Abstract Syntax Tree

Guido Araújo
guido@ic.unicamp.br

Gramática para Expressões

$E \rightarrow E + T$
$E \rightarrow E - T$

$E \rightarrow E + T$
$E \rightarrow E + T$

$E \rightarrow id$
$E \rightarrow num$

$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$

$T \rightarrow T * F$
$T \rightarrow T / F$
$T \rightarrow F$

$F \rightarrow id$
$F \rightarrow num$
$F \rightarrow (E)$

$S \rightarrow E \$$
$E \rightarrow T\ E'$
$E' \rightarrow + T\ E'$
$E' \rightarrow - T\ E'$
$E' \rightarrow$

$T \rightarrow F\ T'$
$T' \rightarrow * F\ T'$
$T' \rightarrow / F\ T'$
$T' \rightarrow$

$F \rightarrow id$
$F \rightarrow num$
$F \rightarrow (E)$

Recursive Descendent Parser

```
class Token {int kind; Object val;
             Token(int k, Object v) {kind=k; val=v;}
            }
final int EOF=0, ID=1, NUM=2, PLUS=3, MINUS=4,  ···

int lookup(String id) {  ···  }

int F_follow[] = { PLUS, TIMES, RPAREN, EOF };

int F() {switch (tok.kind) {
         case ID:    int i=lookup((String)(tok.val)); advance(); return i;
         case NUM:   int i=((Integer)(tok.val)).intValue();
                     advance(); return i;
         case LPAREN:  eat(LPAREN);
                       int i = E();
                       eatOrSkipTo(RPAREN, F_follow);
                       return i;
         case EOF:
         default:    print("expected ID, NUM, or left-paren");
                     skipto(F_follow); return 0;
        }}
```

$F \rightarrow \text{id}$

$F \rightarrow \text{num}$

$F \rightarrow (E)$

Recursive Descendent Parser (cont.)

```
int T_follow[] = { PLUS, RPAREN, EOF };

int T() {switch (tok.kind) {
          case ID:
          case NUM:
          case LPAREN: return Tprime(F());
          default: print("expected ID, NUM, or left-paren");
                   skipto(T_follow);
                   return 0;
         }}

int Tprime(int a)  {switch (tok.kind) {
         case TIMES: eat(TIMES); return Tprime(a*F());
         case PLUS:
         case RPAREN:
         case EOF:  return a;
         default:  ···
        }}

void eatOrSkipTo(int expected, int[] stop) {
   if (tok.kind==expected)
        eat(expected);
   else {print(···); skipto(stop);}
}
```

$T \rightarrow F\ T'$

$T' \rightarrow^{*} F\ T'$

$T' \rightarrow / F\ T'$

$T' \rightarrow$

Geração Automática: JavaCC

```
void Start() :
{ int i; }
{ i=Exp() <EOF>  { System.out.println(i); }
}
int Exp() :
{ int a,i; }
{ a=Term()
  ( "+" i=Term() { a=a+i; }
  | "-" i=Term() { a=a-i; }
  )*
  { return a; }
}
int Term() :
{ int a,i; }
{ a=Factor()
  ( "*" i=Factor() { a=a*i; }
  | "/" i=Factor() { a=a/i; }
  )*
  { return a; }
}
int Factor() :
{ Token t; int i; }
{ t=<IDENTIFIER>       { return lookup(t.image); }
| t=<INTEGER_LITERAL> { return Integer.parseInt(t.image); }
| "(" i=Exp() ")"      { return i; }
}
```

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow \text{id}$

$F \rightarrow \text{num}$

$F \rightarrow (E)$

Construindo Abstract Syntax Tree

```
Exp Start() :
  { Exp e; }
  { e=Exp() { return e; }
  }
Exp Exp() :
  { Exp e1,e2; }
  { e1=Term()
      ( "+" e2=Term() { e1=new  PlusExp(e1,e2); }
      | "-" e2=Term() { e1=new MinusExp(e1,e2); }
      )*
    { return e1; }
  }
Exp Term() :
  { Exp e1,e2; }
  { e1=Factor()
      ( "*" e2=Factor() { e1=new  TimesExp(e1,e2); }
      | "/" e2=Factor() { e1=new DivideExp(e1,e2); }
      )*
    { return e1; }
  }
Exp Factor() :
  { Token t; Exp e; }
  { ( t=<IDENTIFIER>      { return new Identifier(t.image); } |
      t=<INTEGER_LITERAL> { return new IntegerLiteral(t.image); } |
      "(" e=Exp() ")"     { return e; } )
  }
```

Abstract Syntax Tree (AST)

```
public abstract class Exp {
   public abstract int eval();
}
public class PlusExp extends Exp {
   private Exp e1,e2;
   public PlusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int eval() {
       return e1.eval()+e2.eval();
   }
}
public class MinusExp extends Exp {
   private Exp e1,e2;
   public MinusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int eval() {
       return e1.eval()-e2.eval();
   }
}
public class TimesExp extends Exp {
   private Exp e1,e2;
   public TimesExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int eval() {
       return e1.eval()*e2.eval();
   }
}
```

Abstract Syntax Tree (AST)

```
public class DivideExp extends Exp {
   private Exp e1,e2;
   public DivideExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int eval() {
       return e1.eval()/e2.eval();
   }
}
public class Identifier extends Exp {
   private String f0;
   public Identifier(String n0) { f0 = n0; }
   public int eval() {
       return lookup(f0);
   }
}
public class IntegerLiteral extends Exp {
   private String f0;
   public IntegerLiteral(String n0) { f0 = n0; }
   public int eval() {
       return Integer.parseInt(f0);
   }
}
```

cada novo tipo ação (eval) requer modificar todas as classes da AST

Separando Sintaxe de Ação

- Vantagem é a ausência da necessidade de recompilar classes da AST
- Mas requer *instanceof* em cada ação para descobrir a classe ☹

ação
```
TypeCheck(Node n) {
……
if (n instanceof VarDecl) {….}
else if (n instanceof Plus) {….}
```

Visitors

- Padrão de projeto que permite separar ações dos objetos que são seu alvo.
  - Permite inserir novas ações sem recompilar os objetos
  - Não requer o uso de *instanceof* para detectar o tipo do objeto
- Funcionamento
  - Toda classe tem um método *accept* que recebe o tipo *Visitor*
  - Toda ação tem uma interface*Visitor* com as assinaturas dos seus métodos *visit* para cada subtipo específico de *node*
  - A ação chama *accept* passando o *Visitor*
  - O objeto chama o *visit* específico para o subtipo de *node*

Visitors

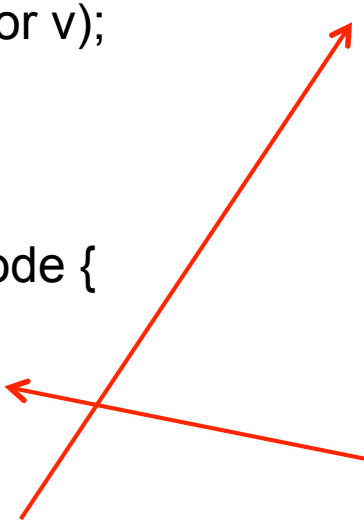
Ex: x + 2

```
abstract class Node {
.....
abstract void accept (Visitor v);
……
}

class Identifier extends Node {
      ……
    void accept(Node v) {
       …..
       v.visit(this);
       …..
    }
}
```

```
interface Visitor {
.....
int visit(Identifier n);
……


class Interpreter implements Visitor {
     ……
     int visit (PlusExp Node n) {
          ……
          n.e1.accept(this);
          n.e2.accept(this);
          ……
     }
}
```



Interpretador usando Visitor

```
public interface Visitor {
   public int visit(PlusExp n);
   public int visit(MinusExp n);
   public int visit(TimesExp n);
   public int visit(DivideExp n);
   public int visit(Identifier n);
   public int visit(IntegerLiteral n);
}

public class Interpreter implements Visitor {
   public int visit(PlusExp n) {
       return n.e1.accept(this)+n.e2.accept(this);
   }
   public int visit(MinusExp n) {
       return n.e1.accept(this)-n.e2.accept(this);
   }
   public int visit(TimesExp n) {
       return n.e1.accept(this)*n.e2.accept(this);
   }
   public int visit(DivideExp n) {
       return n.e1.accept(this)/n.e2.accept(this);
   }
   public int visit(Identifier n) {
       return lookup(n.f0);
   }
   public int visit(IntegerLiteral n) {
       return Integer.parseInt(n.f0);
   }
}
```

Interpretador Simples

- Livro descreve um interpretador simples para MiniJava
- Possui duas classes abstratas básicas
  - Stm e Exp
  - São especializadas para os tipos diferentes de vértices
  - Implementam métodos eval para realizar computação

MiniJava AST

```
package syntaxtree;

Program(MainClass m, ClassDeclList cl)
MainClass(Identifier i1, Identifier i2, Statement s)

abstract class ClassDecl
ClassDeclSimple(Identifier i, VarDeclList vl, MethodDeclList ml)
ClassDeclExtends(Identifier i, Identifier j,
                 VarDeclList vl, MethodDeclList ml)  see Ch.14

VarDecl(Type t, Identifier i)
MethodDecl(Type t, Identifier i, FormalList fl, VarDeclList vl,
           StatementList sl, Exp e)
Formal(Type t, Identifier i)

abstract class Type
IntArrayType()  BooleanType()  IntegerType()  IdentifierType(String s)
```

MiniJava AST

```
abstract class Statement
Block(StatementList sl)
If(Exp e, Statement s1, Statement s2)
While(Exp e, Statement s)
Print(Exp e)
Assign(Identifier i, Exp e)
ArrayAssign(Identifier i, Exp e1, Exp e2)

abstract class Exp
And(Exp e1, Exp e2)
LessThan(Exp e1, Exp e2)
Plus(Exp e1, Exp e2) Minus(Exp e1, Exp e2) Times(Exp e1, Exp e2)
ArrayLookup(Exp e1, Exp e2)
ArrayLength(Exp e)
Call(Exp e, Identifier i, ExpList el)
IntegerLiteral(int i)
True()
False()
IdentifierExp(String s)
This()
NewArray(Exp e)
NewObject(Identifier i)
Not(Exp e)

Identifier(String s)

 list classes
ClassDeclList() ExpList() FormalList() MethodDeclList() StatementList() VarDeclList()
```

MiniJava Visitor

```
public interface Visitor {
  public void visit(Program n);
  public void visit(MainClass n);
  public void visit(ClassDeclSimple n);
  public void visit(ClassDeclExtends n);
  public void visit(VarDecl n);
  public void visit(MethodDecl n);
  public void visit(Formal n);
  public void visit(IntArrayType n);
  public void visit(BooleanType n);
  public void visit(IntegerType n);
  public void visit(IdentifierType n);
  public void visit(Block n);
  public void visit(If n);
  public void visit(While n);
  public void visit(Print n);
  public void visit(Assign n);
  public void visit(ArrayAssign n);
  public void visit(And n);
  public void visit(LessThan n);
  public void visit(Plus n);
  public void visit(Minus n);
  public void visit(Times n);
  public void visit(ArrayLookup n);
  public void visit(ArrayLength n);
  public void visit(Call n);
  public void visit(IntegerLiteral n);
  public void visit(True n);
  public void visit(False n);
  public void visit(IdentifierExp n);
  public void visit(This n);
  public void visit(NewArray n);
  public void visit(NewObject n);
  public void visit(Not n);
  public void visit(Identifier n);
}
```