# *Geração de árvores de sintaxe abstrata*

# Agenda

- Ações semânticas do *parser*
- Porquê Árvores de Sintaxe Abstracta (AST)?
- Definição das classes da AST
- Criação de ASTs pelo *parser*
- Interpretação da AST
  - Métodos dedicados nas classes da AST
  - *Instanceof* e *cast*
  - *Visitors*
- Linguagem MiniJava
  - Definição das classes da AST para o MiniJava

# Segunda etapa da compilação

- Consome os tokens reconhecidos pelo analisador léxico
- Reconhece frases construídas na linguagem fonte
- Gera a árvore de sintaxe abstrata que representa as frases reconhecidas

# Ações semânticas

- O analisador sintático não têm apenas o objetivo de verificar se as frases estão corretas ou não
- Durante o reconhecimento sintático o analisador pode realizar ações semânticas

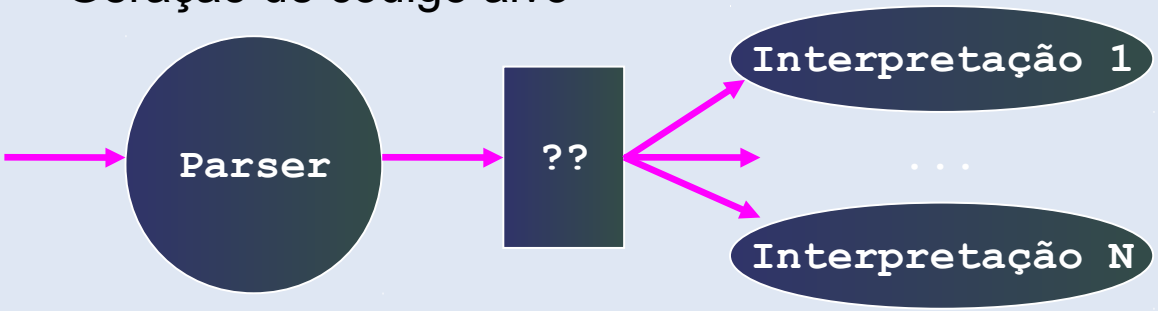
- Exemplo: Quando é reconhecida a sintaxe da atribuição é realizada a ação semântica:
  - Avaliar a expressão
  - Associar o valor da expressão à variável com o nome indicado

```
...
void Assign() : { Token id; int value; } {
  id=<ID> ":=" value=Exp()
  { vars.put(id.image,new Integer(value)); }
}
...
```

# Modularidade dos compiladores

- Pode-se escrever compiladores que geram todo o código alvo apenas nas ações semânticas do analisador sintático (*parser*)
  - Restringe o compilador a realizar a análise do programa pela ordem exata do reconhecimento sintático (*parsing*)
  - Difícil de ler e de manter

- Para melhorar a modularidade, separar:
  - Análise sintática (*parsing*)
  - Análise semântica:
    - Verificação das variáveis declaradas
    - Verificação de tipos (type-checking)
    - ...
  - Geração de código alvo



# Árvore de *parser*

- No limite, uma árvore produzida pelo *parser* tem:
  - Um nó folha para cada símbolo terminal (*token*)
  - Um nó interno para cada símbolo não terminal que resulta da aplicação de uma regra da gramática
- Esta árvore é designada por:
  - Árvore de *parser*
  - Árvore de sintaxe concreta (CST)
- Não é a representação conveniente
  - Tem *tokens* de pontuação (informação redundante)
  - Depende fortemente da gramática (incluindo símbolos não terminais auxiliares)

# Árvore de sintaxe abstrata (AST)

- Representa as frases escritas na gramática original
  - Sem as regras auxiliares
- Sem qualquer processamento semântico
- Os compiladores antigos (anos 70) não usavam esta técnica, por não haver memória suficiente para manter esta árvore
  - Linguagens projetadas nesta época (Pascal, C, …) obrigam a ter declarações *forward* para evitar mais uma passagem

# Definição da AST

Exp→ Exp + Exp

Exp→ Exp * Exp

Exp→ id

Exp→ num

Exp→ ( Exp )

**Hierarquia de classes da AST**

**Árvore de objetos**

**2 * (4 + 6)**

# Definição da AST

Exp→ Exp + Exp

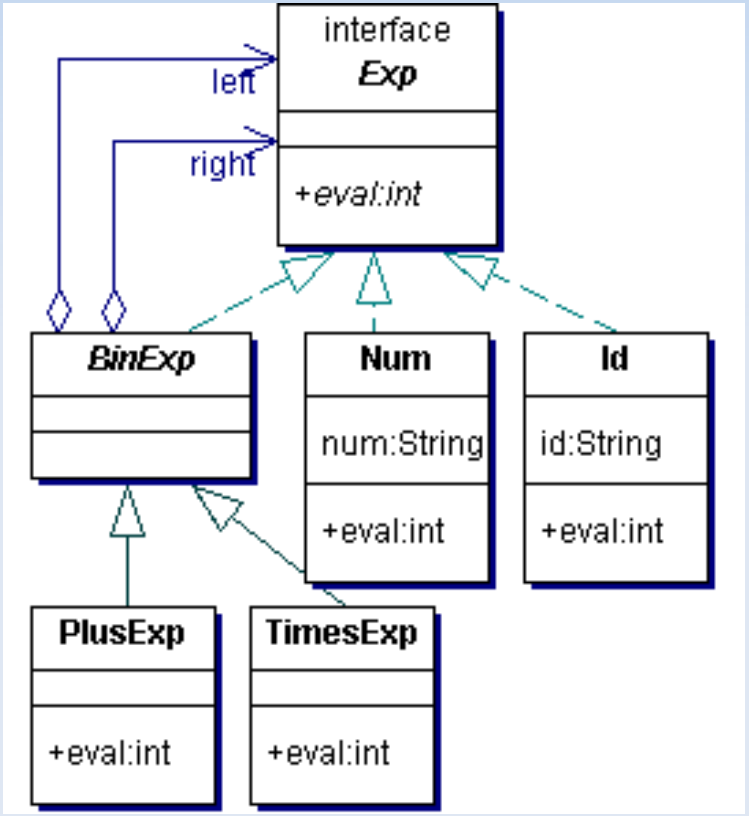
Exp→ Exp * Exp

Exp→ id

Exp→ num

Exp→ ( Exp )

**Hierarquia de classes da AST**




**Árvore de objetos**

2 * (4 + 6)

# Definição da AST

Exp→ Exp + Exp

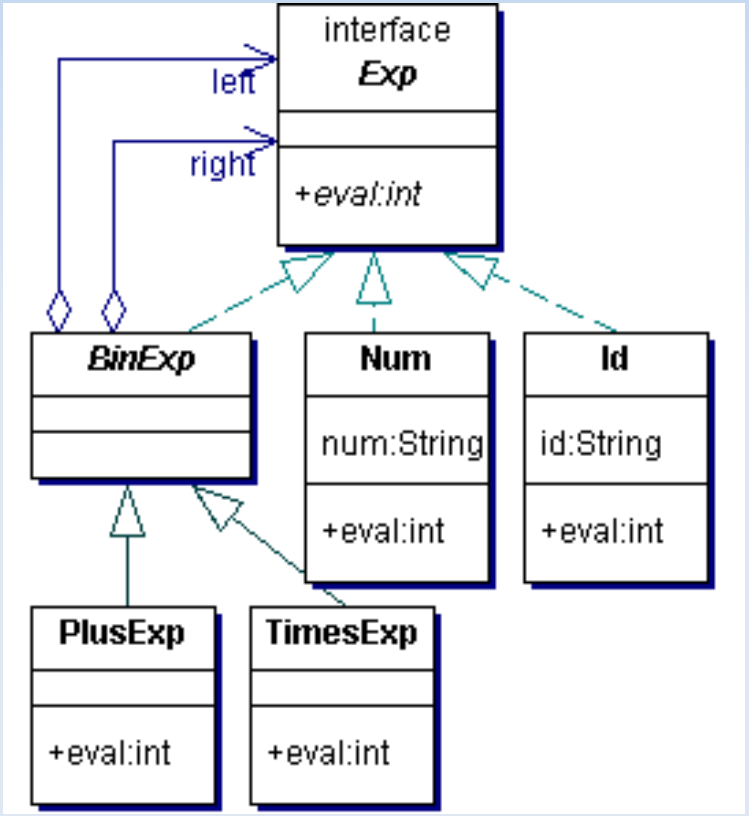
Exp→ Exp * Exp

Exp→ id

Exp→ num

Exp→ ( Exp )

**Hierarquia de classes da AST**




**Árvore de objetos**

2 * (4 + 6)

# Definição das classes da AST em Java

```
public interface Exp {
   public int eval();
}

public abstract class BinExp implements Exp {
   public Exp left, right;
   protected BinExp(Exp l, Exp r) { left=l; right=r; }
}
```

# Definição das classes da AST em Java

```
public interface Exp {
   public int eval();
}

public abstract class BinExp implements Exp {
   public Exp left, right;
   protected BinExp(Exp l, Exp r) { left=l; right=r; }
}
```

```
public class PlusExp extends BinExp {
   public PlusExp(Exp le, Exp re)
    { super(le,re); }
   public int eval()
    { return left.eval()+right.eval(); }
}

public class TimesExp extends BinExp {
   public TimesExp(Exp le, Exp re)
    { super(le,re); }
   public int eval()
    { return left.eval()*right.eval(); }
}
```

# Definição das classes da AST em Java

```
public interface Exp {
   public int eval();
}

public abstract class BinExp implements Exp {
   public Exp left, right;
   protected BinExp(Exp l, Exp r) { left=l; right=r; }
}
```

```
public class PlusExp extends BinExp {
   public PlusExp(Exp le, Exp re)
    { super(le,re); }
   public int eval()
    { return left.eval()+right.eval(); }
}

public class TimesExp extends BinExp {
   public TimesExp(Exp le, Exp re)
    { super(le,re); }
   public int eval()
    { return left.eval()*right.eval(); }
}
```

```
public class Num implements Exp {
   public String num;
   public Num(String image)
    { num=image; }
   public int eval()
    { return Integer.parseInt(num);}
}
public class Id implements Exp {
   public String id;
   public Id(String image)
    { id=image; }
   public int eval()
    { return ... ; }
}
```

# Criação da AST pelo Parser

Exp → Exp **+** Exp

Exp → Exp * Exp

Exp → **id**

Exp → **num**

Exp → **(** Exp **)**

Exp → Term ( **+** Term )*
Term → Fator ( * Fator )*
Fator → **id** | **num** | **(** Exp **)**

# Criação da AST pelo Parser

Exp → Exp **+** Exp

Exp → Exp * Exp

Exp → **id**

Exp → **num**

Exp → **(** Exp **)**

Exp → Term ( **+** Term )*
Term → Fator ( * Fator )*
Fator → **id** | **num** | **(** Exp **)**

```
Exp Exp() : {Exp le,re;} {
  le=Term()
  ( "+" re=Term()  { le=new PlusExp(le,re); } )*
  { return le; }
}

Exp Term() : {Exp le,re;} {
  le=Fator()
  ( "*" re=Fator()  { le=new TimesExp(le,re); } )*
  { return le; }
}

Exp Fator() : { Exp e; Token t; } {
  ( t=<NUM> { return new Num(t.image); }
  | t=<ID>  { return new Id(t.image); }
  | "(" e=Exp() ")" { return e; }
  )
}
```

# Exemplo de utilização

```
public class MainExp {
  public static void main(String[] args) {
    BufferedReader input = new BufferedReader(
                  new InputStreamReader(System.in) );
    String line;
    Parser p;
    Exp exp;
    try {
      for(;;) {
      line = input.readLine();
      p= new Parser(new StringReader(line));
      exp = p.Exp();
      System.out.println("Eval = " + exp.eval());
      }
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

# Percorrer a AST

- Para verificar os limites das constantes é necessário acrescentar um método na interface Exp e implementá-lo nas classes concretas da AST.

```
public interface Exp {
   public int eval();
   public void verifyConsts() throws InvalidConst;
}
```

# Percorrer a AST

- Para verificar os limites das constantes é necessário acrescentar um método na interface Exp e implementá-lo nas classes concretas da AST.

```
public interface Exp {
   public int eval();
   public void verifyConsts() throws InvalidConst;
}
```

```
public abstract class BinExp implements Exp {
   public Exp left, right;
   ...
   public void verifyConsts() throws InvalidConst {
     left.verifyConsts(); right.verifyConsts();
}  }
```

# Percorrer a AST

- Para verificar os limites das constantes é necessário acrescentar um método na interface Exp e implementá-lo nas classes concretas da AST.

```
public interface Exp {
   public int eval();
   public void verifyConsts() throws InvalidConst;
}
```

```
public abstract class BinExp implements Exp {
   public Exp left, right;
   ...
   public void verifyConsts() throws InvalidConst {
     left.verifyConsts(); right.verifyConsts();
}  }
```

```
public class Num implements Exp {
   public String num;
   ...
   public void verifyConsts() throws InvalidConst {
     if (num.length()>MAXDIGITS ||...) throw new InvalidConst(num);

}  }
public class Id implements Exp {
   ...
   public void verifyConsts() {}
}
```

# Percorrer a AST (repetição)

- Para verificar se as variáveis estão definidas é também necessário acrescentar um método na interface Exp e implementá-lo nas classes concretas da AST.

```
public interface Exp {
   public int eval();
   public void verifyConsts() throws InvalidConst;
   public void verifyVars() throws UndeVar;
}
```

- Numa AST podem existir muitas classes.
- Cada vez que se acrescenta uma funcionalidade (interpretação) é necessário alterar todas as classes?

- Para melhorar a modularidade (recompilação, dimensão):
  - O código de cada interpretação deve ficar separado das classes da AST

# Interpretação com *instanceof* e *cast*

- Uma solução seria:

```
public class VerifyConsts {
  public verify( Exp node ) throws InvalidConst {
    if ( node instanceof BinExp ) {
      BinExp be = (BinExp) node;
      verify( be.left );
      verify( be.right );
    }
    else if ( node instanceof Num ) {
      Num n = (Num) node;
      if (n.num.length()>MAXDIGITS||...)
        throw InvalidConst(n.num);
    }
    else ...
  }
  ...
}
```

# Interpretação com *instanceof* e *cast*

- Uma solução seria:

```
public class VerifyConsts {
  public verify( Exp node ) throws InvalidConst {
    if ( node instanceof BinExp ) {
      BinExp be = (BinExp) node;
      verify( be.left );
      verify( be.right );
    }
    else if ( node instanceof Num ) {
      Num n = (Num) node;
      if (n.num.length()>MAXDIGITS||...)
        throw InvalidConst(n.num);
    }
    else ...
  }
  ...
}
```

- Esta solução é eficiente?
- Será possível usar uma solução polimórfica?

# Interpretação com *Visitor*

- As interfaces ou classes abstratas declaram o método que aceita o *visitor*
- As classes concretas implementam o método para chamar o método específico do *visitor*.

```
public interface Exp {
   ...
   public abstract void accept(Visitor v) throws Exception;
}
```

```
public class PlusExp extends BinExp {
   ...
   public void accept(Visitor v) throws Exception { v.visit(this); }
}
...
public class Num implements Exp {
   ...
   public void accept(Visitor v) throws Exception { v.visit(this); }
}
public class Id implements Exp {
   ...
   public void accept(Visitor v) throws Exception { v.visit(this); }
}
```

# Interpretação com *Visitor*

- A interface **`Visitor`** tem um método para cada classe concreta da AST.
- Os *visitors* específicos implementam cada um dos métodos.

```
public interface Visitor {
  public void visit(PlusExp n) throws Exception;
  public void visit(TimesExp n) throws Exception;
  public void visit(Num n) throws Exception;
  public void visit(Id n) throws Exception;
}
```

```
public class VerifyConsts implements Visitor {

  public void visit(PlusExp n) throws InvalidConst
    { n.left.accept(this); n.right.accept(this); }

  public void visit(TimesExp n) throws InvalidConst
    { n.left.accept(this); n.right.accept(this); }

  public void visit(Num n) throws InvalidConst
    { if (n.num.length()>MAXDIGITS ||...)
        throw new InvalidConst(n.num);
    }
  public void visit(Id n) {};
}
```

# Comparação das implementações

| | Recompilações frequentes? Difícil de manter? | Teste exaustivo do tipo? |
|---|---|---|
| Métodos dedicados na AST | Sim | Não |
| *Instanceof* e *cast* | Não | Sim |
| Padrão de projeto *Visitor* | Não | Não |

# Informação sobre posição

- Quando é detectado um erro durante uma interpretação
  - Como saber a posição (linha  e coluna) do erro?
- Esta informação existe durante o *parsing* em cada *token* retornado pelo analisador léxico
- Os nós da AST gerados a partir de *tokens* como <ID> e <NUM> devem ficar com informação de posição

```
public class Num implements Exp {
   public Token num;
   public Num(Token tk) { num = tk; }
   public int eval() {
      return Integer.parseInt(num.image);
   }
   public Position getPos() {
      return new Position(num.beginLine, num.beginColumn);
   }
}
```

# Linguagem MiniJava

- Subconjunto da linguagem Java
- Tipos:
  - Só tem os tipos primitivos int e boolean
  - Arrays só de inteiros (int[]) onde é possível usar o campo readonly length
  - Classes
- Não tem sobrecarga de métodos
- A instrução System.out.println(…); só apresenta valores inteiros
- Operadores binários: &&, <, +, - e *
- Comentários, identificadores, constantes (int e boolean) iguais ao Java

# Gramática do MiniJava

Programa → ClasseMain ClassDecl*
ClasseMain → **class** ID **{ public static void main ( String [ ]** ID **) { Cmd } }**
ClassDecl → **class** ID [ **extends** ID ] **{** DeclVar* DeclMetodo* **}**
DeclVar → Tipo ID **;**
DeclMetodo → **public Tipo** ID **( ListaFormal ) { DeclVar*** Cmd* **return** Exp**; }**
ListaFormal → [ **Tipo** ID ( **, Tipo** ID )* ]
Cmd → **if (** Exp **)** Cmd **else** Cmd | **while (** Exp **)** Cmd
| **System.out.println (** Exp **) ;**
| ID [ **[** Exp **]** ] **=** Exp **;**
| **{** Cmd* **}**
Tipo → **boolean** | **int** | **int [ ]** | ID
Exp → Exp ( **&&** | **<** | **+** | **-** | * ) Exp | **!** Exp | **new int [** Exp **]** | **new** ID **( )**
| Exp **[** Exp **]** | Exp **. length** | Exp **.** ID **(** [ Exp ( **,** Exp )* ] **)**
| **true** | **false** | **this** | NUM | ID | **(** Exp **)**

# Exemplo em MiniJava

```
class Fatorial {
  public static void main( String[] a ) {
    Fat f = new Fat();
    System.out.println( f.calc(10) );
  }
}

class Fat {
  public int calc( int num ) {
    int res;
    if ( num < 1 )
      res = 1;
    else
      res = num * this.calc( num-1 );
    return res;
  }
}
```

# AST para MiniJava

Programa(ClasseMain mc, List<ClassDecl> cs)
ClasseMain(Id n, Cmd main)
ClassDecl(Id n, Id s, List<DeclVar> vs, List<DeclMetodo> ms)
DeclMetodo(Tipo t, Id n, List<Formal> f, List<DeclVar> vs, List<Cmd> s, Exp e)
DeclVar(Tipo t, Id n)  Formal(Tipo t, Id n)  Id(String s)

Tipo:

IntArrayType()  BooleanType()  IntegerType()  IdType(String n)

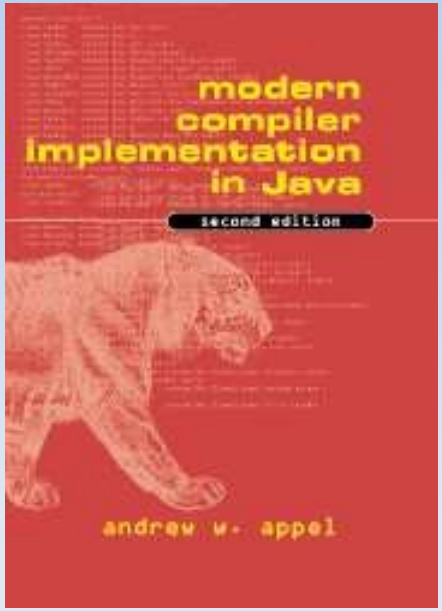
Cmd:

If(Exp e, Cmd ts, Cmd es)  While(Exp e, Cmd s)  Print(Exp e)
Assign(Id v, Exp e)  ArrayAssign(Id v, Exp idx, Exp e)  Block(List<Cmd> ss)

Exp:

E(Exp e1, Exp e2)  Menor(Exp e1, Exp e2)  Nao(Exp e)
Mais(Exp e1, Exp e2)  Menos(Exp e1, Exp e2)  Vezes(Exp e1, Exp e2)
ArrayLookup(Exp a, Exp idx)  ArrayLength(Exp a)
Call(Exp e, Id m, List<Exp> as)  ExpId(String s)
NovoArray(Exp e)  NovoObjeto(Id n)
Num(int v)  True()  False()  This()

# Referências




Andrew W. Appel,
“Modern Compiler Implementation in Java”,
second edition

Capítulo 4