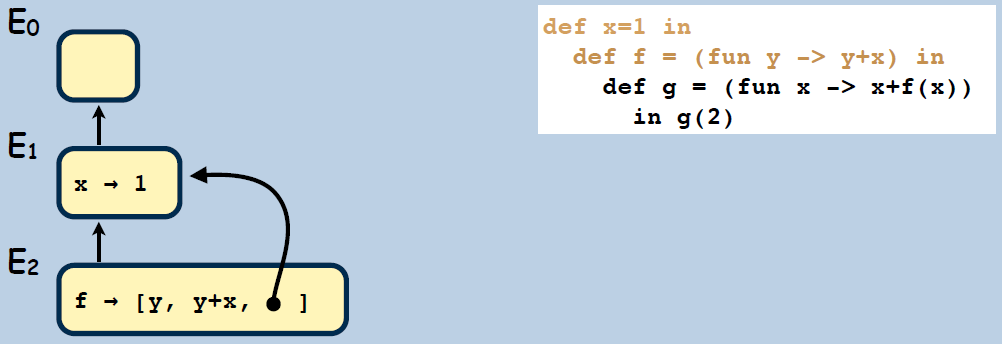
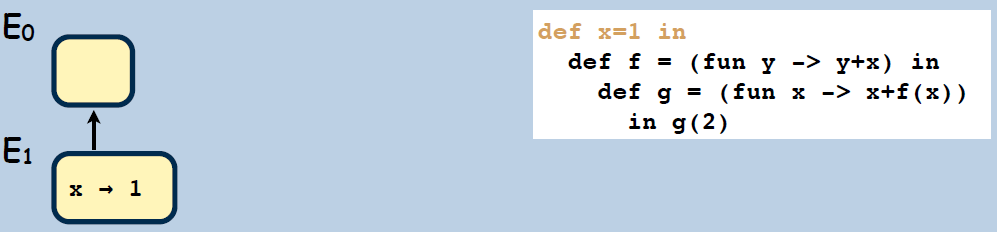
# Dynamic vs Static typecheck

Dynamic typecheck é o equivalente ao runtime typecheck que é um check de erros feito à medida que o programa é executado. O static typecheck ocorre em tempo de compilação, capturando erros antes do programa correr.



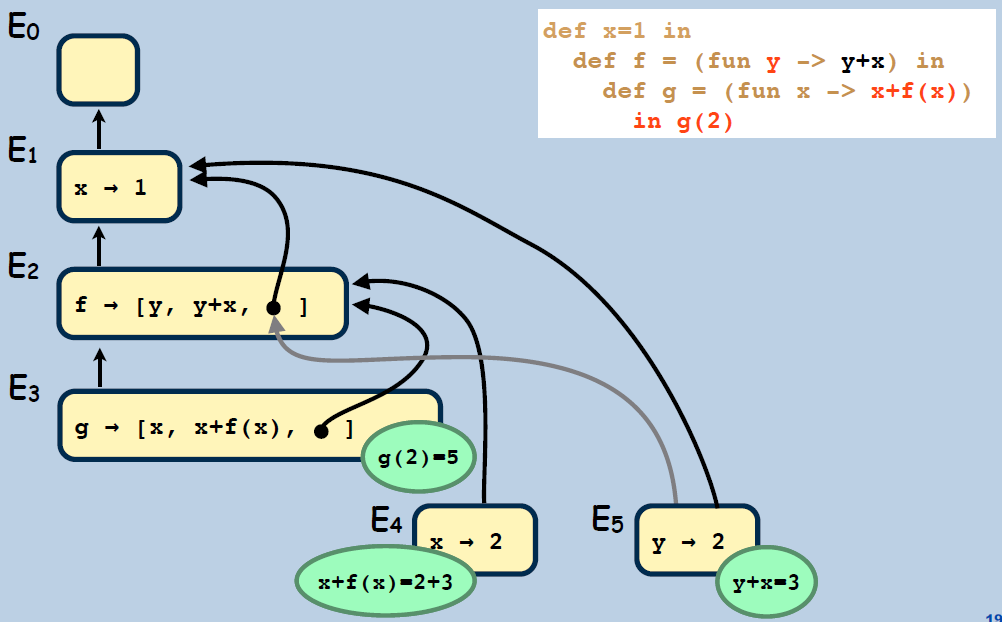
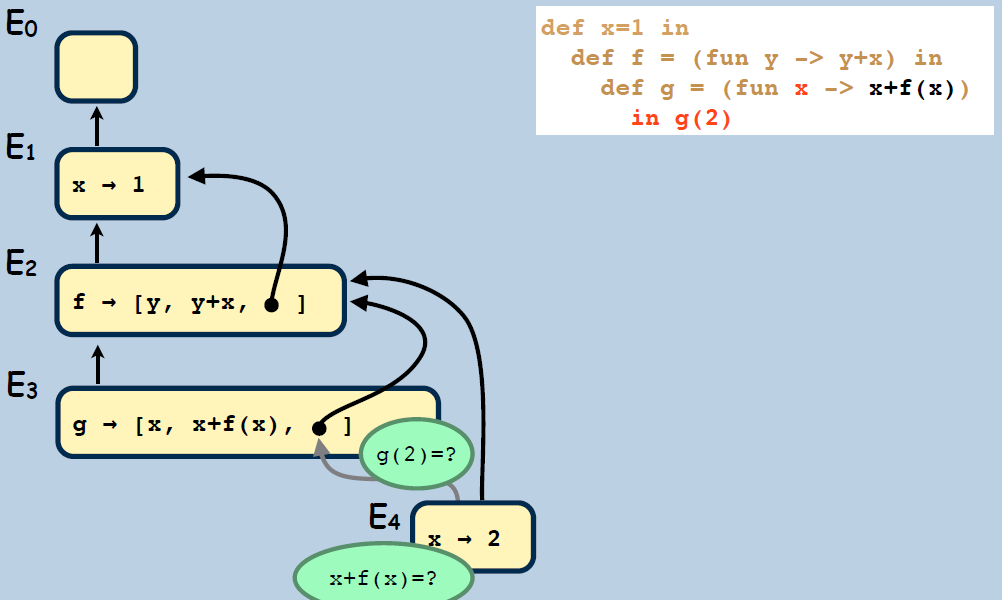
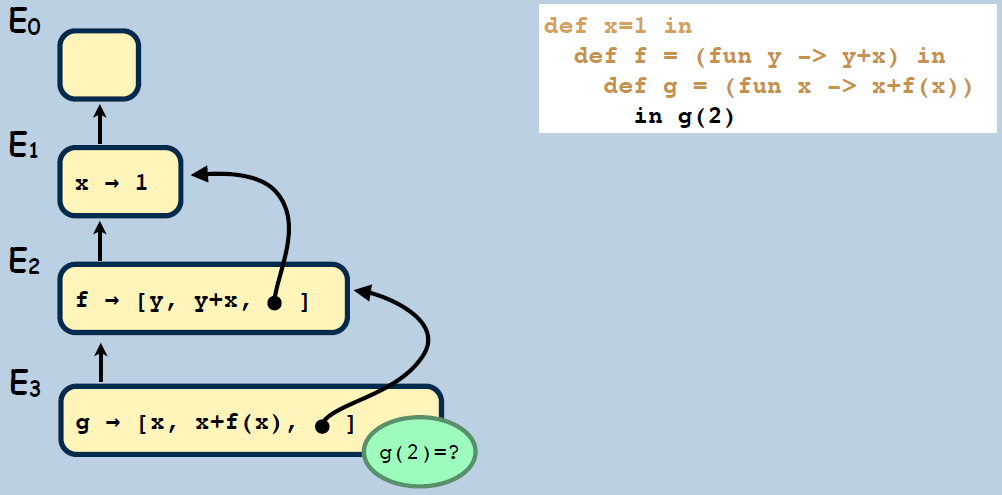


# Functions



2

1



5

4

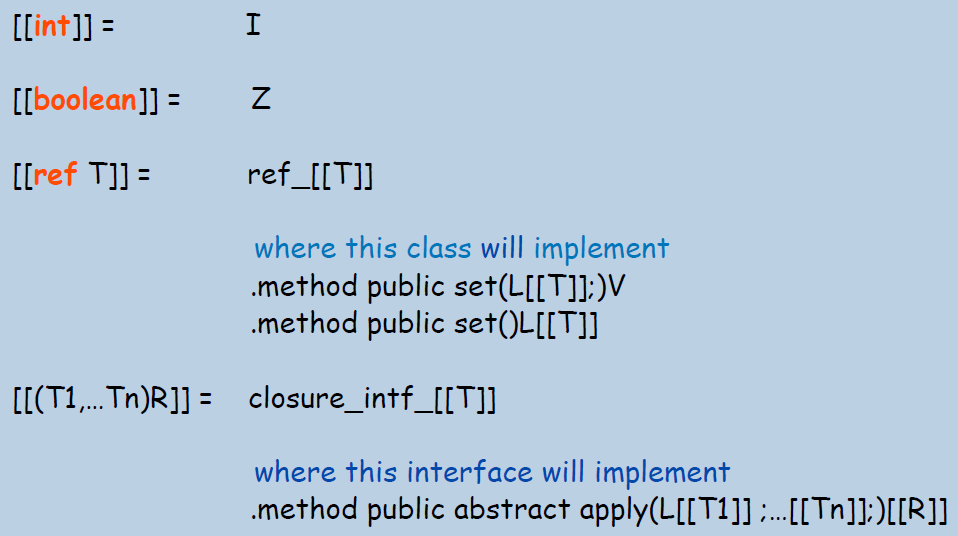
3

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> (x=<Id> <RECEIVE> t1=Assignment() <SEQUENCE> { map.put(x.image,t1);} |

(<MUT> x=<Id> <RECEIVE> t1=Assignment() <SEQUENCE> {t1 = new ASTNew(t1) ;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> {l.add(x.image);} (<COMMA> x = <Id> {l.add(x.image);})\*

<RPAR><LBRA> t1=Assignment() <RBRA>

)

{ return t1; }

}

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

public interface **IValue** {

String toString();

}

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

public class **VCell** implements IValue {

private IValue v;

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

public IValue getVal() { return v; }

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

}

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

public class **VInt** implements IValue {

private int v;

public VInt(int v) { this.v = v; }

public int getVal() { return v; }

public String toString() { return "" + v; }

}

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

public interface **IType** {

String toString();

String getJvmType();

}

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

public class **TypeRef** implements IType {

private IType v;

private String jvmType;

public TypeRef(IType v) {

this.v = v;

this.jvmType = "";

IType aux = v;

while (aux instanceof TypeRef) {

jvmType += "ref\_of\_";

aux = ((TypeRef) v).getVal();

}

if (aux instanceof TypeInt) { jvmType += "int"; }

else { jvmType += "bwool"; }

}

public IType getVal() { return v; }

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

public String getJvmType() { return jvmType; }

}

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

public class **TypeInt** implements IType {

private int v;

public TypeInt(int v) { this.v = v; }

public int getVal() { return v; }

public String toString() { return "" + v; }

public String getJvmType() { return "I"; }

}

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

public class **ASTBool** implements ASTNode { //a diff. para o ASTInt é o boolean -> int

private boolean val;

public ASTBool(boolean n) { this.val = n; }

public IValue eval(Environment<IValue> e) { return new VBool(val); }

public void compile(MainBlock c, Environment<Coordinates> e) {

int v = (val) ? 1 : 0;

c.emit(“sipush ” + v);

}

public IType typecheck(Environment<IType> env) {

return new TypeBool(val);

}

}

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

public class **ASTAssign** implements ASTNode {

private ASTNode lhs;

private ASTNode rhs;

public ASTAssign(ASTNode lhs, ASTNode rhs) { this.lhs = lhs; this.rhs = rhs; }

public IValue eval(Environment<IValue> e) {

IValue v1 = lhs.eval(e);

if (v1 instanceof VCell) {

IValue v2 = rhs.eval(e);

((VCell) v1).set(v2);

return v2;

}

throw new RuntimeException("illegal arguments to := operator");

public void compile(MainBlock c, Environment<Coordinates> env) {

lhs.compile(c, env);

rhs.compile(c, env);

IType iType = typecheck(new Environment<IType>(null, 0));

String classType = "";

String typeJ = iType.getJvmType();

if (iType instanceof TypeInt) {

classType = "int";

} else if (iType instanceof TypeBool) {

classType = "bool";

} else {

classType = iType.getJvmType();

typeJ = "L" + typeJ;

}

c.emit(String.format(“putfield ref\_of\_%s/v %s”, classType, typeJ));

if (iType instanceof TypeRef) {

classType = "ref\_of\_" + classType;

}

CellBlock cBlock = null;

try {

cBlock = new CellBlock(new PrintStream(new File(String.format(FILE\_PATH\_TO\_BLOCKS, classType))), classType,

typeJ);

} catch (FileNotFoundException e) {

e.printStackTrace();

}

}

public IType typecheck(Environment<IType> env) {

IType v1 = lhs.typecheck(env);

if (v1 instanceof TypeRef) {

IType v2 = rhs.typecheck(env);

((TypeRef) v1).set(v2);

return v2;

}

throw new RuntimeException("illegal arguments to := operator");

}

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

public class **ASTDeferencer** implements ASTNode {

private ASTNode node;

public ASTDeferencer(ASTNode node) { this.node = node; }

public IValue eval(Environment<IValue> e) {

IValue v1 = node.eval(e);

if (v1 instanceof VCell) {

return ((VCell) v1).getVal();

}

throw new RuntimeException("illegal arguments to ! operator");

}

public void compile(MainBlock c, Environment<Coordinates> e) {

node.compile(c, e);

IType iType = typecheck(new Environment<IType>(null, 0));

String classType = "";

String typeJ = iType.getJvmType();

if (iType instanceof TypeInt) {

classType = "int";

} else if (iType instanceof TypeBool) {

classType = "bool";

} else {

classType = iType.getJvmType();

typeJ = "L" + typeJ;

}

c.emit(String.format(“getfield ref\_of\_%s/v %s”, classType, typeJ)); // TODO

}

public IType typecheck(Environment<IType> env) {

IType v1 = node.typecheck(env);

if (v1 instanceof TypeRef) {

return ((TypeRef) v1).getVal();

}

throw new RuntimeException("illegal arguments to ! operator");

}

}

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

public class **ASTDifferent** implements ASTNode {

private ASTNode lhs;

private ASTNode rhs;

public ASTDifferent(ASTNode lhs, ASTNode rhs) { this.lhs = lhs; this.rhs = rhs; }

public IValue eval(Environment<IValue> e) {

IValue v1 = lhs.eval(e);

if (v1 instanceof VBool) {

IValue v2 = rhs.eval(e);

if (v2 instanceof VBool) {

if (((VBool) v1).getVal() != ((VBool) v2).getVal())

return new VBool(true);

return new VBool(false);

}

} else if (v1 instanceof VInt) {

IValue v2 = rhs.eval(e);

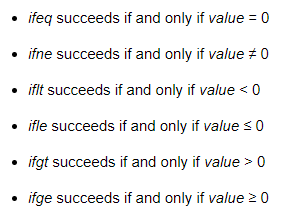
if (v2 instanceof VInt) {

if (((VInt) v1).getVal() != ((VInt) v2).getVal())

return new VBool(true);

return new VBool(false);

}

 }

throw new RuntimeException("illegal arguments to ~= operator");

}

public void compile(MainBlock c, Environment<Coordinates> e) {

lhs.compile(c, e);

rhs.compile(c, e);

c.emit("isub");

c.emit(String.format("ifne %s", “L1”));

c.emit("sipush " + 0);

c.emit(String.format("goto %s", “L2”));

c.emit(String.format("%s: %s", “L1”, "sipush " + 1));

c.emit(String.format("%s: %s", “L2”, ""));

}

public IType typecheck(Environment<IType> env) {

IType v1 = lhs.typecheck(env);

if (v1 instanceof TypeBool) {

IType v2 = rhs.typecheck(env);

if (v2 instanceof TypeBool) {

if (((TypeBool) v1).getVal() != ((TypeBool) v2).getVal())

return new TypeBool(true);

return new TypeBool(false);

}

} else if (v1 instanceof TypeInt) {

IType v2 = rhs.typecheck(env);

if (v2 instanceof TypeInt) {

if (((TypeInt) v1).getVal() != ((TypeInt) v2).getVal())

return new TypeBool(true);

return new TypeBool(false);

}

}

throw new RuntimeException("illegal arguments to ~= operator");

}

}

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

public class **ASTDiv** implements ASTNode {

private ASTNode lhs, rhs;

public ASTDiv(ASTNode lhs, ASTNode rhs) {this.lhs = lhs; this.rhs = rhs; }

public IValue eval(Environment<IValue> e) {

IValue v1 = lhs.eval(e);

if (v1 instanceof VInt) {

IValue v2 = rhs.eval(e);

if (v2 instanceof VInt) {

return new VInt(((VInt) v1).getVal() / ((VInt) v2).getVal());

}

}

throw new RuntimeException("illegal arguments to / operator");

}

public void compile(MainBlock c, Environment<Coordinates> e) { // parcido com o mult,add, e sub

lhs.compile(c, e);

rhs.compile(c, e);

c.emit("idiv");

}

public IType typecheck(Environment<IType> env) {

IType v1 = lhs.typecheck(env);

if (v1 instanceof TypeInt) {

IType v2 = rhs.typecheck(env);

if (v2 instanceof TypeInt) {

return new TypeInt(((TypeInt) v1).getVal() / ((TypeInt) v2).getVal());

}

}

throw new RuntimeException("illegal arguments types to - operator");

}

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

public class **ASTId** implements ASTNode {

private String id;

public ASTId(String id) { this.id = id; }

public IValue eval(Environment<IValue> e) { return e.find(id); }

public void compile(MainBlock c, Environment<Coordinates> e) {

Coordinates coord = e.find(id);

int goalDepth = coord.getDepth();

int currDepth = e.depth();

c.emit("aload 3");

while (currDepth > goalDepth) {

c.emit(String.format("getfield frame%d/sl Lframe%d;", currDepth, --currDepth));

}

c.emit(String.format("getfield frame%d/%s", goalDepth, coord.getFieldName()));

}

public IType typecheck(Environment<IType> env) { return // TODO }

}

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

public class **ASTIf** implements ASTNode {

private ASTNode evaluate;

private ASTNode caseTrue;

private ASTNode caseFalse;

public ASTIf(ASTNode evaluate, ASTNode caseTrue, ASTNode caseFalse) {

this.evaluate = evaluate;

this.caseTrue = caseTrue;

this.caseFalse = caseFalse;

}

public IValue eval(Environment<IValue> e) {

IValue v1 = evaluate.eval(e);

if (v1 instanceof VBool) {

boolean evaluation = ((VBool) v1).getVal();

if (evaluation)

return caseTrue.eval(e);

else

return caseFalse.eval(e);

}

throw new RuntimeException("illegal arguments to if operator");

}

public void compile(MainBlock c, Environment<Coordinates> e) {

evaluate.compile(c, e);

c.emit(String.format("ifeq %s", ”L1”));

caseTrue.compile(c, e);

c.emit(String.format("goto %s", “L2”));

c.emit(String.format("%s: %s", “L1”, ""));

caseFalse.compile(c, e);

c.emit(String.format("%s: %s", “L2”, ""));

}

public IType typecheck(Environment<IType> env) {

IType v1 = evaluate.typecheck(env);

if (v1 instanceof TypeBool) {

boolean evaluation = ((TypeBool) v1).getVal();

if (evaluation){ return caseTrue.typecheck(env); }

else{ return caseFalse.typecheck(env); }

}

throw new RuntimeException("illegal arguments to if operator");

}

}

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

public class **ASTNeg** implements ASTNode {

private ASTNode exp;

public ASTNeg(ASTNode e) {

exp = e;

}

public IValue eval(Environment<IValue> e) {

IValue v = exp.eval(e);

if (v instanceof VInt)

return new VInt(-((VInt) v).getVal());

throw new RuntimeException("illegal arguments to simetric operator");

}

public void compile(MainBlock c, Environment<Coordinates> e) {

exp.compile(c, e);

c.emit("ineg");

}

public IType typecheck(Environment<IType> env) {

IType v = exp.typecheck(env);

if (v instanceof TypeInt)

return new TypeInt(-((TypeInt) v).getVal());

throw new RuntimeException("illegal arguments to simetric operator");

}

}

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

public class **ASTNew** implements ASTNode {

private ASTNode arg;

public ASTNew(ASTNode arg) {this.arg = arg; }

public IValue eval(Environment<IValue> env) {

IValue v1 = arg.eval(env);

return new VCell(v1);

}

public void compile(MainBlock c, Environment<Coordinates> e) {

IType iType = typecheck(new Environment<IType>(null, 0));

String classType = "";

String typeJ = iType.getJvmType();

if (iType instanceof TypeInt) {

classType = "int";

} else if (iType instanceof TypeBool) {

classType = "bool";

} else {

classType = iType.getJvmType();

typeJ = "L" + typeJ;

}

c.emit(String.format("new ref\_of\_%s", classType));

c.emit("dup");

c.emit(String.format("invokespecial ref\_of\_%s/<init>()V", classType));

c.emit("dup");

arg.compile(c, e);

c.emit(String.format("putfield ref\_of\_%s/v %s", classType, typeJ));

}

public IType typecheck(Environment<IType> env) {

IType v1 = arg.typecheck(env);

return new TypeRef(v1);

}

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

public class **ASTNot** implements ASTNode {

ASTNode lhs;

public ASTNot(ASTNode lhs) { this.lhs = lhs; }

public IValue eval(Environment<IValue> e) {

IValue v1 = lhs.eval(e);

if (v1 instanceof VBool) {

if (((VBool) v1).getVal())

return new VBool(false);

return new VBool(true);

}

throw new RuntimeException("illegal arguments to ~ operator");

}

public void compile(MainBlock c, Environment<Coordinates> e) {

lhs.compile(c, e);

c.emit(String.format("ifeq %s", “L1”));

c.emit(String.format("sipush "+ 0));

c.emit(String.format("goto %s", L2));

c.emit(String.format("%s: %s", L1, "sipush "+ 1));

c.emit(String.format("%s: %s", L2, ""));

}

public IType typecheck(Environment<IType> env) {

IType v1 = lhs.typecheck(env);

if (v1 instanceof TypeBool) {

if (((TypeBool) v1).getVal())

return new TypeBool(false);

return new TypeBool(true);

}

throw new RuntimeException("illegal arguments to ~ operator");

}

}

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

public class **ASTPrintln** implements ASTNode {

private ASTNode arg;

public ASTPrintln(ASTNode arg) { this.arg = arg; }

public IValue eval(Environment<IValue> env) {

IValue v1 = arg.eval(env);

System.out.println(v1.toString());

return v1;

}

public void compile(MainBlock c, Environment<Coordinates> e) {

typecheck(new Environment<IType>(null, 0));

c.emit(String.format("getstatic java/lang/System/out Ljava/io/PrintStream;"));

arg.compile(c, e);

c.emit(String.format("invokestatic java/lang/String/valueOf(I)Ljava/lang/String;"));

c.emit(String.format("invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V"));

}

public IType typecheck(Environment<IType> env) {

IType v1 = arg.typecheck(env);

System.out.println(v1.toString());

return v1;

}

}

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

public class **ASTWhile** implements ASTNode {

private ASTNode stopCondition;

private ASTNode codeToRun;

public ASTWhile(ASTNode stopCondition, ASTNode codeToRun) {

this.stopCondition = stopCondition;

this.codeToRun = codeToRun;

}

public IValue eval(Environment<IValue> e) {

IValue v1 = stopCondition.eval(e);

if (v1 instanceof VBool) {

IValue result = null;

while (((VBool) stopCondition.eval(e)).getVal()) {

result = codeToRun.eval(e);

}

return result;

}

throw new RuntimeException("illegal arguments to while operator");

}

public void compile(MainBlock c, Environment<Coordinates> e) {

c.emit(String.format("%s: %s", “L1”, ""));

stopCondition.compile(c, e);

c.emit(String.format("ifeq %s", “L2”));

codeToRun.compile(c, e);

c.emit("pop");

c.emit(String.format("goto %s", “L1”));

c.emit(String.format("%s: %s", “L2”, ""));

}

public IType typecheck(Environment<IType> env) {

IType v1 = stopCondition.typecheck(env);

if (v1 instanceof TypeBool) {

IType result = null;

while (((TypeBool) stopCondition.typecheck(env)).getVal()) {

result = codeToRun.typecheck(env);

}

return result;

}

throw new RuntimeException("illegal arguments to while operator");

}

}

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

public class **ASTSequence** implements ASTNode {

private ASTNode lhs;

private ASTNode rhs;

public ASTSequence(ASTNode lhs, ASTNode rhs) { this.lhs = lhs; this.rhs = rhs; }

public IValue eval(Environment<IValue> e) {

lhs.eval(e);

return rhs.eval(e);

}

public void compile(MainBlock c, Environment<Coordinates> e) {

lhs.compile(c, e);

rhs.compile(c, e);

}

public IType typecheck(Environment<IType> env) {

lhs.typecheck(env);

return rhs.typecheck(env);

}

}

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

public class **ASTDef** implements ASTNode {

private Map<String, ASTNode> init;

private ASTNode body;

private List<String> typesOfArgs;

public ASTDef(Map<String, ASTNode> init, ASTNode body) {

this.init = init;

this.body = body;

typesOfArgs = new LinkedList<>();

}

public IValue eval(Environment<IValue> env) {

env = env.beginScope();

IValue v;

for (Entry<String, ASTNode> exp : init.entrySet()) {

v = exp.getValue().eval(env);

env.assoc(exp.getKey(), v);

}

v = body.eval(env);

env.endScope();

return v;

}

public void compile(MainBlock c, Environment<Coordinates> env) {

typecheck(new Environment<IType>(null, 0));

env = env.beginScope();

String frame = String.format(FRAME\_FORMAT, env.depth());

String previousFrame = frameInit(frame, env, c);

FrameBlock fBlock = createFrameBlock(frame, previousFrame);

frameAtribs(fBlock, frame, c, env);

body.compile(c, env);

framePopOf(frame, previousFrame, c, env);

fBlock.createFrame();

env.endScope();

}

public IType typecheck(Environment<IType> env) {

env = env.beginScope();

IType v;

for (Entry<String, ASTNode> exp : init.entrySet()) {

v = exp.getValue().typecheck(env);

env.assoc(exp.getKey(), v);

typesOfArgs.add(v.getJvmType());

}

v = body.typecheck(env);

env.endScope();

return v;

}

// Private methods

private String framePopOf(String frame, String previousFrame, MainBlock c, Environment<Coordinates> env) {

String framePopOff = "";

if (env.depth() == 1) {

framePopOff = String.format(FRAME\_POPOFF\_FORMAT, frame, JAVA\_LANG\_OBJECT);

} else {

framePopOff = String.format(FRAME\_POPOFF\_FORMAT, frame, previousFrame);

}

c.emit(framePopOff);

return framePopOff;

}

private void frameAtribs(FrameBlock fBlock, String frame, MainBlock c, Environment<Coordinates> env) {

int atribIndex = 0;

for (Entry<String, ASTNode> exp : init.entrySet()) {

String atribName = String.format(ATRIB\_NAME\_FORMAT, atribIndex++, typesOfArgs.get(atribIndex - 1));

String putField = String.format(PUT\_FIELD\_FORMAT, frame, atribName);

fBlock.addVariable(atribName);

c.emit(ALOAD\_REGIST);

exp.getValue().compile(c, env);

c.emit(putField);

env.assoc(exp.getKey(), new Coordinates(env.depth(), atribName));

}

}

private String frameInit(String frame, Environment<Coordinates> env, MainBlock c) {

String previousFrame = null;

if (env.depth() == 1) {

previousFrame = JAVA\_LANG\_OBJECT;

} else {

previousFrame = String.format(FRAME\_FORMAT, env.depth() - 1);

}

c.emit(String.format(FRAME\_INIT\_FORMAT, frame, frame, frame, previousFrame));

return previousFrame;

}

private FrameBlock createFrameBlock(String frame, String previousFrame) {

FrameBlock fBlock = null;

try {

fBlock = new FrameBlock(new PrintStream(new File(String.format(FILE\_PATH\_TO\_BLOCKS, frame))), frame,

previousFrame);

} catch (FileNotFoundException e) {

e.printStackTrace();

}

return fBlock;

}

}

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

public class **ASTAnd** implements ASTNode {

private ASTNode lhs;

private ASTNode rhs;

public ASTAnd(ASTNode lhs, ASTNode rhs) { this.lhs = lhs; this.rhs = rhs; }

public IValue eval(Environment<IValue> e) {

IValue v1 = lhs.eval(e);

if (v1 instanceof VBool) {

IValue v2 = rhs.eval(e);

if (v2 instanceof VBool) {

if (((VBool) v1).getVal() && ((VBool) v2).getVal())

return new VBool(true);

return new VBool(false);

}

}

throw new RuntimeException("illegal arguments to && operator");

}

public void compile(MainBlock c, Environment<Coordinates> e) {

lhs.compile(c, e);

rhs.compile(c, e);

c.emit(“iand”)

}

public IType typecheck(Environment<IType> env) {

IType v1 = lhs.typecheck(env);

if (v1 instanceof TypeBool) {

IType v2 = rhs.typecheck(env);

if (v2 instanceof TypeBool) {

if (((TypeBool) v1).getVal() && ((TypeBool) v2).getVal())

return new TypeBool(true);

return new TypeBool(false);

}

}

throw new RuntimeException("illegal arguments to && operator");

}

}

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

public class **Environment<X>** {

private Map<String, X> links;

private Environment<X> next;

private int depth;

public Environment(Environment<X> next, int depth) {

this.links = new HashMap<>();

this.next = next;

this.depth = depth;

}

public Environment<X> beginScope() {

return new Environment<X>(this, depth + 1);

}

public Environment<X> endScope() { return next; }

public int depth() { return depth; }

public void assoc(String id, X val) {

X value = links.putIfAbsent(id, val);

if (value != null)

throw new RuntimeException("Id already in use: " + id);

}

public X find(String id) {

X value = links.get(id);

if (value != null)

return value;

if (next != null) {

return next.find(id);

}

throw new RuntimeException("Reference not found to id: " + id);

}

}

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

public class **Coordinates** {

private int depth;

private String fieldName;

public Coordinates(int depth, String fieldName) {

this.depth = depth;

this.fieldName = fieldName;

}

public int getDepth() { return depth; }

public String getFieldName() { return fieldName; }

}

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

# JVM instruction

public static final String DIV\_OPERATION = "idiv";

public static final String MULT\_OPERATION = "imul";

public static final String NEG\_OPERATION = "ineg";

public static final String PUSH\_OPERATION = "sipush ";

public static final String ADD\_OPERATION = "iadd";

public static final String SUB\_OPERATION = "isub";

public static final String IFGT = "ifgt %s";

public static final String IFGE = "ifge %s";

public static final String IFLT = "iflt %s";

public static final String IFLE = "ifle %s";

public static final String IFEQ = "ifeq %s";

public static final String IFNE = "ifne %s";

public static final String AND\_OPERATION = "iand";

public static final String OR\_OPERATION = "ior";

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

# Exercices

public class **ASTInc** { //i++

private ASTNode node;

public ASTInc(ASTNode node) {

this.node = node;

}

IValue eval(Environment<IValue> e) throws Exception {

IValue v1 = node.eval(e);

if (v1 instanceof VCell) {

IValue v2 = ((VCell) v1).getVal();

if (v2 instanceof VInt) {

((VCell) v1).set(new VInt(((VInt) v2).getVal() + 1));

return v2;

}

}

throw new Exception("Error types");

}

// Assumindo que a referencia já existe

void compile(CodeBlock c, Environment<Coordinates> e) throws Exception {

node.compile(c, e);

c.emit("dup");

c.emit("getfield ref\_of\_int/v I");

c.emit("swap");

c.emit("dup");

c.emit("getfield ref\_of\_int/v I");

c.emit("sipush 1");

c.emit("iadd");

c.emit("putfield ref\_of\_int/v I");

}

IType typecheck(Environment<IType> e) throws Exception {

IType v1 = node.typecheck(e);

if (v1 instanceof TypeRef) {

IType v2 = ((TypeRef) v1).getVal();

if (v2 instanceof TypeInt) {

((TypeRef) v1).set(new TypeInt(((TypeInt) v2).getVal() + 1));

return v2;

}

}

throw new Exception("Error types");

}

}

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

**def i = ref 0 in**

**while (++i < 100) do**

**println !i**

**end**

**end;;**

new frame\_0

dup

invokespecial frame\_0/<init>()V

dup

aload\_3

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

astore\_3

aload\_3

new ref\_of\_int

dup

invokespecial ref\_of\_int/<init>()V

dup

sipush 0

putfield ref\_of\_int/v I

putfield frame\_0/v0 Lref\_of\_int;

L1:

aload\_3

getfield frame\_0/v0 Lref\_of\_int;

dup

getfield ref\_of\_int/v I

swap

dup

getfield ref\_of\_int/v I

sipush 1

iadd

putfield ref\_of\_int/v I

sipush 100

isub

iflt L3

sipush 0

goto L4

L3:

sipush 1

L4:

ifeq L2

getstatic java/lang/System/out Ljava/io/PrintStream;

aload\_3

getfield frame\_0/v0 Lref\_int;

getfield ref\_of\_int/v I

invokestatic java/lang/String/valueOf(I)Ljava/lang/String;

invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

goto L1

L2:

aload\_3

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

astore\_3

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

**While (3 < 5) { //do this! }**

L1:

ipush 3

ipush 5

isub

iflt L2

ipush 0

goto L3

L2:

//do this!

pop

goto L1

L3:

ifeq

. . .

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

Public class **ListType** implements IType {

private List<IType> l;

private String jvmType;

private Class<?> listType;

public ListType(List<IType> l) {

Class<?> typeInList = null;

for(IType t: l){

if(typeInList == null){

typeInList = t.getClass();

} else {

if(t.getClass() != typeInList) throw new Exception("Error of types in list");

}

}

listType = typeInList;

if(typeInList == Class.TypeInt) jvmType = “intList”;

if(typeInList == Class.TypeBool) jvmType = “boolList”;

if(typeInList == Class.TypeList) jvmType = “listList”;

this.l = l;

}

public List<IType> getList() { return this.l; }

public void set(List<IType> l) { this.l = l; }

public add(IType v) { this.l.add(v); }

public remove(IType v) { this.l.remove(v); }

public String getJvmType() { return this.jvmType; }

}

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

Public class **ASTConcat** implements ASTNode {

private List<ASTNode> lhList;

private List<ASTNode> rhList;

public ASTDef(List<ASTNode> lhList, List<ASTNode> rhList) {

this.lhList = lhList;

this.list = rhList;

}

public IValue eval(Environment<IValue> e) {

IValue v1 = lhList.eval(e);

if(v1 instanceof IList){

IValue v2 = rhList.eval(e);

if(v2 instanceof IList){

return Lists.newArrayList(Iterables.concat((IList)v1, (IList)v2)); //concat

}

}

throw new Exception("Error types");

}

public void compile(MainBlock c, Environment<Coordinates> env) {

lhList.compile(c, e);

rhList.compile(c, e);

c.emit("aload\_3”)

c.emit("gefield frame\_0/v0 Lref\_of\_list”)

c.emit("dup”)

c.emit("getfield ref\_of\_list/v0 L");

c.emit("getfield ref\_of\_list/v0 L");

// instruct to concat

c.emit("putfield ref\_of\_list /v0 L");

}

public IType typecheck(Environment<IType> env) {

lhList.typecheck(env);

return rhList.typecheck(env);

throw new Exception("Error types");

}

}

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

Public class **ASTHead** implements ASTNode { //vai ser

private ASTNode list;

public ASTHead(ASTNode list) {

this.list = list;

}

public IValue eval(Environment<IValue> e) {

IValue v1 = list.eval(e);

if(v1 instanceof IList) {

List auxL = (IList)(((IList) v1).getVal());

int val = auxL[0];

return new VInt(val);

}

}

public void compile(MainBlock c, Environment<Coordinates> e) {

list.compile(c, e);

. . .

}

public IType typecheck(Environment<IType> env) {

IType v1 = list.typecheck(env);

if (v1 instanceof typeList){

List auxL = (IList)(((IList) v1).getVal());

int val = auxL[0];

return new VInt(val);

}

throw new Exception("Error types");

}

}

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

Public class **ASTEmpty** implements ASTNode {

private ASTNode list;

public ASTEmpty(ASTNode list) {

this.list = list;

}

public IValue eval(Environment<IValue> e) {

IValue v1 = list.eval(e);

if(v1 instanceof IList) {

List auxL = (IList)(((IList) v1).getVal());

boolean val = auxL.isEmpty();

return new VBool(val);

}

}

public void compile(MainBlock c, Environment<Coordinates> env) {

list.compile(c, e);

. . .

}

public IType typecheck(Environment<IType> env) {

IType v1 = list.typecheck(env);

if (v1 instanceof typeList){

List auxL = (IList)(((IList) v1).getVal());

boolean val = auxL.isEmpty();

return new VBool(val); } throw new Exception("Error types") } }

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

**{let x = new 2; {**

**let y = !x + 2;**

**if(!x \* y >= 0)**

**{ x := !x }**

**{ }**

**}**

**};;**

new frame\_0

dup

invokespecial frame\_0/<init>()V

dup

aload\_3

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

astore\_3

aload\_3

new ref\_of\_int

dup

invokespecial ref\_of\_int/<init>()V

dup

sipush 2

putfield ref\_of\_int/v I;

putfield frame\_0/v0 Lref\_of\_int;

new frame\_1

dup

invokespecial frame\_1/<init>()V

dup

aload\_3

putfield frame\_1/sl Lframe\_0;

astore\_3

aload\_3

dup

getfield frame\_1/sl Lframe\_0;

getfield frame\_0/v0 Lref\_of\_int;

getfield Lref\_of\_int/v I

sipush 2

iadd

putfield frame\_1/v0 I;

aload\_3

getfield frame\_1/sl Lframe\_0;

getfield frame\_0/v0 Lref\_of\_int;

getfield ref\_of\_int/v I

aload\_3

getfield frame\_1/v0 I;

imult

ifge L1

goto L2

L1:

aload\_3

dup

getfield frame\_1/sl Lframe\_0;

getfield frame\_0/v0 Lref\_of\_int;

aload\_3

getfield frame\_1/sl Lframe\_0;

getfield frame\_0/v0 Lref\_of\_int;

getfield ref\_of\_int/v I

putfield ref\_of\_int/v I

goto L3

L2:

// nothing

L3:

aload\_3

getfield frame\_1/sl Lframe\_0;

astore\_3

aload\_3

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

astore\_3