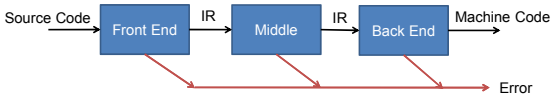
CS 1622: Abstract Syntax & Semantic Analysis

Jonathan Misurda
jmisurda@cs.pitt.edu

Three-pass Compiler



Passes: number of times through a program representation
- 1-pass, 2-pass, multi-pass compilation
- Language becomes more complex → more passes

Phases: conceptual and sometimes physical stages
- Symbol table coordinates information between phases
- Phases are not completely separate
- Semantic phase may do things that syntax phase should do
- Interaction is possible

Data Structures for Compilation

Abstract Syntax Tree
- Stores the information from the parse and lexing phases
- Walk the tree to produce IR or target code

Symbol Table
- Collect and maintain information about identifiers
  - Attributes: type, address, scope, size
- Used by most compiler passes and phases
  - Some phases add information:
    - lexing, parsing, semantic analysis
  - Some phases use information:
    - Semantic analysis, code optimization, code generation
- Debuggers also can make use of a symbol table
  - `gcc -g` keeps a version of the symbol table in the object code

Building the AST

Since our parser is a bottom-up parser, we will need to build a tree from the leaves upwards.

The leaves of the tree will be our terminals and the parser actions will allow us to combine them into subtrees.

We may omit some terminals such as parentheses due to the hierarchy of the tree containing the same information for expressions.

Our non-terminals will likely be represented as roots of subtrees, and ultimately we'll have a root node representing a whole program, class, function, etc.

Building the AST

```
abstract class Node {
        Node parent;
        ArrayList<Node> children;
        Node() { children = new ArrayList<Node>(); }
}

class IntNode extends Node {
        Integer myInt;
        public IntNode(Integer i) { myInt = i; }
}

class AddNode extends Node {
        public AddNode(Node l, Node r) {
            children.add(l);
            children.add(r);
        }
}
```

Building the AST

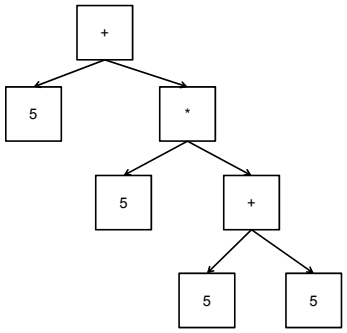
```
non terminal ArrayList<Node> line_list;
non terminal Node line;
non terminal Node expr;

line_list ::= line_list:list line:l
        {: if(list == null) list = new ArrayList<Node>();
           list.add(l); RESULT = list;
        :}
        | /* epsilon */
        ;

line ::= expr:result EOL
        {: RESULT = result; :}
        |
        expr:result
        {: RESULT = result; :}
        ;
```

Building the AST

```
expr ::=expr:l PLUS expr:r
        {: RESULT = new AddNode(l, r); :}
        |
        expr:l MINUS expr:r
        {: RESULT = new MinusNode(l, r); :}
        |
        expr:l DIV expr:r
        {: RESULT = new DivNode(l, r); :}
        |
        expr:l TIMES expr:r
        {: RESULT = new TimesNode(l, r); :}
        |
        LPAREN expr:e RPAREN
        {: RESULT = e; :}
        |
        INT:i
        {: RESULT = new IntNode(i); :}
        ;
```

AST

For 5 + 5 * (5 + 5):



Walking the AST

```
abstract class Node {
    …
    public abstract int visit();
}

class IntNode extends Node {
    Integer myInt;

    public IntNode(Integer i) {
        myInt = i;
    }

    public int visit() {
        System.out.print(myInt);
        return myInt.intValue();
    }
}
```

Walking the AST

```
class AddNode extends Node {
    public AddNode(Node l, Node r) {
        children.add(l);
        children.add(r);
    }

    public int visit() {
        int l = children.get(0).visit();
        System.out.print(" + ");
        int r = children.get(1).visit();

        return l + r;
    }
}
```

Driver

```
Symbol parse_tree = null;
try {
        ExprParser parser_obj = new ExprParser(
                new ExprLex(new FileInputStream(args[0])));
        parse_tree = parser_obj.parse();

        ArrayList a = ((ArrayList)parse_tree.value);

        for(Iterator iter=a.iterator(); iter.hasNext(); )
        {
                Node root = (Node)iter.next();
                int result = root.visit();
                System.out.println(" = " + result);
        }
}
```

Pretty Printing

java Calc test.txt

```
3 + 4 = 7
3 * 4 – 2 = 10
3 + 2 * -2 = -10
```

The result of the calculation is correct but we lost the parenthesis because we made an AST (as opposed to a concrete syntax tree). How do we fix this?

Two possibilities:

1. Print parenthesis around all of our children subtrees as we visit them
2. Print parenthesis around children when their operator precedence is lower than the current node

Precedence

```
class TimesNode extends Node {
    …
    public int visit() {
        Node leftChild = children.get(0);
        Node rightChild = children.get(1);
        int l = 0; int r = 0;

        if(leftChild instanceof AddNode
                || leftChild instanceof MinusNode) {
            System.out.print("( ");
            l = leftChild.visit();
            System.out.print(" )");
        } else {
             l = leftChild.visit();
        }
        System.out.print(" * ");
        …
```

Success!

java Calc test.txt

```
3 + 4 = 7
3 * 4 - 2 = 10
( 3 + 2 ) * -2 = -10
```

Design Patterns

A **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design.

For instance, we may only want to have at most a single instance of an object instantiated.

We can use a *Singleton Pattern*. Implemented in Java as:

```
public class Singleton {
       private Singleton() { }
       private static Singleton myInstance;
       public static Singleton getInstance() {
              if(myInstance == null)
                     myInstance = new Singleton();
              return myInstance;
       }
}
```

Visitor Pattern

The **visitor design pattern** is a way of separating an algorithm from an data structure on which it operates.

We have an AST – a tree data structure – that we will want to visit to do various things, such as type checking, code optimization, code generation, etc.

The visitor pattern allows us to move each tree algorithm into a single class, and use double invocation to traverse our data structure.

The visitor object is passed to an `accept()` method in each node. The `accept()` method then calls the appropriate method in the visitor for the type of node that it is.

By passing different visitors around to the accept methods, we can do different work on the same structure without changing each node.

Visitor Pattern

```
public interface Visitor {
       public int visit(AddNode n);
       public int visit(MinusNode n);
       public int visit(TimesNode n);
       public int visit(DivNode n);
       public int visit(IntNode n);
}

class IntNode extends Node {
       …
       public int accept(Visitor v) {
              return v.visit(this);
       }
}
```

Separation of Concerns

```
public class Interpreter implements Visitor {
       public int visit(AddNode n) {
              Node lhs = n.children.get(0);
              Node rhs = n.children.get(1);
              int l; int r;
              l = lhs.accept(this);
              r = rhs.accept(this);
              return l+r;
       }

       public int visit(IntNode n) {
              return n.myInt.intValue();
       }

       …
```

Separation of Concerns

```
public class PrettyPrinter implements Visitor {
        public int visit(AddNode n) {
                Node lhs = n.children.get(0);
                Node rhs = n.children.get(1);

                lhs.accept(this);
                System.out.print( " + " );
                rhs.accept(this);

                return 0;
        }

        public int visit(IntNode n) {
                System.out.print(n.myInt.intValue());
                return 0;
        }

        …
```

Driver

```
Symbol parse_tree = null;
try {
        ExprParser parser_obj = new ExprParser(
                new ExprLex(new FileInputStream(args[0])));
        parse_tree = parser_obj.parse();

        ArrayList a = ((ArrayList)parse_tree.value);
        Visitor v = new Interpreter();

        for(Iterator iter=a.iterator(); iter.hasNext(); )
        {
                Node root = (Node)iter.next();
                int result = root.visit();
                System.out.println(" = " + result);
                System.out.println("Visitor:");
                result = root.accept(v);
                System.out.println(" = " + result);
```

Both Implementations

java Calc test.txt

```
3 + 4 = 7
Visitor:
3 + 4 = 7
----------------------------------------
3 * 4 - 2 = 10
Visitor:
3 * 4 - 2 = 10
----------------------------------------
( 3 + 2 ) * -2 = -10
Visitor:
( 3 + 2 ) * -2 = -10
----------------------------------------
```