

**Compiler Construction**

**Abstract Syntax Tree**

**Ran Shaham and Ohad Shacham**  
**School of Computer Science**  
**Tel-Aviv University**

---

---

---

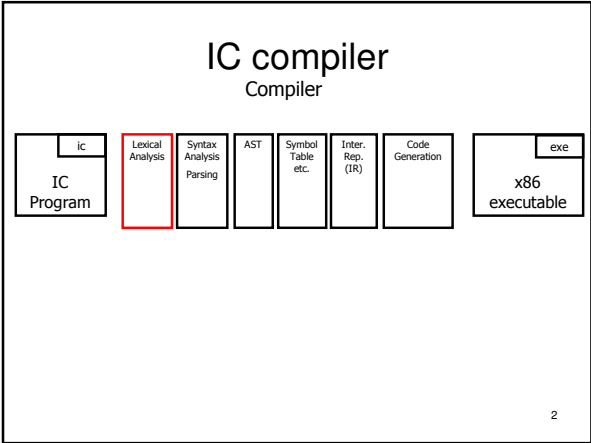
---

---

---

---

---



---

---

---

---

---

---

---

---

**Abstract Syntax Tree**

- More useful representation of syntax tree
  - Less clutter
  - Actual level of detail depends on your design
- Basis for semantic analysis
- Later annotated with various information
  - Type information
  - Computed values

3

---

---

---

---

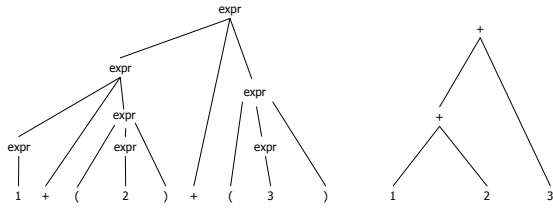
---

---

---

---

## Parse tree vs. AST



4

---

---

---

---

---

---

---

---

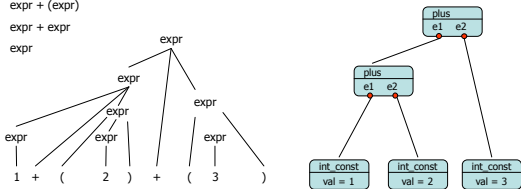
## AST construction

```

1 + (2) + (3)
expr + (2) + (3)
expr + (expr) + (3)
expr + expr + (3)
expr + (3)
expr + (expr)
expr + expr
expr
    
```

```

expr ::= expr:e1 PLUS expr:e2
      | LPAREN expr:e RPAREN
      | INT_CONST:i
      { RESULT = new plus(e1, e2); ; }
      { RESULT = e; ; }
      { RESULT = new int_const(..., i); ; }
    
```



5

---

---

---

---

---

---

---

---

## PA2

- Write parser for IC
- Write parser for `libc.sig`
- Check syntax
  - Emit either “Parsed [file] successfully!” or “Syntax error in [file]: [details]”
- `-print-ast` option
  - Prints one AST node per line

6

---

---

---

---

---

---

---

---

## PA2 – step 1

- Understand IC grammar in the manual
  - Don't touch the keyboard before understanding spec
- Write a debug JavaCup spec for IC grammar
  - A spec with "debug actions" : print-out debug messages to understand what's going on
- Try "debug grammar" on a number of test cases
- Keep a copy of "debug grammar" spec around

7

---

---

---

---

---

---

---

---

## PA2 – step 2

- Flesh out AST class hierarchy
  - Don't touch the keyboard before you understand the hierarchy
  - Keep in mind that this is the basis for later stages
- Web-site contains an AST adapted with permission from Tovi Almozlino
- Change CUP actions to construct AST nodes

8

---

---

---

---

---

---

---

---

## Abstract Syntax Trees

- Intermediate program representation
- Defines a tree
  - Preserves program hierarchy
- Node types defined by class hierarchy
- Generated by parser
- Keywords and punctuation symbols not stored
- Provides clear interface to other compiler phases

9

---

---

---

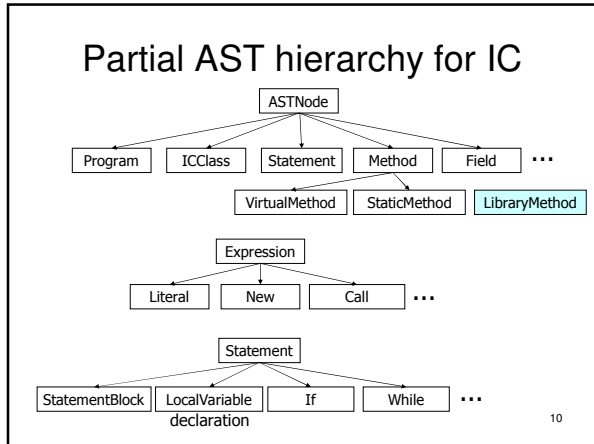
---

---

---

---

---




---

---

---

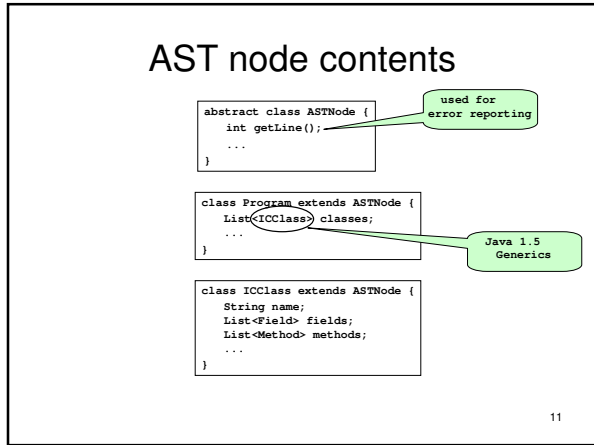
---

---

---

---

---




---

---

---

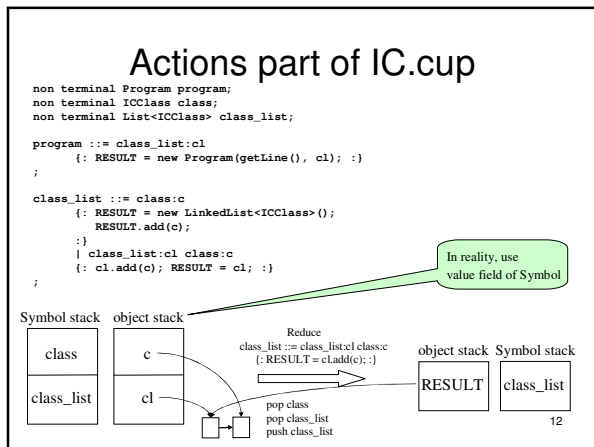
---

---

---

---

---




---

---

---

---

---

---

---

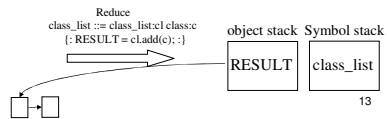
---

## Actions part of IC.cup

```
non terminal Program program;
non terminal ICClass class;
non terminal List<ICClass> class_list;

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

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



## 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*

14

## Non-Object Oriented approach

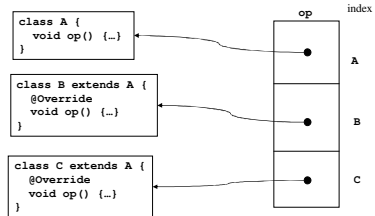
```
prettyPrint(ASTNode node) {
  if (node instanceof Program) {
    Program prog = (Program) node;
    for (ICClass icc : prog.classes) {
      prettyPrint(icc);
    }
  }
  else if (node instanceof ICClass) {
    ICClass icc = (ICClass) node;
    printClass(icc);
  }
  else if (node instanceof BinaryExpression)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

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

15

## Single dispatch - polymorphism

conceptually,  
one-dimensional table



16

---

---

---

---

---

---

---

---

## 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

17

---

---

---

---

---

---

---

---

## What if we need more operations?

```
class A {
}
class B extends A {
}
class C extends A {
}
```

×

```
class op1 extends op {
  - // lots of code
}
class op2 extends op {
  - // lots of code
}
class op3 extends op {
  - // lots of code
}
```

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

18

---

---

---

---

---

---

---

---

## What if we need more operations?

```
class A {  
}  
  
class B extends A {  
}  
  
class C extends A {  
}
```

```
class op1 extends op(  
doOp(A a) {  
- // lots of code  
}  
doOp(B b) {  
- // lots of code  
}
```

```
class op2 extends op(  
doOp(A a) {  
- // lots of code  
}  
doOp(B b) {  
- // lots of code  
}
```

```
class op3 extends op(  
doOp(A a) {  
- // lots of code  
}  
doOp(B b) {  
- // lots of code  
}
```

Overloading is static

---

---

---

---

---

---

---

---

## 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](#)

---

---

---

---

---

---

---

---

## 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);  
}  
}
```

X

```
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) {...}  
}
```

---

---

---

---

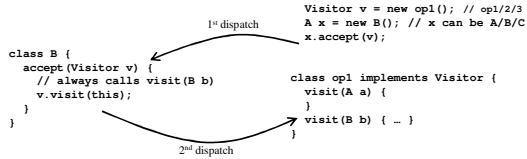
---

---

---

---

## Double dispatch example



22

---

---

---

---

---

---

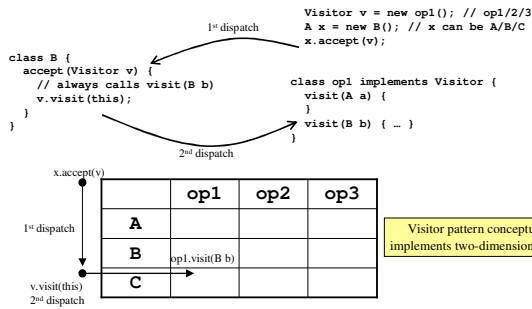
---

---

---

---

## Double dispatch example



23

---

---

---

---

---

---

---

---

---

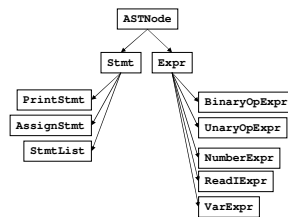
---

## 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)

24

---

---

---

---

---

---

---

---

---

---

## Printing visitor example

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

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

    // x = 2*7
    public void visit(AssignStmt 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);
    }
    ...
}
```

```
interface Visitor {
    void visit(StmtList stmts);
    void visit(Stmt stmt);
    void visit(PrintStmt stmt);
    void visit(AssignStmt stmt);
    void visit(Expr expr);
    void visit(ReadExpr expr);
    void visit(VarExpr expr);
    void visit(NumberExpr expr);
    void visit(UnaryOpExpr expr);
    void visit(BinaryOpExpr expr);
}
```

25

## 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)

26

## Evaluating visitor example

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

    /** x = 2*7
     */
    public Object visit(AssignStmt 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) [-]
}
```

27

# 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);
    }
}
    
```

SLPEvaluator ev = new SLPEvaluator();  
 Integer result = (Integer)root.accept(ev);

---

---

---

---

---

---

---

---

# Visitor + Generics

```

interface PropagatingVisitor<DownType,UpType> {
    UpType visit(Stmnt 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);
    }
    ...
}
    
```

---

---

---

---

---

---

---

---