1 Part V: Abstract Syntax

- Abstract Syntax
- Abstract Syntax Trees
- Accessing Trees
- Object Oriented Decomposition
- Visitors

2 Syntax Trees

- In a multi - pass compiler the parser builds a syntax tree explicitely.
- All later phases of a compiler work on the *abstract syntax tree*, not the program source.
- The tree could be the concrete syntax tree (parse tree) corresponding to the context-free grammar.
- Usually, there is a better choice.

3 Abstract Syntax / Concrete Syntax

Compare to the concrete syntax tree, some simplifications are possible, hence

- No need for parentheses: $\mathsf{A} * (\mathsf{B} + \mathsf{C})$ becomes ...
- No need to maintain terminals **if** $(\mathsf{x} == 0)$ $\mathsf{y} = 1$; **else** $\mathsf{y} = 2$; becomes ...

4 Abstract Syntax Tree

- An abstract syntax tree is a tree with one kind of node for each alternative in the abstract syntax.
- We represent a tree using a set of Java classes, one for each alternative.
- Common abstract superclass: Tree.
- Each class represents subtrees as instance variables.
- Each class has a constructor to construct a node of the given kind.

5 Abstract Syntax Tree (2)

Abstract Syntax of Expressions

```
Expression = OPERATION Expression Expression Op
           | NUMLIT int
           ;
```

Parenthesis are not necessary in abstract syntax!

```
public abstract class Tree {
    public static class NumLit extends Tree {
        int num;
    }
    public static class Operation extends Tree {
        Tree left, right;
        int op;
    }
}
```


6 Accessing Trees

- Abstract syntax trees are the central input data structures of later phases of the compiler.
- It is important to find a representation, which can be used in flexible ways.
- How do tree processors access the tree?
- Simple (and crude) solution: use **instanceof** to find out the kind of the tree node and then cast to access tree elements.

```
if (tree instanceof NumLit) {
    return ((NumLit) tree).num;
}
```

- This is neither elegant nor efficient.
- Better solution: object-oriented decomposition
- Even better solution: Visitors.

7 Example: Expressions

- We now present both object-oriented decomposition and visitor access, using arithemetic expressions as an example.
- Two kind of nodes: Operation, NumLit
- Two kind of actions: eval, print
- Very simple example.
- Typical languages have 20 (Jex) - 40 (Java) or more kinds of nodes.
- A typical compiler has 5-10 processors.
- But the basic framework stays the same.

8 The Tree Class

```
public abstract class Tree {
    int pos;
    public Tree(int pos) {
        this.pos = pos;
    }
    public static class NumLit extends Tree {
        int num;
        public NumLit(int pos, int num) {
            super(pos);
            this.num = num;
        }
    }
```

```
public static class Operation extends Tree {
    Tree left, right;
    int op;
    public Operation(int pos, Tree left, Tree right, int op) {
        super(pos);
        this.left = left;
        this.right = right;
        this.op = op;
    }
  }
}
```


9 A Parser that builds a Tree

- We introduce types for terminals and non terminals

  terminal Integer NUMLIT;
  non terminal Tree Program, Expression, Term, Factor;
- We give names to the components of a rule

  Expression ::= Expression:e PLUS:o Term:t
- We construct the tree for the left-hand side from the components for the right-hand side

```
Expression ::= Expression:e PLUS:o Term:t
                 {: RESULT = new Tree.Operation(
                       oleft, e, t, Tokens.PLUS); :}
```
- We attach left to a name to get the position. oleft refers to the position of the PLUS.

10 A Parser that builds a Tree (2)

```
Program     ::= Expression:e
                {: RESULT = e; :}
              ;
Expression  ::= Expression:e PLUS:o Term:t
                {: RESULT = new Tree.Operation(
                        oleft, e, t, Tokens.PLUS); :}
              | Expression:e MINUS:o Term:t
                {: RESULT = new Tree.Operation(
                        oleft, e, t, Tokens.MINUS); :}
              | Term:t
                {: RESULT = t; :}
              ;
```



```
Term     ::= Term:t TIMES:o Factor:f
             {: RESULT = new Tree.Operation(
                   oleft, t, f, Tokens.TIMES); :}
             Term:t DIV:o Factor:f
             {: RESULT = new Tree.Operation(
                   oleft, t, f, Tokens.DIV); :}
           | Factor:f
             {: RESULT = f; :}
           ;
Factor   ::= NUMLIT:n
             {: RESULT = new Tree.NumLit(
                   nleft, n.intValue()); :}
           | LPAREN Expression:e RPAREN
             {: RESULT = e; :}
           ;
```

11 Object-oriented Decomposition

- Every tree processor P is represented by a dynamic method P() in every tree class.
- The method is abstract in class Tree, implemented in every subclass.
- To process a subtree, simply call its processor method t.P().
- In our example: define methods eval() and print() in classes NumLit and Operation
- The methods eval() and print() are abstract in Tree, so they can be invoked on every tree.
- What they do will depend on the concrete kind of tree.

12 Object-oriented Decomposition

```
public abstract class Tree {
    int pos;
    public Tree(int pos) { ... }
    public abstract void print();
    public abstract int eval();
    public static class NumLit extends Tree {
        int num;
        public NumLit(int pos, int num) { ... }
        public void print() {
            System.out.print("" + num);
        }
        public int eval() {
            return num;
        }
    }
```



```
    public static class Operation extends Tree {
        Tree left, right
        int op;
        public Operation(int pos, Tree left,
                    Tree right, int op) { ... }
        public void print() {
            System.out.print("(");
            left.print();
            System.out.print(" "
                    + Scanner.representation(op) + " ");
            right.print();
            System.out.print(")");
        }
```



```
        public int eval() {
            int l = left.eval();
            int r = right.eval();
            switch(op) {
                case Tokens.PLUS:
                    return l + r;
                case Tokens.MINUS:
                    return l - r;
                case Tokens.TIMES:
                    return l * r;
                case Tokens.DIV:
                    return l / r;
                default:
                    throw new InternalError();
            }
        }
    }
}
```

13 A Driver Class

```
class EvalTest {
    public static void main(String args[ ]) throws Exception {
        jaco.framework.parser.Symbol sym;
        sym = new Parser(new Scanner(System.in)).parse();
        if (sym.value != null) {
            ((Tree)sym.value).print();
            System.out.println(" = " + ((Tree)sym.value).eval());
        }
    }
}
```

```
java expression.EvalTest
3 * (2 - 5)
(3 * (2 - 5)) = -9
```


14 A Typical Stack Trace

```
(#*).print()
(#-).print()
(#2).print()
System.out.print(2)
```

```
(3 * (
```


15 Extensibility

- With an abstract syntax tree, there can be extensions in two dimensions.
  - Add a new kind of node.
  - Add a new kind of processor method.
- Which one is more common?
- Which one is easier to do?
- Add a new kind of node: add a new subclass.
- Add a new kind of processor method: add processor method to every subclass.

16 Visitors

- The visitor design pattern allows simple extension by new processors.
- All methods of a processor are grouped together in a visitor object
  ⇒it is easy to share common code and data
- A visitor object contains for each kind K of trees a method called caseK that can process trees of that kind.
- The tree contains only a simple generic processor method which applies a given visitor object.

17 Visitable Trees for Expressions

```
public abstract class Tree {
    int pos;
    public Tree(int pos) { ... }

    public abstract void apply(Visitor v);

    public static class NumLit extends Tree {
        int num;
        public NumLit(int pos, int num) { ... }
        public void apply(Visitor v) {
            v.caseNumLit(this);
        }
    }
```



```
    public static class Operation extends Tree {
        Tree left, right;
        int op;
        public Operation(int pos, Tree left, Tree right, int op) { ... }
        public void apply(Visitor v) {
            v.caseOperation(this);
        }
    }

    public interface Visitor {
        void caseOperation(Operation tree);
        void caseNumLit(NumLit tree);
    }
}
```


18 A Print Visitor

```
public class Printer implements Tree.Visitor {
    public static void print(Tree tree) {
        tree.apply(new Printer());
    }
    public void caseOperation(Tree.Operation tree) {
        System.out.print("(");
        print(tree.left);
        System.out.print(" "
            + Scanner.representation(tree.op) + " ");
        print(tree.right);
        System.out.print(")");
    }
    public void caseNumLit(Tree.NumLit tree) {
        System.out.print("" + tree.num);
    }
}
```


19 A Typical Stack Trace

We call the Printer objects P, P', P'':

```
Printer.print(#*)
(#*).apply(P)
P.caseOperation(#*)
Printer.print(#-)
(#-).apply(P')
P'.caseOperation(#-)
Printer.print(#2)
(#2).apply(P'')
P''.caseNumLit(#2)
System.out.print(2)
```

(3 * (

20 Reusing the Visitor

- Creating a new visitor object for every invocation is expensive.
- One routine is globally available and creates a new visitor.
- Another routine is local and reuses the visitor.

21 Reusing the Visitor

```
public class Printer implements Tree.Visitor {
    public static void print(Tree tree) { tree.apply(new Printer()); }
    protected void printRec(Tree tree)  { tree.apply(this); }
    public void caseOperation(Tree.Operation tree) {
        System.out.print("(");
        printRec(tree.left);
        System.out.print(" "
            + Scanner.representation(tree.op) + " ");
        printRec(tree.right);
        System.out.print(")");
    }
    public void caseNumLit(Tree.NumLit tree) { ... }
}
```

Exercise: Give a call-tree for 3+4+5, in object-oriented and vistor style.

22 An Evaluation Visitor

- Because we have only one general apply method, we have to pass the result differently.
- We keep it in a local instance variable val, that eval reads after apply finished.

```
public class Evaluator implements Tree.Visitor {
    int val;
    public static int eval(Tree tree) {
        Evaluator ev = new Evaluator();
        tree.apply(ev);
        return ev.val;
    }
    public void caseNumLit(Tree.NumLit tree) {
        val = tree.num;
    }
```



```
    public void caseOperation(Tree.Operation tree) {
        switch (tree.op) {
            case Tokens.PLUS:
                val = eval(tree.left) + eval(tree.right);
                break;
            case Tokens.MINUS:
                val = eval(tree.left) - eval(tree.right);
                break;
            case Tokens.TIMES:
                val = eval(tree.left) * eval(tree.right);
                break;
            case Tokens.DIV:
                val = eval(tree.left) / eval(tree.right);
                break;
            default: throw new InternalError();
        }
    }
}
```

23 Reusing the Evaluation Visitor

- A case might set val and afterwards do a recursive call
- We have to save val before each recursive call and reset it after the call.

```
public class Evaluator implements Tree.Visitor {
    int val;
    public static int eval(Tree tree) { ... }
    public static int evalRec(Tree tree) {
        int saveVal, retVal;
        saveVal = val;
        tree.apply(this);
        retVal = val;
        val = saveVal;
        return retVal;
    }
```


24 Driver Class for Visitors

```
class EvalTest {
    public static void main(String args[ ]) throws Exception {
        jaco.framework.parser.Symbol sym;
        sym = new Parser(new Scanner(System.in)).parse();
        if (sym.value != null) {
            Printer.print((Tree)sym.value);
            System.out.println(" = " +
                    Evaluator.eval((Tree)sym.value));
        }
    }
}
```


25 Which one is better ?

- Extensibility
  - OO Decomposition makes adding new kinds of nodes easy
  - Visitors make adding of new processors easy
- Modularity
  - OO allows sharing of data and code in a tree node between phases
  - Visitors allow sharing of data and code between methods of same processor.
- Which is more important?

26 Trees in Other Contexts

- Trees with multiple kinds of nodes arise not only in compilation
- They are also found in text layout, structured documents such as HTML or XML, graphical user interfaces.
- Components of a GUI
  - Which method of tree access is used for GUI components?
  - Which kind of extension is more common?

27 Extensibility

Compiler

- Operations
  - type-check
  - translate to Pentium
  - translate to SPARC
  - optimize
  - find uninitialized vars
- Kinds
  - Ident
  - Numeric literal
  - String literal
  - If statement

GUI

- Operations
  - redisplay
  - move
  - iconize
  - highlight
- Kinds
  - Scrollbar
  - Menu
  - Canvas
  - Dialogbox
  - Statusbar

28 Abstract Syntax of Jex

```
Program     = DEFLIST { Definition | Statement }
            ;
Definition  = Formal
            | FUNDEF Type ident { Formal } Statement
            | IMPORT { ident } boolean
            ;
Formal      = VARDEF Type ident
            ;
Statement   = ASSIGN Expr Expr
            | IF Expr Statement Statement
            | WHILE Expr Statement
            | BLOCK { Statement | Formal }
            | EVAL Expr
            | RETURN Expr
            ;
```



```
Expr        = NUMLIT int
            | STRINGLIT String
            | BOOLEANLIT boolean
            | IDENT ident
            | FUNCALL ident { Expr }
            | METHODCALL Expr ident { Expr }
            | FIELDACCESS Expr ident
            | OPERATION Expr Expr op
            | NEW Type { Expr }
            ;
Type        = IDENT ident
            | INTEGERTP
            | BOOLEANTP
            ;
```


29 Abstract Syntax to Abstract Syntax Trees

- Simplify further by merging Definition, ExprOrType, Statement, and Formal into Tree:

```
Tree  = DEFLIST { Tree }
      | VARDEF Tree ident
      | FUNDEF ident { Tree } Tree
      | IMPORT { ident } boolean
      | NUMLIT int
      | STRINGLIT String
      ...
      | ASSIGN Tree Tree
      | IF Tree Tree Tree
      ...
```

- Define an abstract class Tree with inner subclasses for the nodes.
- Define a visitor interface.

30 The Tree Class for Jex

```
package jex;
public abstract class Tree {
    int pos;
    public Tree(int pos) { ... }
    public abstract void apply(Visitor v);

    public static class DefList extends Tree {
        Tree[ ] defs;
        public DefList(int pos, Tree[ ] defs) {
            super(pos);
            this.defs = defs;
        }
        public void apply(Visitor v) {
            v.caseDefList(this);
        }
    }
```



```
    public static class VarDef extends Tree {
        String name;
        Tree type;
        public VarDef(int pos, String name, Tree type) {
            super(pos);
            this.name = name;
            this.type = type;
        }
        public void apply(Visitor v) {
            v.caseVarDef(this);
        }
    }
    ...
```



```
    public interface Visitor {
        void caseDefList(DefList tree);
        void caseVarDef(VarDef tree);
        ...
    }
}
```


31 Explanations

- Each class has a constructor to construct a node of the given kind and an apply method which applies a visitor.
- Repetition is expressed by arrays. { T } in the syntax becomes T[ ] in the tree.
- Terminal symbols are represented by their essential information
  - for an IDENT: the string naming the identifier.
  - for a NUMLIT: its value as an integer.
  - for a STRINGLIT: its value as a string.
- The pos field contains the current position of the tree, important for error messages. this field is common for all kind of trees; that's why it is a member of Tree.

32 Helper class: TreeArrayBuffers

```
class TreeArrayBuffer {
/** append a tree to the end of the buffer
  */
      public TreeArrayBuffer append(Tree t) { ... }
/** return current elements as an array. the length of the
  * returned array matches exactly the number of elements
  * in the buffer.
  */
      public Tree[ ] toArray() { ... }

/** appends a tree to a tree array; this method can be
  * used to append a tree to an array after toArray was
  * called
  */
      public static Tree[ ] append(Tree[ ] trees, Tree tree) { ... }
}
```


33 Building the Tree

```
non terminal Tree Program, Definition, FunDef, Import;
non terminal TreeArrayBuffer DefList;
non terminal StringArrayBuffer Package;

Program  ::= DefList:p
             {: RESULT = new Tree.DefList(pleft, p.toArray()); :}
           ;
DefList  ::= /* empty */
             {: RESULT = new TreeArrayBuffer(); :}
           | DefList:p Statement:s
             {: RESULT = p.append(s); :}
           | DefList:p Definition:d
             {: RESULT = p.append(d); :}
           ;
```



```
Definition ::= Formal:f SEMI
               {: RESULT = f; :}
             | FunDef:f
               {: RESULT = f; :}
             | Import:i
               {: RESULT = i; :}
             ;
Import     ::= IMPORT:im Package:p TIMES SEMI
               {: RESULT = new Tree.Import(imleft, p.toArray(), true); :}
             | IMPORT:im Package:p IDENT:id SEMI
               {: RESULT = new Tree.Import(imleft,
                  p.append(id).toArray(), false); :}
             ;
```



```
Package  ::= /* empty */
             {: RESULT = new StringArrayBuffer(); :}
           | Package:p IDENT:id DOT
             {: RESULT = p.append(id); :}
           ;
FunDef   ::= Type:tp IDENT:id LPAREN RPAREN Statement:s
             {: RESULT = new Tree.FunDef(idleft,
                   tp, id, new Tree[ ]{}, s); :}
           | Type:tp IDENT:id LPAREN Formals:fs RPAREN Statement:s
             {: RESULT = new Tree.FunDef(idleft,
                   tp, id, fs.toArray(), s); :}
           ;
```

34 Example Visitor: A Pretty Printer

```
public class Printer implements Tree.Visitor {

/** the main printer method
  */
        public static void prettyPrint(Tree tree) {
                tree.apply(new Printer());
                System.out.println();
        }

        protected void print(Tree tree) {
                tree.apply(this);
        }
```



```
/** return and align to lmargin
  */
        protected void align() {
                ...
        }
/** indent left margin
*/
        protected void indent() {
                ...
        }
/** reverse indentation
*/
        protected void undent() {
                ...
        }
```



```
/** the visitor methods
  */
        public void caseDefList(Tree.DefList tree) {
                for(int i = 0; i < tree.defs.length; i++) {
                        if (i > 0) align();
                        print(tree.defs[i]);
                }
        }

        public void caseVarDef(Tree.VarDef tree) {
                print(tree.type);
                System.out.print(" " + tree.name);
        }
```