

Winter 2007-2008 Compiler Construction T5 – AST

Mooly Sagiv and Roman Manevich
School of Computer Science
Tel-Aviv University

Today

ic
IC
Language

Lexical
Analysis

Syntax
Analysis
Parsing

AST

Symbol
Table
etc.

Inter.
Rep.
(IR)

Code
Generation

exe
Executable
code

- Today:
 - EBNF
 - AST construction
 - AST traversal
 - Visitor pattern
- Next week:
 - Annotating ASTs
 - Symbol tables
 - Type-checking

2

EBNF

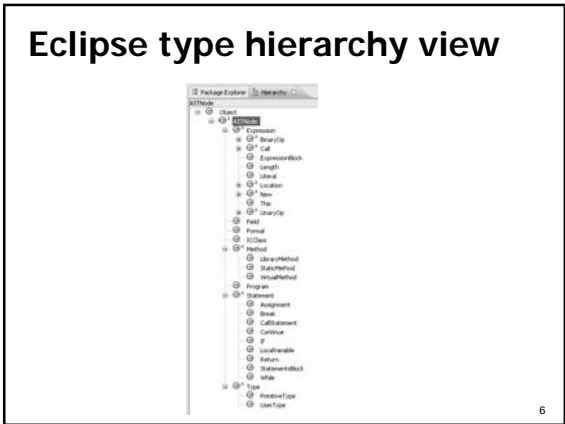
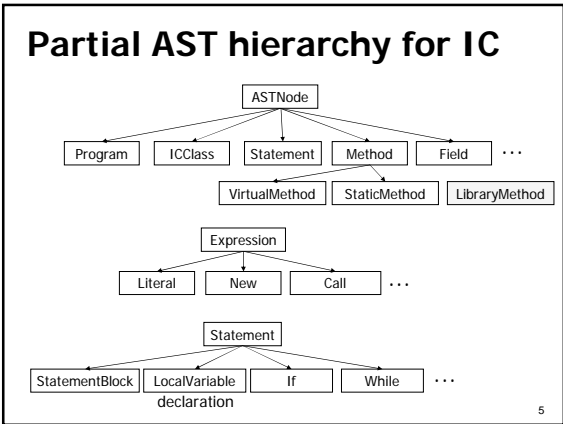
- Extended **B**ackus-**N**aur form
 - Extends BNF with regular expressions
 - R^* R^+ (ABC) $R?$ $[R]$
 - $[R]$ stands for optional part (same as $R?$)
- Not supported by CUP
- Translate EBNF rules to BNF
 - $list ::= x^*$
 $list ::= | x list$
 - $x ::= y [z]$
 $x ::= y | y z$

3

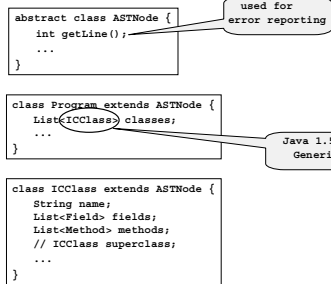
Abstract Syntax Trees

- Intermediate program representation
- Defines a tree
 - What is the root of the AST?
 - Preserves program hierarchy
- Node types defined by class hierarchy
- Generated by parser
- Keywords and punctuation symbols not stored (not relevant once parse tree exists)
- Provides clear interface to other compiler phases

4

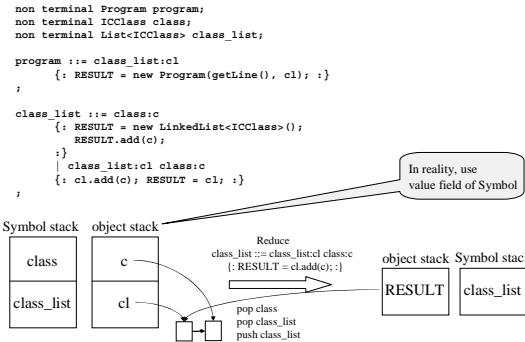


AST node contents



7

Actions part of IC.cup



8

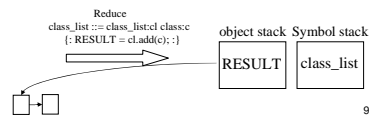
Actions part of IC.cup

```

non terminal Program program;
non terminal IClass class;
non terminal List<IClass> class_list;

program ::= class_list:cl
    { : RESULT = new Program(getLine(), cl); : }
;

class_list ::= class:c
    { : RESULT = new LinkedList<IClass>();
      RESULT.add(c);
    }
    | class_list:cl class:c
    { cl.add(c); RESULT = cl; : }
;
    
```



9

AST traversal

- Once AST stable want to operate on tree
 - AST traversal for type-checking
 - AST traversal for transformation (IR)
 - AST traversal for pretty-printing (-dump-ast)
- Each operation in separate *pass*

10

Non-Object Oriented approach

```

prettyPrint(ASTNode node) {
    if (node instanceof Program) {
        Program prog = (Program) node;
        for (IClass icc : prog.classes) {
            prettyPrint(icc);
        }
    }
    else if (node instanceof IClass) {
        IClass icc = (IClass) node;
        printClass(icc);
    }
    else if (node instanceof BinaryExpression) {
        BinaryExpression be = (BinaryExpression) node;
        prettyPrint(be.lhs);
        System.out.println(be.operator);
        prettyPrint(be.rhs);
    }
    ...
}
    
```

- Messy code
- `instanceof` + down-casting error-prone
- Not extensible

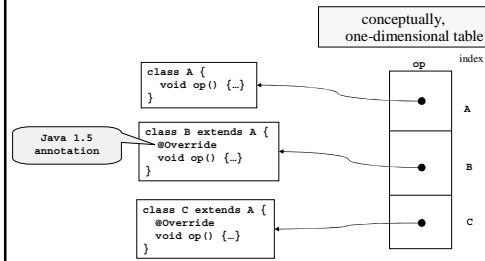
11

Visitor Pattern

- Separate operations on objects of a data structure from object representation
- Each operation (pass) may be implemented as separate visitor
- Use double-dispatch to find right method for object
- Instance of a design pattern
 - Recommended book
Design Patterns / Gang of Four

12

Single dispatch - polymorphism



13

What if we need more operations?

```
class A {
    void op1() {...}
    void op2() {...}
    void op3() {...}
}
```

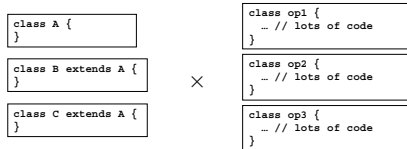
```
class B extends A {
    @Override
    void op1() {...}
    @Override
    void op2() {...}
    @Override
    void op3() {...}
}
```

```
class C extends A {
    @Override
    void op1() {...}
    @Override
    void op2() {...}
    @Override
    void op3() {...}
}
```

Want to separate complicated operations from data structures

14

What if we need more operations?



Problem: OO languages support only single-polymorphism.
We seem to need double-polymorphism

15

Visitor pattern in Java

```
class A {
    A x;
    accept(Visitor v) {
        v.visit(this);
    }
}
```

```
class B extends A {
    accept(Visitor v) {
        v.visit(this);
    }
}
```

```
class C extends A {
    accept(Visitor v) {
        v.visit(this);
    }
}
```

```
interface Visitor {
    visit(A a);
    visit(B b);
    visit(C c);
}
```

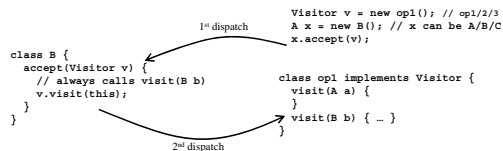
```
class op1 implements Visitor {
    visit(A a) {...}
    visit(B b) {...}
    visit(C c) {...}
}
```

```
class op2 implements Visitor {
    visit(A a) {...}
    visit(B b) {...}
    visit(C c) {...}
}
```

```
class op3 implements Visitor {
    visit(A a) {...}
    visit(B b) {...}
    visit(C c) {...}
}
```

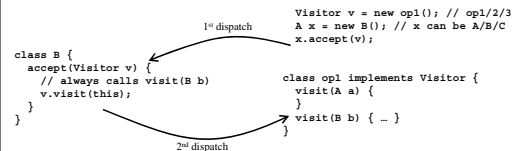
16

Double dispatch example



17

Double dispatch example



	op1	op2	op3
A			
B	op1.visit(B b)		
C			

Visitor pattern conceptually implements two-dimensional table

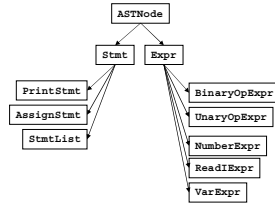
18

Straight Line Program example

```

prog → stmt_list
stmt_list → stmt
stmt_list → stmt_list stmt
stmt → var = expr;
stmt → print(expr);

expr → expr + expr
expr → expr - expr
expr → expr * expr
expr → expr / expr
expr → - expr
expr → ( expr )
expr → number
expr → read()
expr → var
    
```



(Code available on [web site](#).
Demonstrates scanning,
parsing, AST + visitors)

19

Printing visitor example

```

interface Visitor {
    void visit(StatList stmts);
    void visit(Stat stmt);
    void visit(PrintStat stmt);
    void visit(AssignStat stmt);
    void visit(ReadIExpr expr);
    void visit(VarExpr expr);
    void visit(BinaryOpExpr expr);
}

public class PrettyPrinter implements Visitor {
    public void print(ASTNode root) {
        root.accept(this);
    }

    public void visit(StatList stmts) {
        for (Stat s : stmts.statements) {
            s.accept(this);
            System.out.println();
        }
    }

    // x = 2*7
    public void visit(AssignStat stmt) {
        stmt.varExpr.accept(this);
        System.out.print("=");
        stmt.rhs.accept(this);
        System.out.print(";");
    }

    // x
    public void visit(VarExpr expr) {
        System.out.print(expr.name);
    }

    // 2*7
    public void visit(BinaryOpExpr expr) {
        expr.lhs.accept(this);
        System.out.print(expr.op);
        expr.rhs.accept(this);
    }
    ...
}
    
```

Java 1.5 enhanced for loop

20

Visitor variations

```

interface PropagatingVisitor {
    /** Visits a statement node with a given
     * context object (book-keeping)
     * and returns the result
     * of the computation on this node.
     */
    Object visit(Stmt st, Object context);
    Object visit(Expr e, Object context);
    Object visit(BinaryOpExpr e, Object context);
    ...
}
    
```

- Propagate values down the AST (and back)

21

Evaluating visitor example

```

public class SLPEvaluator implements PropagatingVisitor {
    public void evaluate(ASTNode root) {
        root.accept(this);
    }

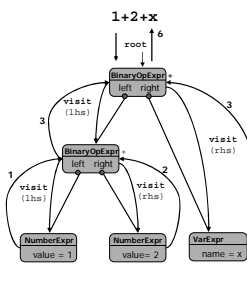
    /** x = 2*7
     */
    public Object visit(AssignStat stmt, Object env) {
        Expr rhs = stmt.rhs;
        Integer expressionValue = (Integer) rhs.accept(this, env);
        VarExpr var = stmt.varExpr;
        ((Environment)env).update(var, expressionValue);
        return null;
    }

    /** expressions like 2*7 and 2*y
     */
    public Object visit(BinaryOpExpr expr, Object env) {
        Integer lhsValue = (Integer) expr.lhs.accept(this, env);
        Integer rhsValue = (Integer) expr.rhs.accept(this, env);
        int result;
        switch (expr.op) {
            case PLUS:
                result = lhsValue.intValue() + rhsValue.intValue();
                ...
        }
        return new Integer(result);
    }
    ...
}
    
```

class Environment {
Integer get(VarExpr ve) {...}
void update(VarExpr ve, int value) {...}

22

AST traversal



```

class BinaryOpExpr extends Expression {
    Object accept(Visitor v) {
        return v.visit(this);
    }
    Expression lhs, rhs;
}
class NumberExpr extends Expression {
    Object accept(Visitor v) {
        return v.visit(this);
    }
    int val;
}

public class SLPEvaluator ... {
    public Object visit(BinaryOpExpr e, Object env) {
        Integer lhsValue = (Integer)e.lhs.accept(this, env);
        Integer rhsValue = (Integer)e.rhs.accept(this, env);
        int result;
        switch (expr.op) {
            case PLUS:
                result = lhsValue.intValue() + rhsValue.intValue();
                ...
        }
        return new Integer(result);
    }
    public Object visit(NumberExpr e, Object env) {
        return e.value;
    }
    public Object visit(VarExpr e, Object env) {
        return ((Environment)env).get(e);
    }
}
    
```

23

Visitor + Generics

```

interface PropagatingVisitor<DownType, UpType> {
    UpType visit(Stmt st, DownType d);
    UpType visit(Expr e, DownType d);
    UpType visit(VarExpr ve, DownType d);
    ...
}

public class SLPEvaluator implements
    PropagatingVisitor<Environment, Integer> {
    public Integer visit(VarExpr expr, Environment env) {
        return env.get(expr);
    }
    ...
}
    
```

24

See you next week

25