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

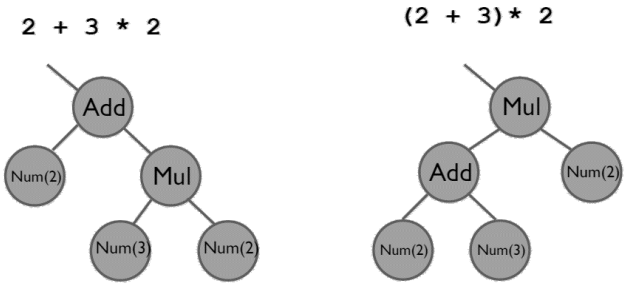
Concreta – 2+1

- 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

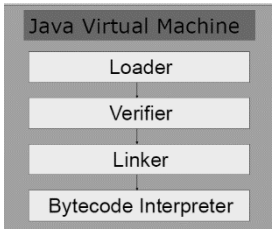
Alg. **eval(*E*)** computa a notação de de uma qualquer *CALC expression*
O interpretador vai interpretar qual a denotação da expressão(add, mult, sub, ...).

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

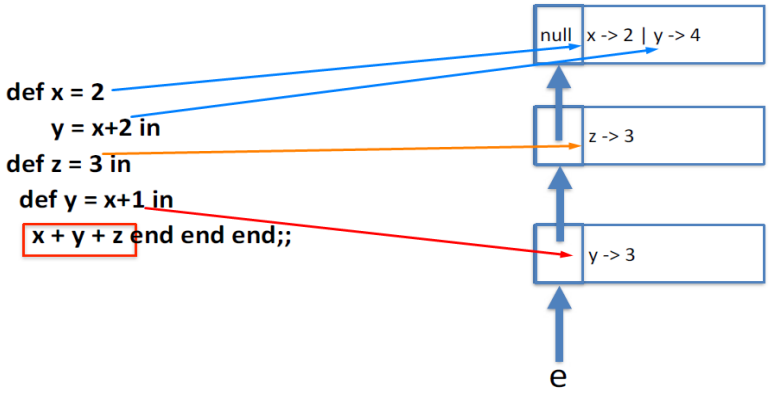
Stack Machine: todas as instruções consomem seus argumentos do topo da pilha e deixam um resultado no topo da pilha

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

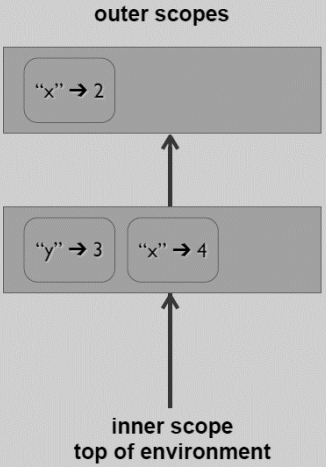
O ambiente de compilação D mapeia cada nome livre do programa a ser compilado em suas coordenadas:
D(x) = (d, s) onde . . .

- **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)

```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x");        // returns 2
val = env.Find("y");        // raises "Not declared"
env = env.BeginScope();
env.Assoc("y", 3);
env.Assoc("x", 4);
val = env.Find("y");        // returns 3
val = env.Find("x");        // returns 4
```



CALC compiler

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

comp(**num**(n)) ≜ < **ldc.i4 n** >

comp(**add**(E',E'')) ≜ comp(E') @ comp(E'') @ < **add** >

comp(**mul**(E',E'')) ≜ comp(E') @ comp(E'') @ < **mul** >

comp(**sub**(E',E'')) ≜ comp(E') @ comp(E'') @ < **sub** >

comp(**div**(E',E'')) ≜ comp(E') @ comp(E'') @ < **div** >

first" (5) machine instructions of the JVM:
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
```

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.

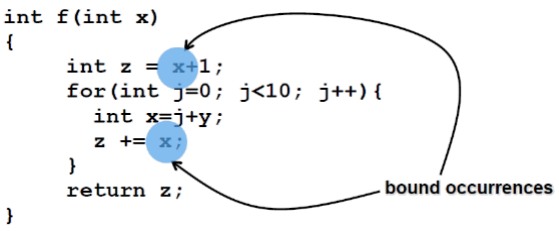
Binding and Scope

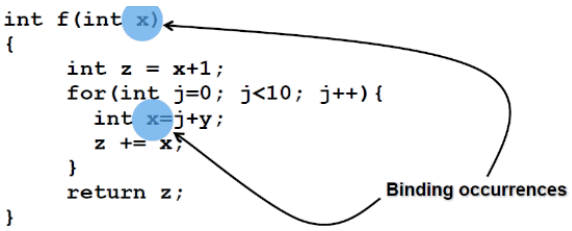
Binding → associação entre um intentificador e um valor

- 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

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```

free occurrences

Language CALC

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

```
def Id = Exp1 in Exp2 end
```

Um programa CALCI é uma expressão fechada de CALCI

```
def x=2 in def y=x+2 in (x+y) end end
```

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

The JVM code generated for an expression

def x1 = E1 ... xn = En **in** E **end**

1. creates and pushes a new stack frame (heap allocated) to store the value of the n identifiers x1 ... xn

2. initialises the slot for each xi with the value of Ei

3. pops off of the frame

The generated code for an id does not need to dynamically search up in the stack, it knows its coordinates (from D(id))

```
------------------------------------------- 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
```

```
------------------------------------------- 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
```

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 }

Fator -> (abstract syntax)

| num
| id
| EE + EE | EE - EE
| EE * EE | EE / EE
| -EE | (EE)
| { (let id = EE ;)+ EE }

Project code

```
public interface ASTNode {
    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 Environment {
    private Map<String, Integer> links;
    private Environment prevEnv;

    public Environment(Environment prevEnv) {
        this.links = new HashMap<>();
        this.prevEnv = prevEnv;
    }
    public Environment beginScope() {
        return new Environment(this);
    }
    public Environment endScope() {
        return prevEnv;
    }
    public void assoc(String id, int val) {
        Integer value = links.putIfAbsent(id, val);
        if (value != null)
            throw new RuntimeException("Id already in use: " + id);
    }

    public int find(String id) {
        Integer value = links.get(id);
        if (value != null)
            return value.intValue();
        if (prevEnv != null) {
            return prevEnv.find(id);
```

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

```
public class ASTId implements ASTNode {
    String id;

    public ASTId(String id) {
        this.id = id;
    }
    public int eval(Environment e) {
        return e.find(id);
    }
}
```

```
public class ASTDef implements ASTNode {
    Map<String, ASTNode> init;
    ASTNode body;

    public ASTDef(Map<String, ASTNode> init, ASTNode body) {
        this.init = init;
        this.body = body;
    }
    public int eval(Environment env) {
        // def x1 = E1 … xn = En in Body end
        env = env.beginScope();
        int 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;
    }
}
```

```
class ASTFor implements ASTNode {
    ASTNode begin;
    ASTNode end;
    String id;
    ASTNode sum;

    public ASTFor(String id, ASTNode begin, ASTNode end, ASTNode sum) {
        this.id = id;
        this.sum = sum;
        this.begin = begin;
        this.end = end;
    }
    @Override
    public int eval(Environment e) {
        int v1 = begin.eval(e);
        int v2 = end.eval(e);

        int sum = 0;
        for (int x = v1; x <= v2; x++) {
            e.beginScope();
            e.assoc(id, x);
            sum += this.sum.eval(e);
            e.endScope();
        }
        return sum;
    }
}
```

```
public class ASTPlus implements ASTNode {
        ASTNode lhs, rhs;

        public ASTPlus(ASTNode l, ASTNode r) {
                lhs = l;
                rhs = r;
        }
        public int eval() {
                int v1 = lhs.eval();
                int v2 = rhs.eval();
                return v1 + v2;
        }
        @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);
            }
        }
    }
}
-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------
SKIP:
{
  " "
| "\t"
| "\r"
}
TOKEN:
{
  < Id: ["a"-"z","A"-"Z"] ( ["a"-"z","A"-"Z","0"-"9"] )* >
  |
  < Num: (["0"-"9"]) + >
  |
  < PLUS : "+" >
  |
  < MINUS : "-">
  |
  < MULT : "*">
  |
  < DIV : "/">
  |
  < LPAR : "(" >
  |
  < RPAR : ")" >
  |
  < LET : "let" (" ")*>
  |
  < LBRA : "{" >
  |
  < RBRA : "}" >
  |
  < EQUAL : "=">
  |
  < SEMICOLON : ";">
  |
  < 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 ASTAdd(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 t;
  Map<String,ASTNode> map;
}
{
   (
     x= <Num> { t = new ASTNum(Integer.parseInt(x.image)); }
   | x = <Id> { t = new ASTId(x.image);}
   | <LPAR> t=Exp() <RPAR>
   | <MINUS> t = Fact() { t=new ASTNeg(t); }
   | <LBRA> {map = new HashMap<>();}
            (<LET> x=<Id> <EQUAL> t=Exp() <SEMICOLON> (<EL>)? { map.put(x.image,t);})+ t = Exp()
     <RBRA> { t = new ASTDef(map,t); }
    ) { return t; }
}
```

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)))))