

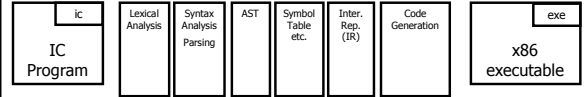
Compiler Construction

Abstract Syntax Tree

Rina Zviel-Girshin and Ohad Shacham
School of Computer Science
Tel-Aviv University

IC compiler

Compiler



2

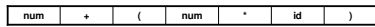
From text to abstract syntax

program text

5 + (7 * x)



token stream

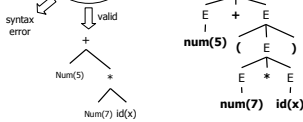


Grammar:

- E → id
- E → num
- E → E + E
- E → E * E
- E → (E)



Abstract syntax tree



3

PA2

```

program ::= classDecl*
classDecl ::= class CLASS [extends CLASS] '{ (field|method) }'
field ::= type ID (',' ID)* ';'
method ::= [static] (type|void) ID '{ [formals] }' '{ stmt* }'
    
```

```

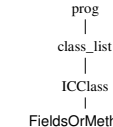
FieldsOrMethods ::=
    |
    Field FieldsOrMethods |
    Method FieldsOrMethods
    
```

```

ICClass ::= CLASS CLASS_ID EXTENDS CLASS_ID LCBR FieldsOrMethods RCBR |
    CLASS CLASS_ID LCBR FieldsOrMethods RCBR
    
```

4

PA2

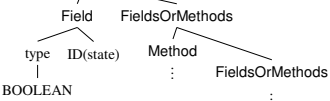


```

FieldsOrMethods ::=
    |
    Field FieldsOrMethods |
    Method FieldsOrMethods
    
```

```

ICClass ::= CLASS CLASS_ID EXTENDS CLASS_ID LCBR FieldsOrMethods RCBR |
    CLASS CLASS_ID LCBR FieldsOrMethods RCBR
    
```



5

AST

```

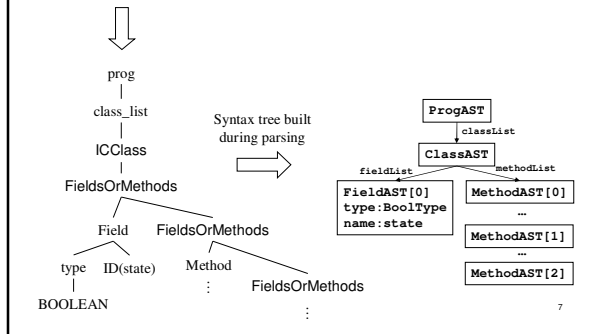
program ::= classDecl*
classDecl ::= class CLASS [extends CLASS] '{ (field|method) }'
field ::= type ID (',' ID)* ';'
method ::= [static] (type|void) ID '{ [formals] }' '{ stmt* }'
    
```

```

FieldsOrMethods ::= {; RESULT = new FieldsMethods(); ;} |
    Field:field FieldsOrMethods:next |
    {; RESULT = next;
    RESULT.addField(field); ;} |
    Method:method FieldsOrMethods:next
    {; RESULT = next;
    RESULT.addMethod(method); ;}
    
```

6

Parsing and AST



Designing an AST

```

public abstract class ASTNode {
    // common AST nodes functionality
}

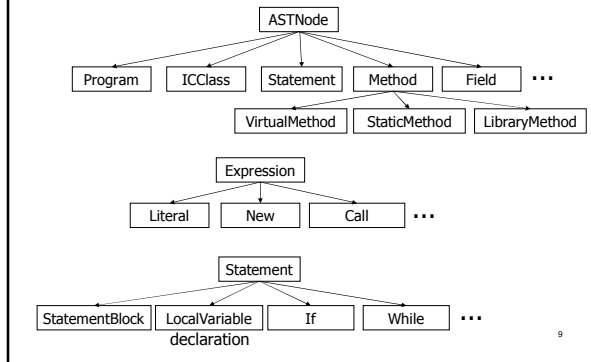
public class Expr extends ASTNode {
    private int value;
    private Expr left;
    private Expr right;
    private String operator;

    public Expr(Integer val) {
        value = val.intValue();
    }

    public Expr(Expr operand, String op) {
        this.left = operand;
        this.operator = op;
    }

    public Expr(Expr left, Expr right, String op) {
        this.left = left;
        this.right = right;
        this.operator = op;
    }
}
    
```

Partial AST hierarchy for IC



AST node contents

```

abstract class ASTNode {
    int getLine();
    ...
}

class Program extends ASTNode {
    List<ICClass> classes;
    ...
}

class ICClass extends ASTNode {
    String name;
    List<Field> fields;
    List<Method> methods;
    ...
}
    
```

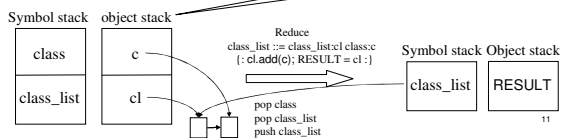
Actions part of IC.cup

```

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*

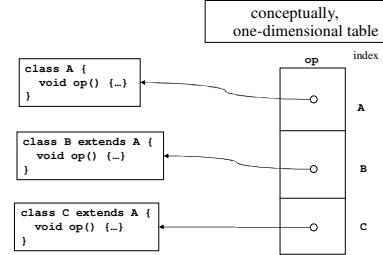
Non-Object Oriented approach

```
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

13

Single dispatch - polymorphism



14

What if we need more operations?

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

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

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

Want to separate complicated operations from data structures

15

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
}
```

16

What if we need more operations?

```
class A {
}
```

```
class B {
}
```

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

Overloading is static

```
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
    }
}
```

```
op o = new op3();
A a = new B();
o.doOp(a);
```

17

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

18

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) {...}
}
    
```

19

Double dispatch example

```

class B {
    accept(Visitor v) {
        // always calls visit(B b)
        v.visit(this);
    }
}

class op1 implements Visitor {
    visit(A a) {
        // 1st dispatch
        Visitor v = new op1(); // op1/2/3
        A x = new B(); // x can be A/B/C
        x.accept(v);
    }
}

class op2 implements Visitor {
    visit(A a) {
        // 2nd dispatch
        visit(B b) { ... }
    }
}
    
```

20

Double dispatch example

```

class B {
    accept(Visitor v) {
        // always calls visit(B b)
        v.visit(this);
    }
}

class op1 implements Visitor {
    visit(A a) {
        // 1st dispatch
        Visitor v = new op1(); // op1/2/3
        A x = new B(); // x can be A/B/C
        x.accept(v);
    }
}

class op2 implements Visitor {
    visit(A a) {
        // 2nd dispatch
        visit(B b) { ... }
    }
}

class op3 implements Visitor {
    visit(A a) {
        // 2nd dispatch
        visit(B b) { ... }
    }
}
    
```

x.accept(v)	op1	op2	op3	Visitor pattern conceptually implements two-dimensional table
A				
B				
C				

1st dispatch: v.visit(this) → op1.visit(B b)

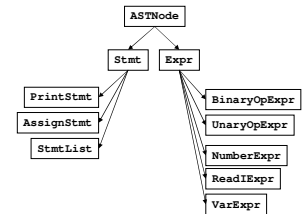
21

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 → readi()
expr → var
    
```



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

22

Printing visitor example

```

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

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);
    }
    ...
}
    
```

23

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)

24

Evaluating visitor example

```

public class SLPEvaluator 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);
}
...
}

```

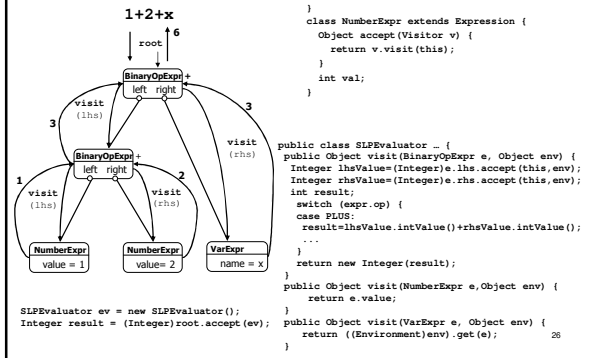
```

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

```

25

AST traversal



Error recovery

- How to catch errors
 - public void syntax_error(Symbol cur_token)
- Optional error recovery
 - Use error token in your grammar
 - stmt ::= expr SEMI
 - | while_stmt SEMI
 - | if_stmt SEMI
 - | ...
 - | error SEMI ;