PADRÃO DE PROJETO VISITOR

Baseado no Capítulo 5 de Programming Language Processors in Java, de Watt & Brown

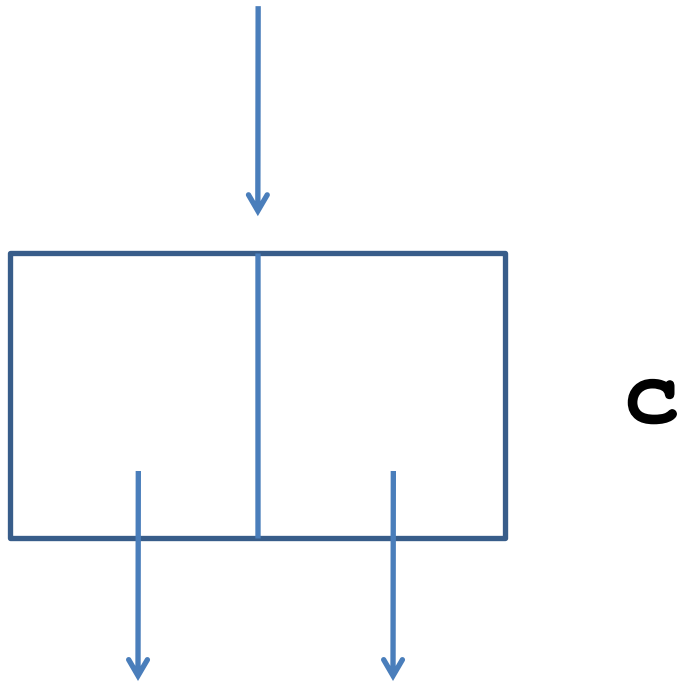
$$
\begin{aligned}
P &\rightarrow DC \\
D &\rightarrow \#ID \mid \varepsilon \\
C &\rightarrow \$I=EC \mid \varepsilon \\
E &\rightarrow T+E \mid T \\
T &\rightarrow F*T \mid F \\
F &\rightarrow (E) \mid I \\
I &\rightarrow A \mid B \mid \ldots \mid Z
\end{aligned}
$$

```
#A
#A#B
$A=B
$A=B+C*D
#A#B$A=B
#A#B#C#D$A=B$B=C+D
```

- Todos os identificadores (I) utilizados nos comandos (C), tanto no lado esquerdo quanto do lado direito da atribuição, devem ser declarados antes do uso;
- Cada identificador só pode ser declarado uma única vez;

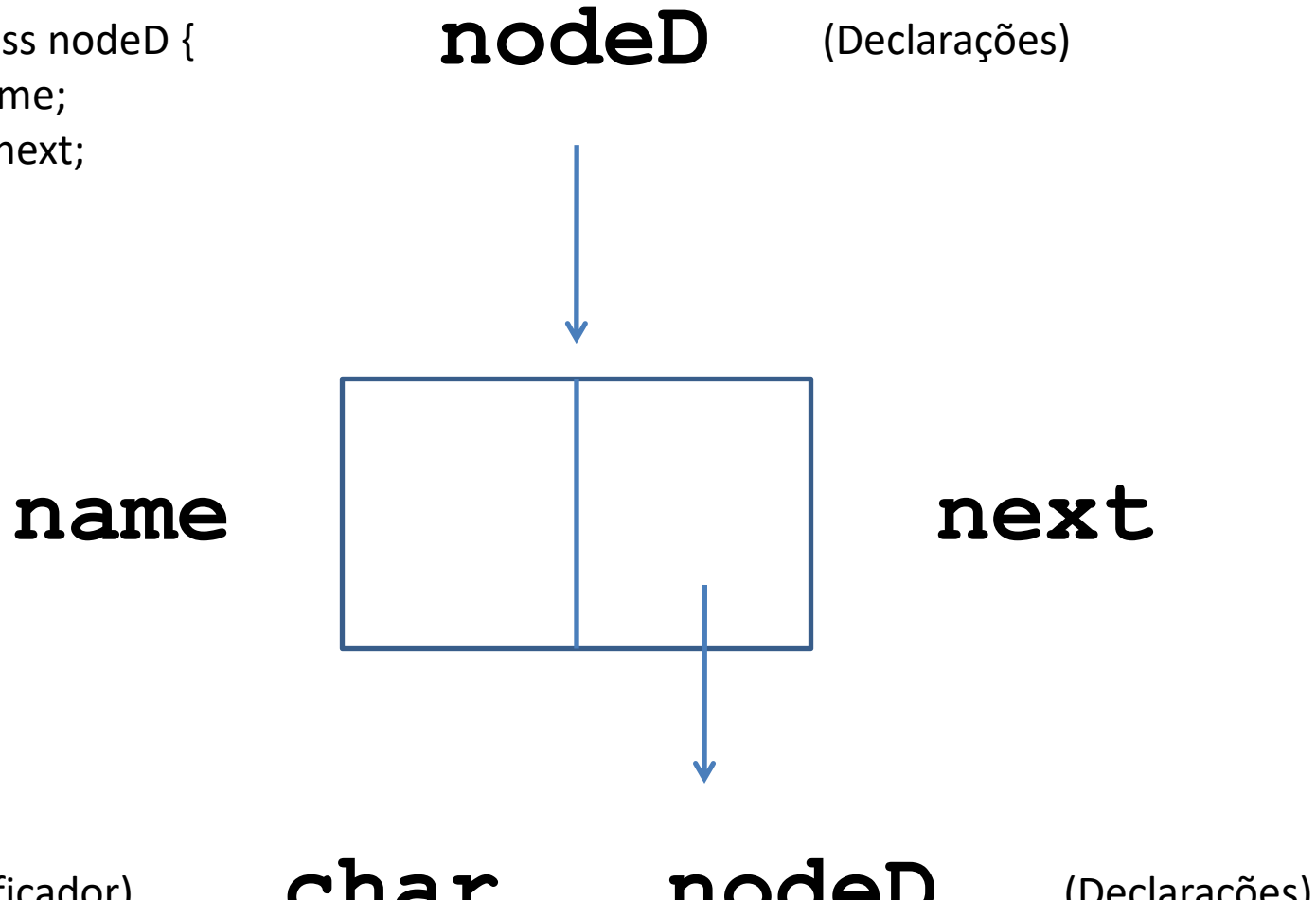
```
public class nodeP {
   nodeD d;
   nodeC c;
}
```

nodeP (Programa)

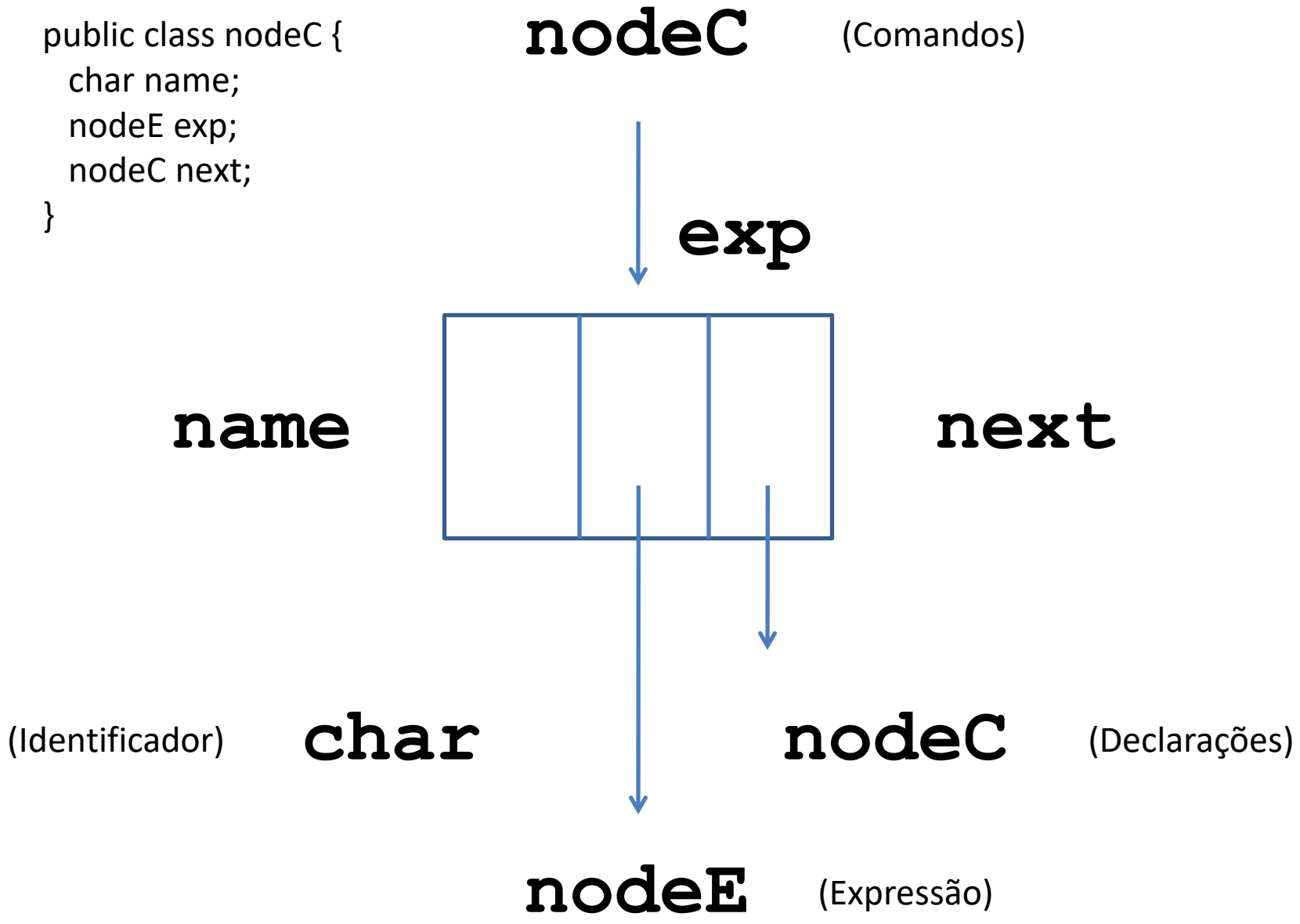


(Declarações) **nodeD** **nodeC** (Comandos)

```
public class nodeD {
  char name;
  nodeD next;
}
```

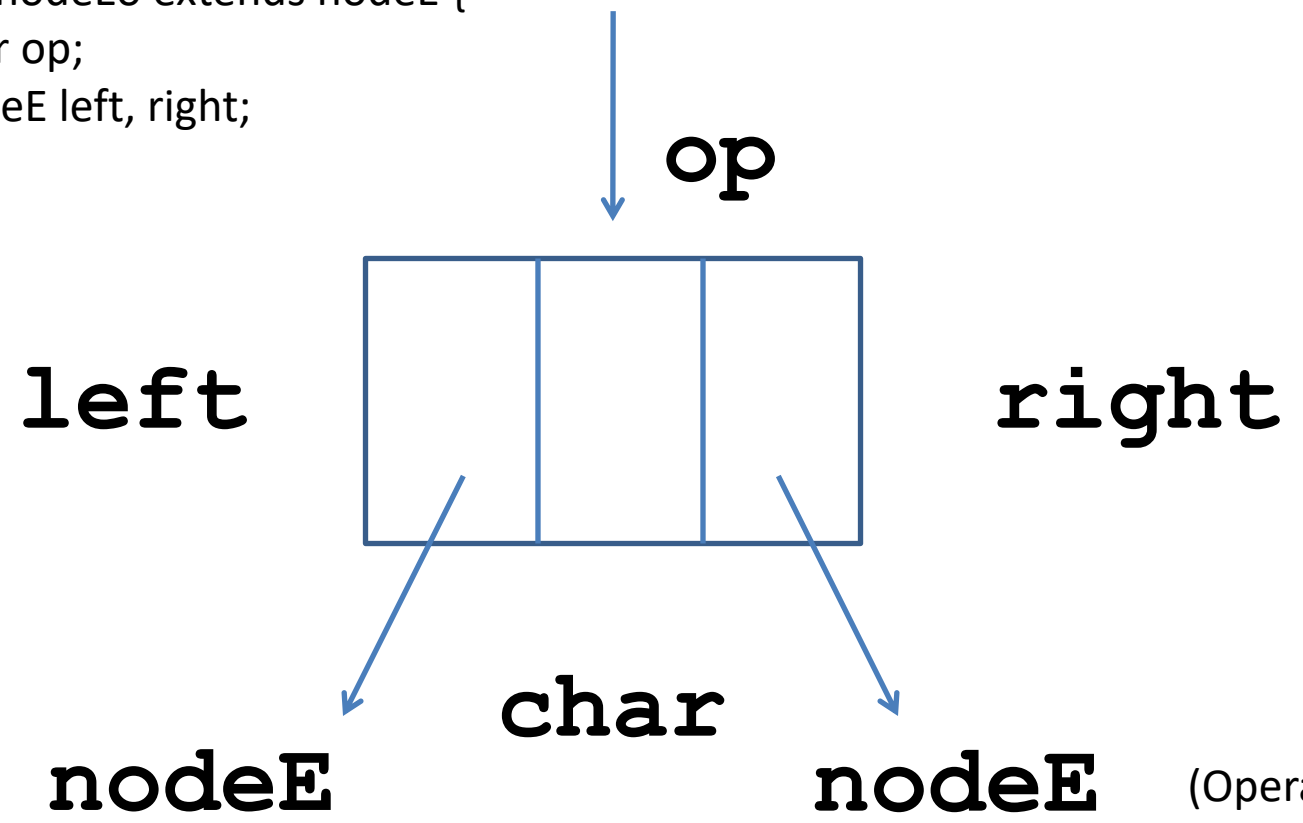


```
public class nodeC {
  char name;
  nodeE exp;
  nodeC next;
}
```



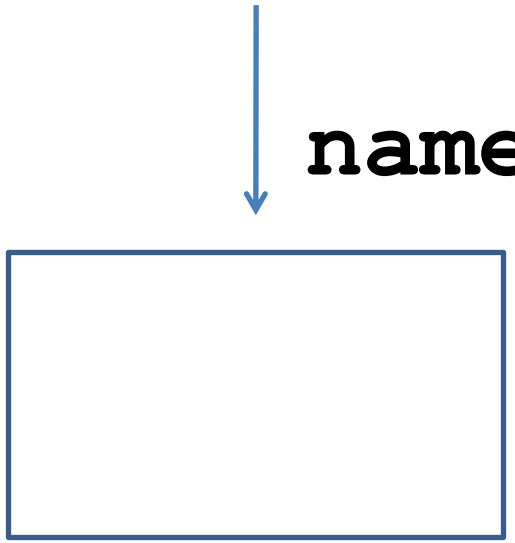
nodeEo (Expressão com operador)

```
public class nodeEo extends nodeE {
   public char op;
   public nodeE left, right;
}
```



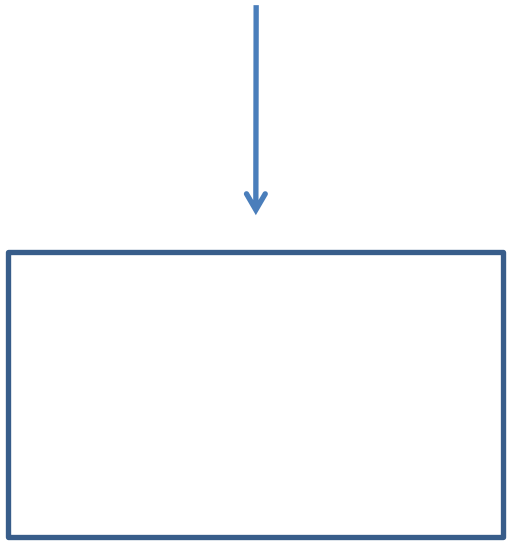
(Operando 1) (Operando 2)

nodeEn (Expressão com operando)



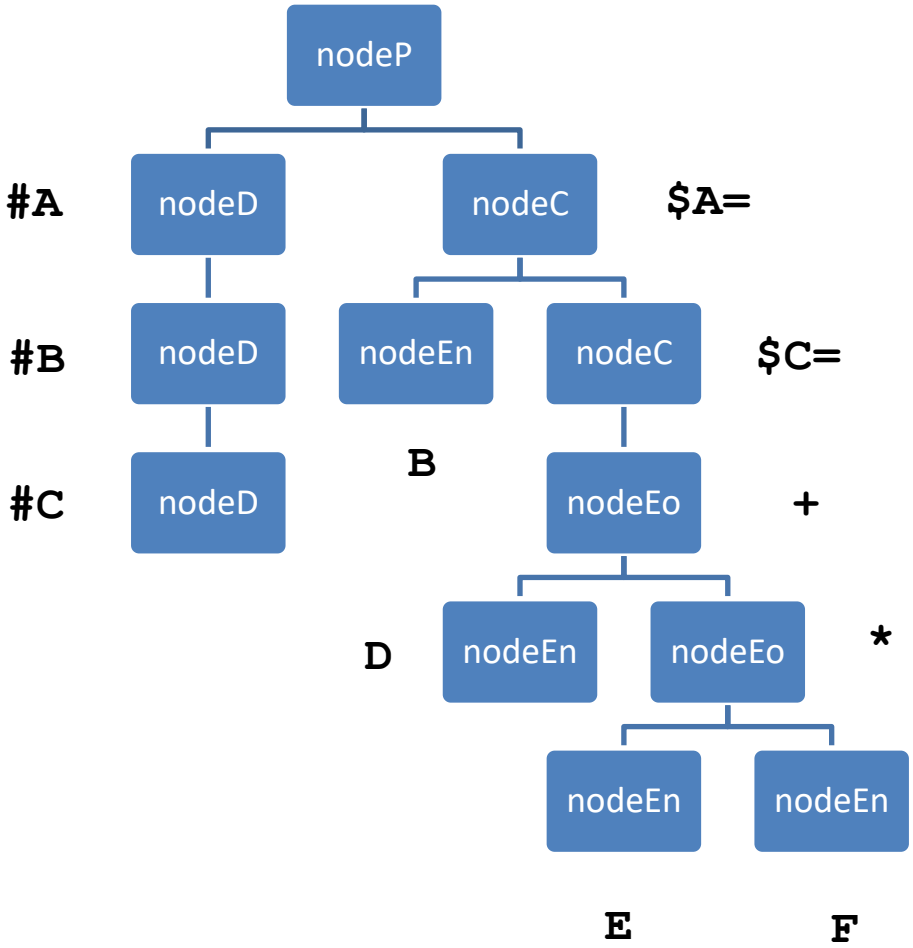

```
public class nodeEn extends nodeE {
   public char name;
}
```

nodeE



```
public abstract class nodeE {
}
```

#A#B#C$A=B$C=D+E*F


nodeP
#A
nodeD
nodeC
$A=
#B
nodeD
nodeEn
nodeC
$C=
B
#C
nodeD
nodeEo
+
D
nodeEn
nodeEo
*
nodeEn
nodeEn
E
F

1. Montar o analisador sintático (classe Parser);
2. Montar a árvore de sintaxe (classes nodeP, nodeD, nodeC, nodeEo, nodeEn);
3. Implementar a interface Visitor;
4. Imprimir a árvore (classe Printer);
5. Fazer a análise de contexto (classe Checker);
6. Fazer a geração de código (classe Coder).

```
public class Parser {

nodeP parseP() {...}
nodeD parseD() {...}
nodeC parseC() {...}
nodeE parseE() {...}
nodeE parseT() {...}
nodeE parseF() {...}
char parseI()  {...}

public nodeP parse(String arg){
   nodeP p;
   System.out.println();
   System.out.println ("---> Iniciando analise sintatica");
init(arg);
p = parseP();
return p;
}

}
```

```
nodeP parseP(){
   nodeP p = new nodeP();
   p.d=parseD();
   p.c=parseC();
   if (i!=n) error();
   return p;
}

nodeD parseD(){
   nodeD first, last, d;
   first = null;
   last = null;
   while (token=='#') {
      takeIt();
      d = new nodeD();
      d.name = parseI();
      d.next = null;
      if (first==null) first=d;
         else last.next=d;
      last=d;
      }
   return first;
}
```

```
nodeC parseC(){
   nodeC first, last, c;
   first = null;
   last = null;
   while (token=='$') {
      takeIt();
      c = new nodeC();
      c.name = parseI();
      take ('=');
      c.exp = parseE();
      c.next = null;
      if (first==null) first=c;
         else last.next=c;
      last=c;
   }
   return first;
}
```

```
nodeE parseE(){
      nodeE last;
      last = parseT();
      while (token=='+'){
         takeIt();
         nodeEo current = new nodeEo();
         current.init ('+',last,parseT());
         last = current;
         }
      return last;
   }

nodeE parseT(){
      nodeE last;
      last = parseF();
      while (token=='*'){
         takeIt();
         nodeEo current = new nodeEo();
         current.init ('*',last,parseF());
         last = current;
         }
      return last;
   }
```

```
 nodeE parseF(){
      nodeE e;
      if (token=='('){
            takeIt();
            e=parseE();
            take(')');
            }
         else {
              nodeEn ex = new nodeEn();
              ex.init (parseI());
              e=ex;
              }
      return e;
   }

char parseI(){
char d=' ';
if ((token>='A')&&(token<='Z')) {
   d = token;
   takeIt();
   }
else error();
return d;
}
```

VISITOR

- Permite desvincular a descrição de uma estrutura de dados da especificação das ações que devem ser executadas sobre a mesma;
- Através da INTERFACE, torna-se obrigatória a implementação da ação a ser executada para todos os nós que compõem a estrutura de dados em questão.

VISITOR

Como implementar?

VISITOR

1. Criar uma INTERFACE com métodos abstratos de visitação para todos os nós da árvore:
   a) Todos os nós da árvore deverão estar associados ao seu correspondente método de visitação;
   b) O nome do método deverá fazer referência ao tipo de nó visitado;
   c) O método deverá receber como argumento um objeto correspondente ao tipo de nó visitado.

VISITOR

```
public class Compiler{

public static void main(String args[]){
    nodeP p;
    Parser parser = new Parser();
    Printer printer = new Printer();
    Checker checker = new Checker();
    Coder coder = new Coder();
    p = parser.parse(args[0]);
    printer.print(p);
    checker.check(p);
    coder.code(p);
    }

}
```

VISITOR

```
public interface Visitor {

   public void visitP (nodeP p);
   public void visitD (nodeD d);
   public void visitC (nodeC c);
   public void visitEn (nodeEn e);
   public void visitEo (nodeEo e);

}
```

VISITOR

2. Acrescentar, em todas classes que representam os nós da árvore, um método de visitação:
   a) Todos os métodos deverão ter o mesmo nome;
   b) Todos os métodos deverão receber como argumento um objeto da classe Visitor;
   c) O corpo desses métodos será composto por um único comando, responsável pela chamada no método de visitação específico, no argumento, conforme o tipo de nó em questão.

VISITOR

```
public class nodeP {

   nodeD d;
   nodeC c;

   public void visit  (Visitor v){
      v.visitP(this);
      }

}
```

VISITOR

```
public class nodeD {

   char name;
   nodeD next;

   public void visit  (Visitor v) {
      v.visitD(this);
      }

}
```

VISITOR

```
public class nodeC {

   char name;
   nodeE exp;
   nodeC next;

   public void visit (Visitor v) {
      v.visitC(this);
      }

}
```

VISITOR

```
public abstract class nodeE {

   public abstract void visit (Visitor v);

}
```

VISITOR

```
public class nodeEo extends nodeE {

   public char op;
   public nodeE left, right;

   public void init (char o, nodeE l, nodeE r) {
      op=o;
      left=l;
      right=r;
      }

   public void visit (Visitor v) {
      v.visitEo (this);
      }

}
```

VISITOR

```
public class nodeEn extends nodeE {

   public char name;

   public void init (char n) {
      name=n;
      }

   public void visit (Visitor v) {
      v.visitEn (this);
      }

}
```

VISITOR

3. Para cada tipo de operação diferente que se deseje executar sobre a árvore (impressão, análise de contexto, geração de código etc), criar uma nova classe que implementa a interface VISITOR:
   a) Todas as classes assim especificadas deverão conter implementações para todos os métodos relacionados na interface;
   b) Cada método é responsável por executar a operação requerida sobre o particular tipo de nó a quel ele se refere;
   c) Quando a ação de um método sobre um nó depender da execução de ação de natureza semelhante sobre outros nós a ele vinculados, o método em questão deverá invocar a aplicação dos correspondentes métodos de visitação para os respectivos nós;
   d) A referência THIS deverá ser passada como argumento em toda chamada de métodos de visitação.

VISITOR

```
public class Printer implements Visitor {

   public void visitP (nodeP p) {...}
   public void visitD (nodeD d) {...}
   public void visitC (nodeC c) {...}
   public void visitE (nodeE e) {...}
   public void visitEo (nodeEo e) {...}
   public void visitEn (nodeEn e) {...}

   public void print (nodeP p) {
      System.out.println ("---> Iniciando impressao da arvore");
      p.visit (this);
   }

}
```

VISITOR

```
public void visitP (nodeP p) {

     if (p!=null) {
        if (p.d!=null) p.d.visit(this);
        if (p.c!=null) p.c.visit(this);
        }

  }
```

VISITOR

```
public void visitD (nodeD d) {
      if (d!=null) {
         System.out.println ("#"+d.name);
         if (d.next!=null) {
            i++;
            indent();
            d.next.visit (this);
            i--;
            }
         }
   }
```

VISITOR

```
public void visitC (nodeC c) {
      if (c!=null) {
         System.out.println ("$"+c.name);
         c.exp.visit(this);
         if (c.next!=null) {
            i++;
            indent();
            c.next.visit (this);
            i--;
            }
         }
   }
```

VISITOR

```
public void visitE (nodeE e) {
      if (e!=null) {
         if (e instanceof nodeEo) ((nodeEo)
e).visit (this);
            else ((nodeEn) e).visit (this);
         }
   }
```

VISITOR

```
public void visitEo (nodeEo e) {
      i++;
      indent();
      System.out.println (e.op);
      e.left.visit (this);
      e.right.visit (this);
      i--;
   }
```

VISITOR

```
public void visitEn (nodeEn e) {
      i++;
      indent();
      System.out.println (e.name);
      i--;
   }
```

VISITOR

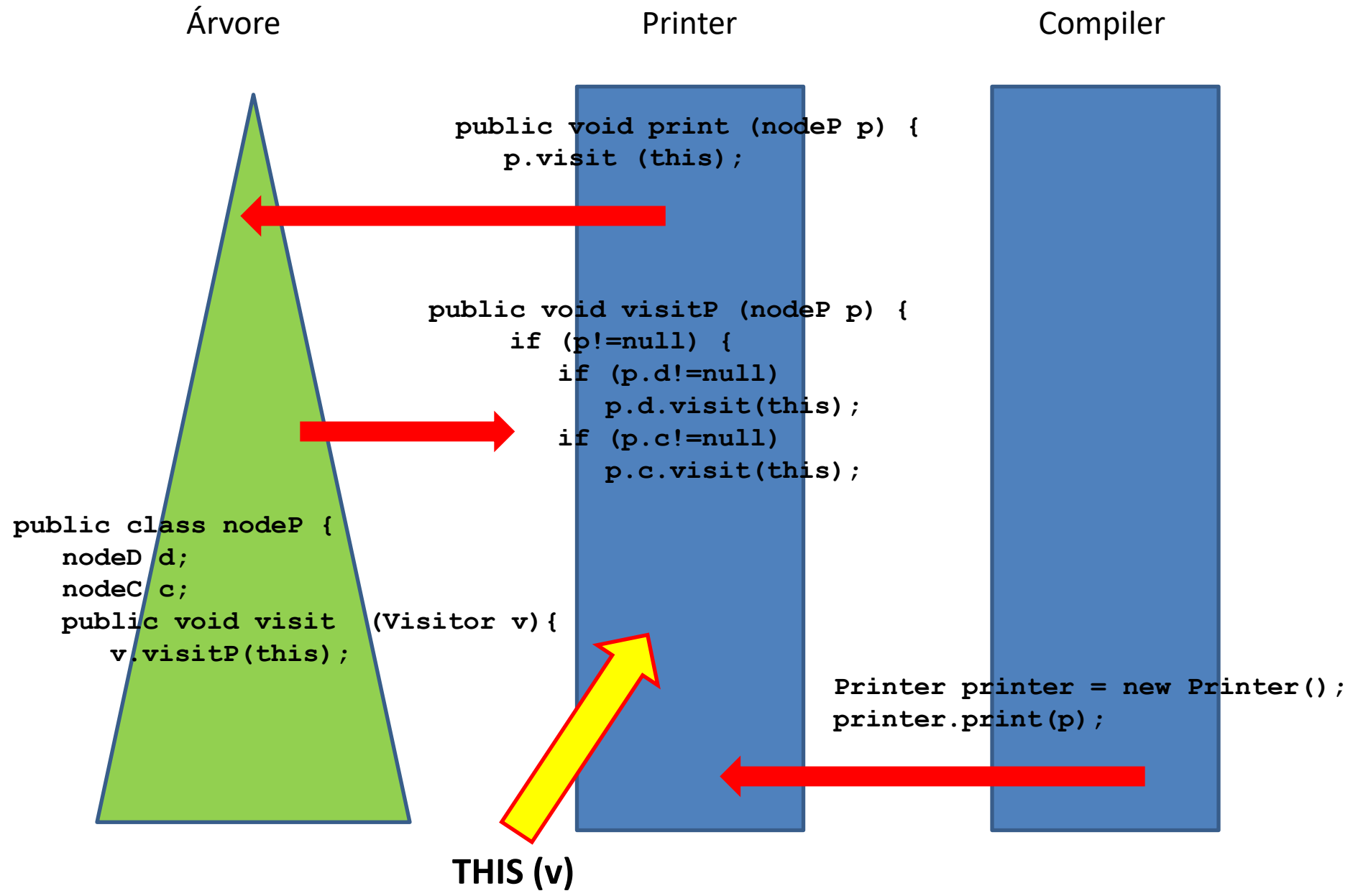
4. Para iniciar, o programa principal deverá:
    a) Instanciar cada uma das classes que implementam a interface VISITOR;
    b) Invocar o método inicial de cada classe.

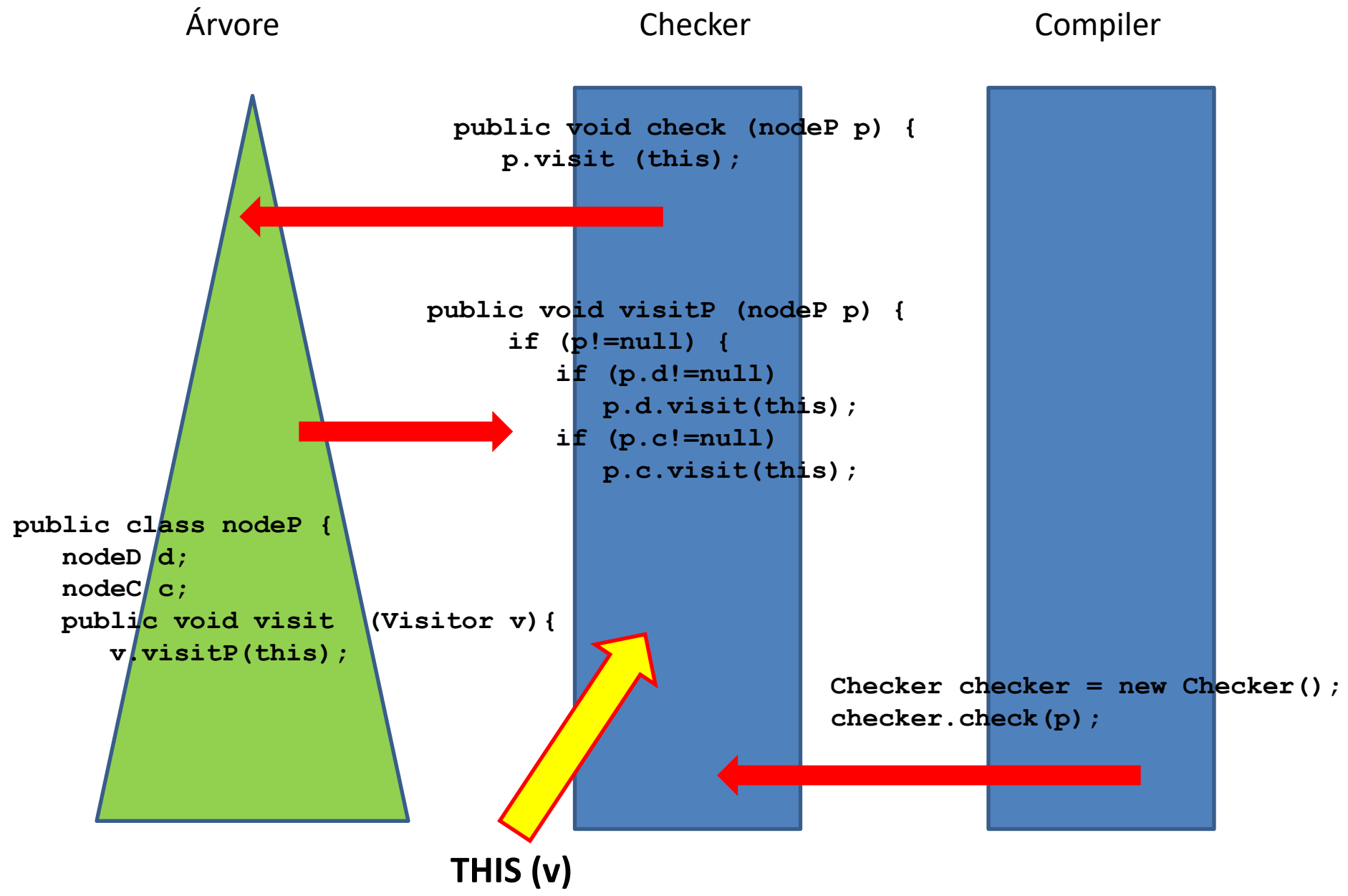
VISITOR

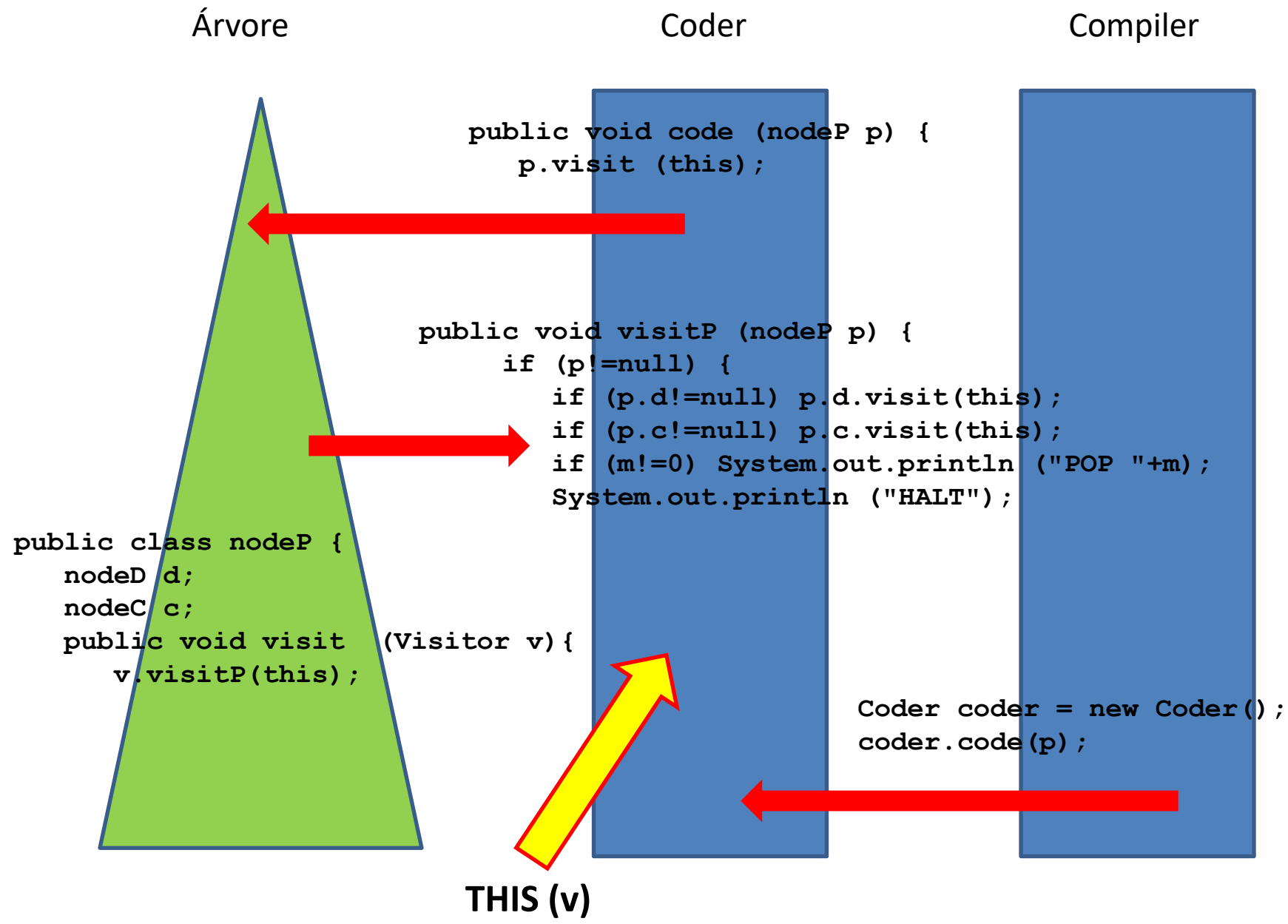
```
public class Compiler{

public static void main(String args[]){
    nodeP p;
    Parser parser = new Parser();
    Printer printer = new Printer();
    Checker checker = new Checker();
    Coder coder = new Coder();
    p = parser.parse(args[0]);
    printer.print(p);
    checker.check(p);
    coder.code(p);
    }

}
```

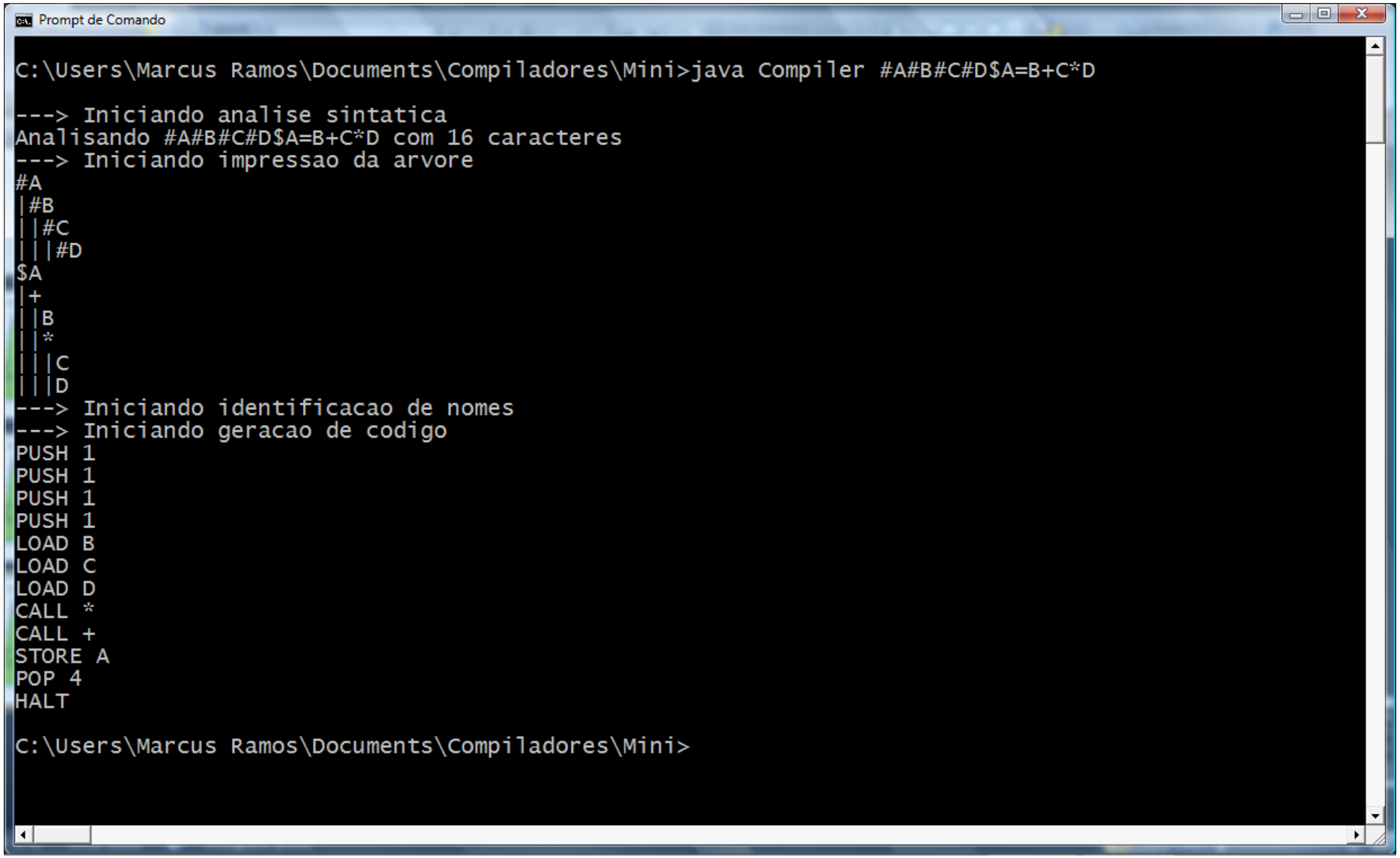
VISITOR

Como funciona?


Árvore
Printer
Compiler
public void print (nodeP p) {
p.visit (this);
public void visitP (nodeP p) {
if (p!=null) {
if (p.d!=null)
p.d.visit(this);
if (p.c!=null)
p.c.visit(this);
public class nodeP {
nodeD d;
nodeC c;
public void visit (Visitor v){
v.visitP(this);
Printer printer = new Printer();
printer.print(p);
THIS (v)


Árvore
Checker
Compiler
public void check (nodeP p) {
p.visit (this);
public void visitP (nodeP p) {
if (p!=null) {
if (p.d!=null)
p.d.visit(this);
if (p.c!=null)
p.c.visit(this);
public class nodeP {
nodeD d;
nodeC c;
public void visit (Visitor v){
v.visitP(this);
Checker checker = new Checker();
checker.check(p);
THIS (v)


Árvore
Coder
Compiler
public void code (nodeP p) {
p.visit (this);
public void visitP (nodeP p) {
if (p!=null) {
if (p.d!=null) p.d.visit(this);
if (p.c!=null) p.c.visit(this);
if (m!=0) System.out.println ("POP "+m);
System.out.println ("HALT");
public class nodeP {
nodeD d;
nodeC c;
public void visit (Visitor v){
v.visitP(this);
Coder coder = new Coder();
coder.code(p);
THIS (v)


Prompt de Comando
C:\Users\Marcus Ramos\Documents\Compiladores\Mini>java Compiler #A#B#C#D$A=B+C*D
---> Iniciando analise sintatica
Analisando #A#B#C#D$A=B+C*D com 16 caracteres
---> Iniciando impressao da arvore
#A
|#B
||#C
|||#D
$A
|+
||B
||*
|||C
|||D
---> Iniciando identificacao de nomes
---> Iniciando geracao de codigo
PUSH 1
PUSH 1
PUSH 1
PUSH 1
LOAD B
LOAD C
LOAD D
CALL *
CALL +
STORE A
POP 4
HALT
C:\Users\Marcus Ramos\Documents\Compiladores\Mini>