Part IV : Abstract Syntax

- Semantic Actions
- Abstract Syntax
- Abstract Syntax Trees
- Accessing Trees
- OO
- Object-Oriented Decomposition
- Visitors

Semantic Actions

- A parser usually does more than just recognize syntax.
- It could :
  - Evaluate code (interpreter)
  - Emit code (single pass compiler)
  - Build an internal data structure (multi pass compiler)
- Generally, a parser performs *semantic actions.*
- In a recursive descent parser, semantic actions are embedded in the recognizer routines.
- In a machine generated bottom-up parser, they are added to the grammar submitted to the parser generator.

A Parser / Interpreter

The source language : Arithmetic expressions.

E = E + T | E - T
T = T * F | T / F
F = number | "(" E ")"

The source language in LL(1) form :

The Basic Parser Program

```
Package calc;

import java.io*;

class Parser extends Scanner {

  public Parser (InputStream in) {
    super (in);
    nextSym();
  }

  public void expression () {
    term();
    while (sym == PLUS || sym ==
           MINUS) {
      nextSym();
      term();
    }
  }

  public void term() {
    factor ();
    while (sym == MUL || sym == DIV) {
      nextSym();
      factor();
    }
  }

  public void factor () {
    if (sym == NUMBER) {
      nextSym();
    } else if (sym == LPAREN) {
      nextSym();
      expression();
      if (sym == RPAREN) nextSym();
      else error ("')' expected");
    } else error ("illegal start of
                   expression");
  }

}
```

The Interpreter

```
Package calc;

import java.io*;

class Parser extends Scanner {

  public Parser (InputStream in) {
    super (in);
    nextSym();
  }

  public int expression () {
    int v = term();
    while (sym == PLUS || sym ==
           MINUS) {
      int operator = sym; nextSym();
      int v2 = term();
      if (operator == PLUS) v = v + v2;
      else v = v - v2
    }
    return v;
  }

  public int term() {
    int v = factor ();
    while (sym == MUL || sym == DIV) {
      int operator = sym; nextSym();
      int v2 = factor();
      if (operator == MUL) v = v * v2;
      else v = v / v2;
    }
    return v;
  }

  public int factor () {
    int v = 0;
    if (sym == NUMBER) {
      v = Integer.parseInt(chars);
      nextSym();
    } else if (sym == LPAREN) {
      nextSym();
      v = expression();
      if (sym == RPAREN) nextSym();
      else error ("')' expected");
    } else error ("illegal start of
                   expression");
    return v;
  }

}
```

Syntax Trees

- In a multi-pass compiler, the parser builds a syntax tree explicitly.
- All later phases of a compiler work on the abstract syntax tree, not the program source.
- The tree can be the concrete syntax tree corresponding to the context-free grammar.
- But usually we use a simplified form.

Abstract Syntax vs Concrete Syntax

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

1. No need to parse text in the abstract language, hence ( ) unnecessary.

   A * (B + C)   becomes ...

2. No need to maintain terminal symbols.

   If (x == 0) y = 1 else y = 2   becomes ...

Abstract Syntax Trees

- 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.

Abstract Syntax Trees (2)

- Example: For arithmetic expressions:

```
abstract class Tree {
  class NumLit extends Tree {
    int value; ...
  }
  class Operation extends Tree {
    int operator; Tree left, right; ...
  }
}
```

Accessing Trees

- Abstract Syntax Trees are the central input data structures of later phases of the compiler.
- ⇒ 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 kind of tree node, then cast to access tree elements, E.g.

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

- But this is neither elegant nor efficient.
- Better Solution : Object-oriented decomposition.
- Even better solution : Visitors.

Example : Arithmetic Expressions

- We now present both object-oriented decomposition and visitor access, using arithmetic expressions as an example.
- Two kinds of nodes: **`Operation`**, **`NumLit`**.
- Two kinds of actions: **`eval`**, **`toString`**.
- Very simple example.
- Typical languages have 20 (J0), 40 (Java) or more kinds of nodes.
- A typical compiler has 5 - 10 processors.
- But these are differences of scale only - the basic framework stays the same.

Class Framework

```
package calc;

abstract class Tree {
   int pos; // position for error reports
            // maintained in all nodes.

   /** a subclass for trees representing
       numbers */
   static class NumLit extends Tree {

      int value;

      NumLit(int pos, int value) {
         this.pos = pos;
         this.value = value;
      }
   }

   /** a subclass for trees representing
       operations */
   Static Class Operation extends Tree {

      int operator;
      Tree left, right;

      Operation (int pos, int
                 operator, Tree
                 left, Tree right) {
         this.pos = pos;
         this.operator = operator;
         this.left = left;
         this.right = right;
      }
   }
}
```

A Parser which builds a tree

```
package calc;
import java.io*;
import calc.Tree.*;

class Parser extends Scanner {

  public Parser (InputStream in) {
      super (in);
      nextSym();
  }

public Tree expression() {
  Tree t = term();
  while (sym == PLUS ||
          sym == MINUS) {
      int startpos = pos;
      int operator = sym;
      nextSym();
      t = new Operation (startpos,
          operator, t, term());
  }
  return t;
}

public Tree term() {
  Tree t = factor();
  while (sym == MUL || sym = DIV) {
      int startpos = pos;
      int operator = sym;
      nextSym();
      t = new Operation (startpos,
          operator, t, factor());
    }
    return t;
  }

  public Tree factor() {
    Tree t = null;
    if (sym == NUMBER) {
      t = new NumLit (pos,
          Integer.parseInt (chars));
      nextSym();
    } else if (sym == LPAREN) {
      nextSym();
      t = expression();
      if (sym == RPAREN) nextSym();
      else error ("')' expected »);
    } else error("illegal start of
                  expression");
    return t;
  }
}
```

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 **`toString`** in classes **`NumLit`**, **`Operation`**.
- Methods **`eval`** and **`toString`** are abstract in class Tree, so they can be invoked on every tree.
- What they do will depend on the concrete kind of tree.

OO-Decomposition for Expressions

(constructors have been omitted)

```
package calc;
abstract class Tree {

  int pos;

  abstract public String toString();
  abstract public int eval();

  static class NumLit extends Tree {
     int value;
     public String toString() {
        return String.valueOf(value);
     }
     public int eval() {
        return value;
     }
  }

static class Operation extends Tree {
     int operator; Tree left, right;
     public String toString() {
        return "("+ left.toString() +
          Scanner.representation
            (operator) +
          right.toString() + ")";
     }

     public int eval() {
        int l = left.eval();
        int r = right.eval();
        switch (operator) {
        case Scanner.PLUS: return
                                 l + r;
        case Scanner.MINUS: return
                                  l - r;
        case Scanner.MUL: return
                                l * r;
        case Scanner.DIV: return
                                l / r;
        default: throw new Internal-
                   Error();
        }
     }
  }
}
```

A Driver Class

```
package calc;
class Main {

  static public void main(String[] args) {
    System.out.print("> ");
    Tree t = new Parser(System.in).expression();
    if (t != null)
      System.out.println(t.toString() + " evaluates
            to " + t.eval());
  }
}
```

- Usage:

```
java calc.Main
> 2 * (3 + 4);
(2'*'(3'+'4)) evaluates to 14
```

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 processor method: add processor methods to every subclass.

Visitors

- The visitor design pattern allows simple extension by new processors.
- All methods of a processor are grouped together in a visitor object

  ⇒ 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.

Visitable Trees for Expressions

(Constructors have been omitted)

```
package calc;
abstract class Tree {
  int pos;
  abstract void apply(Visitor v);
  static class NumLit extends Tree {
    int value;
    void apply(Visitor v) { v.caseNumLit(this); }
  }
  static class Operation extends Tree {
    int operator; Tree left, right;
    void apply(Visitor v) { v.caseOperation(this); }
  }
  interface Visitor {
    void caseNumLit(NumLit tree);
    void caseOperation(Operation tree);
  }
}
```

A ToString Visitor

```
package calc;
import calc.Tree.*;
class ToString implements Tree.Visitor {
  String result;
  public void caseNumLit(NumLit tree) {
    result = String.valueOf(tree.value);
  }
  public void caseOperation(Operation tree) {
    result = "(" + visit(tree.left) +
      Scanner.representation(tree.operator) +
      visit(tree.right) + ")";
  }
  static String visit(Tree tree) {
    ToString v = new ToString();
    tree.apply(v);
    return v.result;
  }
}
```

An Evaluation Visitor

```
package calc;
import calc.Tree.*;

class Eval implements Tree.Visitor {
   int result;

   public void caseNumLit (NumLit
   tree) {
      result = tree.value;
   }

   public void caseOperation
     (Operation tree) {
      int l = visit(tree.left);
      int r = visit(tree.right);
      switch (tree.operator) {
      case Scanner.PLUS :
         result = l + r; break;
      case Scanner.MINUS:
         result = l - r; break;
      case Scanner.MUL:
         result = l * r; break;
      case Scanner.DIV:
         result = l / r; break;
      default:
         throw new InternalError();
      }
   }

   static int visit(Tree tree) {
      Eval v = new Eval();
      tree.apply(v);
      return v.result;
   }
}
```

Driver Class for Visitors

```
package calc;

class Main {

  static public void main(String[] args) {
    System.out.print("> ");
    Tree t = new Parser(System.in).expression();
    if (t != null)
       System.out.println(ToString.visit(t) + " evaluates to "
          + Eval.visit(t));
  }
}
```

Which one is better ?

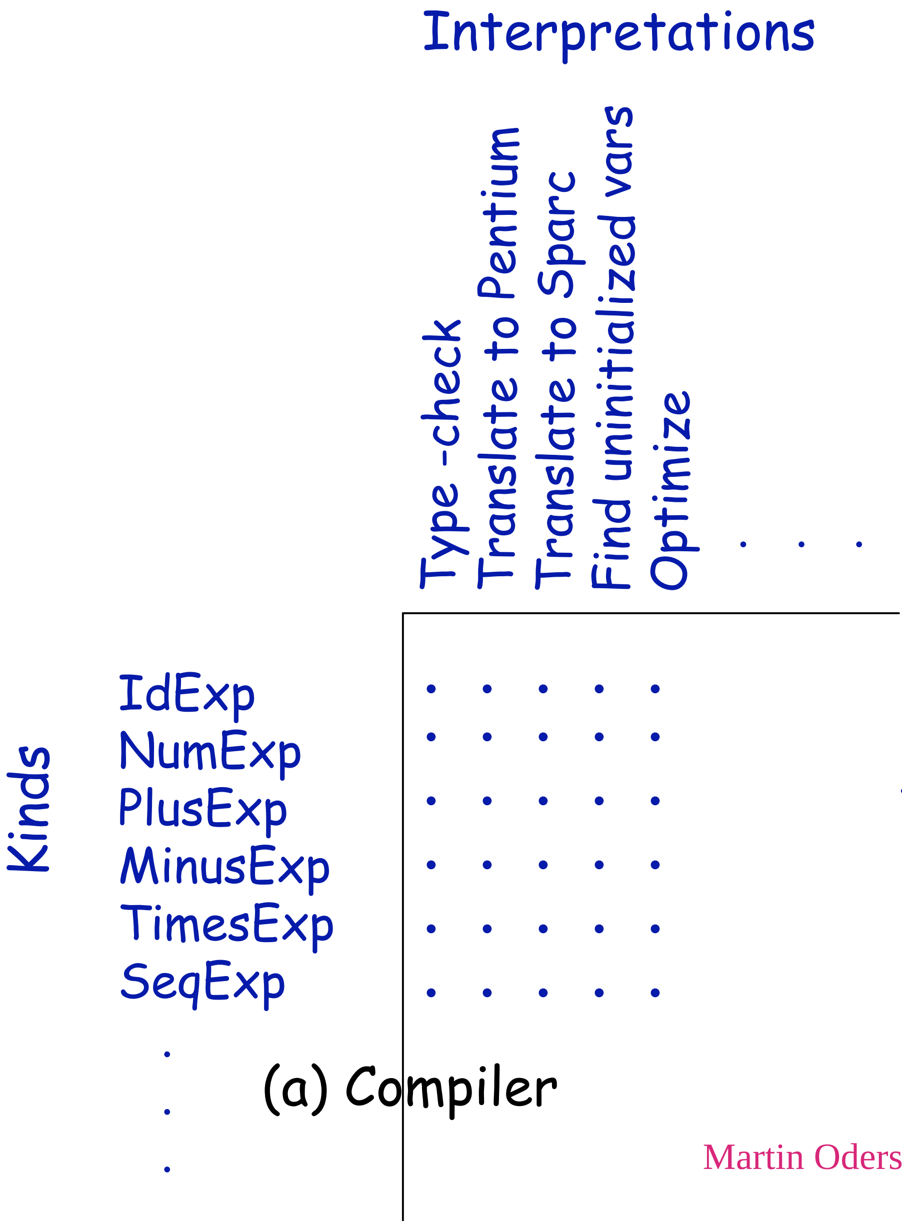
- Extensibility
  - OO-Decomp makes adding new kinds of nodes easy.
  - Visitors make adding new processors easy.
- Modularity
  - OO-Decomp allows sharing of data and code in a tree node between phases
  - Visitors allow sharing of data and code between methods of same processor.
- Which is more important?

Trees in Other Contexts

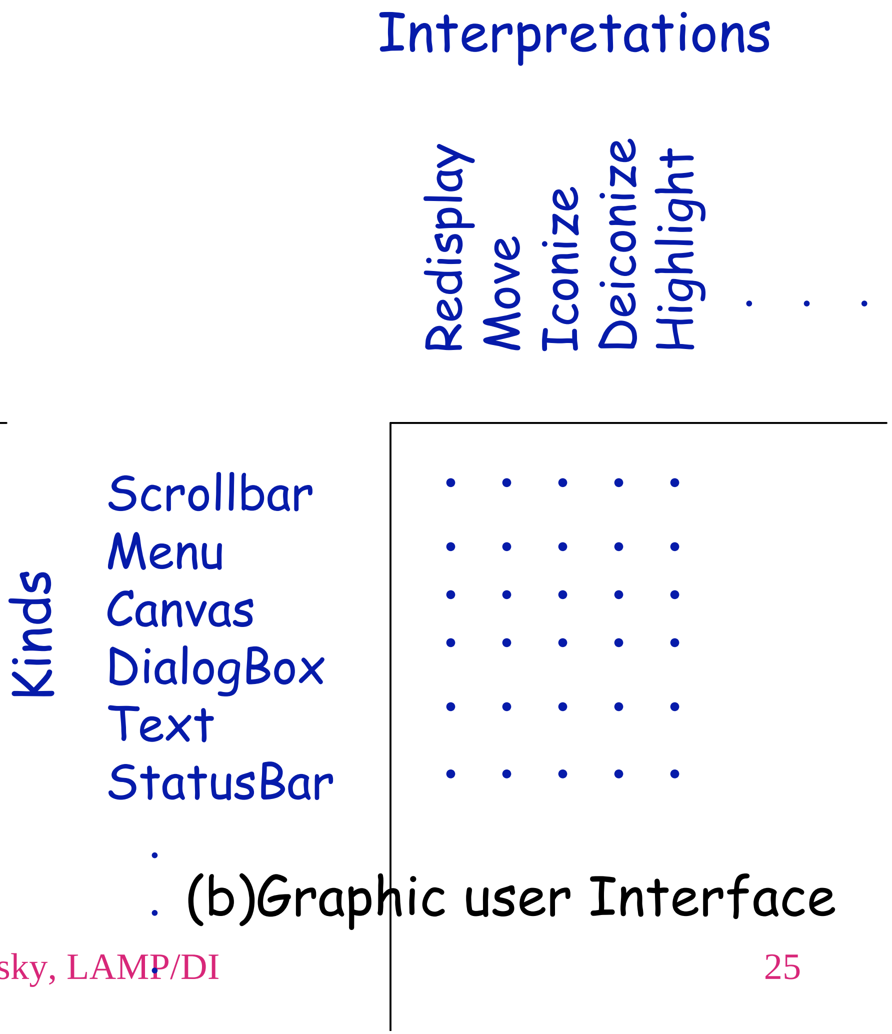
- Trees with multiple kinds of nodes arise not only in compilation
- They are also found in text layout, structured documents such as HTML or XML, graphical user interfaces.
- Example: Components of a GUI

- Which method of tree access is used for GUI components?
- Which kind of extension is more common?

Appel Extensibility



(a) Compiler



(b)Graphic user Interface

Abstract Syntax of JO

```
P  =  ModDecl ident {D}

D  =  FD | VD
VD =  VarDecl T ident
FD =  FunDecl RT ident {VD} S

S  =  VD
   |  FunCall ident {E}
   |  Assignment E E
   |  Block {S}
   |  IfStmt E S [S]
   |  WhileStmt E S
   |  ReturnStmt E

E  =  Operation Op E E
   |  FunCall ident {E}
   |  Indexed E E
   |  Ident ident
   |  NumLit int
   |  StrLit String
   |  NewArray T E

T  =  IntType
   |  StringType
   |  ArrayType T

RT =  T | VoidType

Op =  And | Or | Eq | Ne |
      Lt | Gt | Le | Ge |
      Add | Sub | Mul | Div |
      Mod | Neg | Not
```

From Abstract Syntax to Abstract Syntax Trees

- Simplify even further by merging P, D, S, E, T, ...

```
Tree = ModDecl ident {Tree}
     | VarDecl Tree ident
     | FuncDecl Tree ident {Tree} Tree
     | FunCall ident {Tree}
     | Block {Tree}
     | ...
```

- Define abstract class `Tree` with concrete inner subclasses `ModDecl`, `VarDecl`, `FunDecl`, `FunCall`, etc.

- Define visitor interface with methods

```
caseModDecl(ModDecl tree)
caseVarDecl(VarDecl tree)
...
```

The Tree Class for JO

```
package j0c;
abstract class Tree {
   int pos; // common for all trees

   abstract void apply(Visitor v);

  static class ModDecl extends Tree {
     String id; Tree[] dcls
     ModDecl(int pos,
             String id, Tree[] dcls) {
        this.pos = pos;
        this.id = id;
        this.dcls = dcls;
     }
     void apply(Visitor v) {
        v.caseModDecl(this);
     }
   }

  static class VarDecl extends Tree {
     Tree type; String id;
     VarDecl(int pos,
             Tree type, String id) {
        this.pos = pos;
        this.type = type;
        this.id = id;
     }
     void apply(Visitor v) {
        v.caseVarDecl(this);
     }
  }
  ...
  interface Visitor {
     void caseModDecl(ModDecl tree);
     void caseVarDecl(VarDecl tree);
     ...
  }
}
```

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 number: its value as an **`int`**.
  - for a string: it's characters as a **`String`**.
- The **`pos`** field contains the current position of the tree, important for error messages. This field is common for all kinds of trees; thats why it is a member of class **`Tree`**.

Building the Tree

```
package j0c;
import calc.Tree.*;

class Parser extends Scanner {

   ...

  Tree whileStatement {
      int startPos = pos;
      nextSym();
      accept (LPAREN);
      Tree c = expression();
      accept(RPAREN);
      Tree s = statement();
      return
         new WhileStmt(startPos, c, s);
   }
```

```
Tree block {
   int startpos = pos;
   nextSym();
   stats = new TreeArrayBuffer();
   while (sym != EOF && sym != RBRACE)
   {
      stats.append(statement());
   }
   return
      new Block(startpos,
                 stats.toArray());
}
...
```

Helper class : TreeArrayBuffer

```
class TreeArrayBuffer {
    /** append an element to end of buffer */
    void append(Tree t);

    /** return current elements as an array.
     *  the length of the returned array matches
     *  exactly the number of elements in the buffer.
     */
    Tree[] toArray(t);
  }
```

Example Visitor : A Pretty Printer

```
package j0c;
import calc.Tree.*;

class Pretty implements Visitor {

  public void caseModDecl(ModDecl tree) {
    System.out.print("module " + tree.id + "{");
    for (int i = 0; i < tree.dcls.length; i++)
      System.out.println(); print(tree.dcls[i]);
    }
    System.out.println();
    System.out.print("}");
  }
  ...
  public void print() {
    tree.apply(this);
  }
}
```