Abstract Syntax

Part IV

Syntax Trees

- In a multi-pass compiler the parser builds a syntax tree representing the program to compile.
- Later phases then work on this *abstract syntax tree (AST)*.
- The tree could be the concrete syntax tree (the parse tree) corresponding to the grammar, but this is often not very practical.

A Note on the assignment

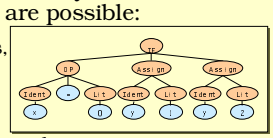
- The skeleton for the parser that you have been given (expressive.cup) does not generate an AST.
- It just generates a string with the parsed constructs. (On success you get the input program back as a string.)
- This is useful for testing the first part of your parser, but it is not useful for the rest of the compiler.
- You will have to replace this with a parser that generates an AST.

Abstract vs. Concrete Syntax

- Compared to the concrete syntax tree, some simplifications are possible:

– No need for parentheses, *A* * (*B* + *C*) becomes:

```
OP(MULT, A,
   OP(PLUS, B, C))
```



– **No need to maintain terminals,**

`if(x = 0){y:=1.} else {y:=2.}.` becomes:

```
IF(OP(EQ,Ident("x"),Lit(0)),
   [Assign(Ident("y"),Lit(1))],
   [Assign(Ident("y"),Lit(2))])
```


Abstract Syntax Tree

- An AST is a tree with one kind of node for each alternative in the abstract syntax.
- It is simpler than a parse tree and hence easier to use.
- Still it has all the information that is necessary for the compilation.

Abstract Syntax Tree

- We represent ASTs using a set of Java classes, one for each alternative.
- We use one common abstract superclass: `Tree`.
- Each class represents subtrees as instance variables.
- Each class has a constructor to construct a node of the given kind.

Example Abstract Syntax

```
Expr ::= OP Op Expr Expr
       | Lit int
       | Ident String
Op ::= MULT | PLUS | EQ …
```


Example AST

```
public abstract class Tree {
  public static class Lit extends Tree {
    int num;
    public Lit(int num) {
      this.num = num;
    }
  }
  public static class OP extends Tree {
    Tree left, right;
    int Op;
    public OP(int Op, Tree left, Tree right) {
      this.Op = Op;
      this.right = right;
      this.left = left;
    }
  }
}
```


Repetition

- Repetition in the grammar is usually implemented by arrays.

```
Program → Statements
Statements → Statement '.' Statements
   | ε
```

```
public static abstract class Stat extends Tree {
  …
}
public static class Prog extends Tree {
  public final Stat[] body;
  …
}
```


Designing an AST

- The AST will be used by most phases of the compiler.
- Constructs in the language that can be handled in the same or in similar ways should be grouped as one node-type.
- One has to find the balance between a deep and a broad AST.
  - A deep AST has many node-types.
  - A broad AST has many instances of a few node-types.

Deep vs. Broad

Deep

```
Expr - CompOp: COp E E
     |          |- <
     |          |- =
     |
     |- ArithOp - BinA: Op E E
     |          |       |- +
     |          |       |- -
     |          |
     |          |- Una: Op E E
     |                  |- -
     |- Lit - Int: I
     |      |- Bool: Val
     |                 | - TRUE
     |                 | - FALSE
     |- ID: S
```

Broad

```
Expr: Type Op E E S I
    |     |- <
    |     |- =
    |     |- +
    |     |- -
    |
    |- BINOP
    |- UNOP
    |- ID
    |- LIT
```


Finding The Balance

- The design of the AST depends on how the AST is going to be used in later phases of the compiler.
- Getting it right takes experience and usually some experimentation.
- (For the project you will be given a complete AST.)

Constructing ASTs

- Each syntax tree recognized by the parser should somehow be turned into an AST.
- This mapping is not always one to one. Some parse trees has to be passed around in the parser before they are inserted into the AST.

Example

```
Program → Statements
Statements → Statement '.' Statements | ε
```

- In the grammar and the parser we have `Statements` as a node in the parse tree. But in the AST statements will just be an "`statement array`" in the `Prog` node.
- The parser can collect all `Statements` in a list and then convert the list to an array and insert it in the `Prog` node.

Constructing an AST in JavaCUP

- In JavaCUP, just as in JLex, there is an action for each rule in the grammar.
  (Actually there can be several actions for a rule, one after each symbol on the rhs)
- The action is a piece of Java code, usually used for building and returning the AST.
- The action could be used for other things, for example to implement a simple interpreter (more on this later).

Actions in JavaCup

- Actions are written in *code strings* surrounded by delimiters of the form `{:` and `:}`.
- The action for a lhs symbol S, can return values to the places where S is used on the rhs. This is done by assigning the value to `RESULT`.
- Symbols on the rhs can be labeled in order to refer to their values in the action. This is done as `sym:lab`.

Example

The Grammar:

```
Expr → Expr + Expr
     | Integer | Identifier
```

could be recognized by the rule:

```
expr ::= expr PLUS expr
     |  INT
     |  ID
     ;
```

Adding actions to generate the AST we get:

```
expr ::= expr:e1 PLUS expr:e2
         {: RESULT = new Op(Tokens.PLUS, e1, e2); :}
       | INT:i {: RESULT = new Lit(i.intValue()); :}
       | ID:id {: RESULT = new Ident(id); :}
;
```


Types

- We have to give the types of the values returned by the symbols used in the parser:

```
terminal String ID;
terminal Integer INT;
non terminal Expr expr;
```

Errors & Positions

- In order to give precise error messages we have to keep track of where in the program file a certain construct was located at.
- Since we need to be able to give error messages during the semantic analysis we need to store the positions of some tokens in the AST.

Adding pos to the AST

```
public abstract class Tree {
  public final int pos;
  public Tree(int pos) { this.pos = pos; }

  public static class Lit extends Tree {
    int num;
    public Lit(int pos, int num) {
      super(pos);
      this.num = num;
    }
  }
  …
}
```


Positions in JavaCUP

- JavaCUP uses Symbol objects for tokens and non terminals while parsing. Symbol objects has two integer fields called left and right that can be used to store the position of a token.
- In the parser these values can be accessed by appending the words *left* or *right* to the label.
- (With the given scanner and Position class, the line and column is encoded in the value for right.)

Adding positions to the example

```
expr ::=
  expr:e1 PLUS:t expr:e2
    {:
     RESULT= new Op(tright, Tokens.PLUS, e1, e2);
    :}
  | INT:i
    {:
     RESULT = new Lit(iright, i.intValue());
    :}
  | ID:id {: RESULT = new Ident(idright, id); :}
  ;
```


A “complete” parser

```
package expressive;
import expressive.Tree.*;
terminal PLUS;
terminal String ID;
terminal Integer INT;
non terminal Expr expr;
start with expr;
expr ::=
  expr:e1 PLUS:t expr:e2
  {: RESULT= new Op(tright, Tokens.PLUS, e1, e2); :}
  | INT:i {: RESULT = new Lit(iright, i.intValue()); :}
  | ID:id {: RESULT = new Ident(idright, id); :}
  ;
```


Summary

- The abstract syntax should describe the programming language concisely without unnecessary syntactical constructs.
- The parser should create an abstract syntax tree while parsing the input.
- The AST should be designed so that the following phases can use it easily.
- The AST need information about source code positions for error handling.

Using Trees

- The AST will be used by later phases of the compiler.
- How do tree processors (compiler phases) access the tree?
  - Simple solution: use **instanceof** to find the kind of a tree node and cast
    `if (tree instanceof Lit) { return ((Lit) tree).num;}`
  - Better solution: object-oriented decomposition.
  - Even better solution: Visitors.

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()`.

Example: Expressions

- Two kinds of nodes: `Op`, `Lit`.
- Two kind of actions: `eval`, `print`.

```
public abstract class Tree {
  public static class Lit extends Tree {
    int num;
    public Lit(int num) {this.num = num;}
  }
 public static class Op extends Tree {
    Tree left, right; int Op;
    public OP(int Op, Tree left, Tree right) {
      this.Op = Op; this.right = right; this.left = left;
    }
  }
}
```


OO Decomposition

- Define methods `eval()` and `print()` in classes `Lit` and `Op`.
- The methods `eval()` and `print()` are abstract in `Tree`, so they can be invoked on every tree.
- What they do depend on the concrete kind of tree.

OO Decomposition

```
public abstract class Tree {
  public abstract void print();
  public abstract int eval();
  public static class Lit extends Tree {
    int num;
    public Lit(int num) {this.num = num;}
    public void print() {
        system.out.print(" " + num);
    }
    public int eval() { return num; }
  }
…
```


OO Decomposition

```
…
public static class Op extends Tree {
    Tree left, right; int Op;
    public OP(int Op, Tree left, Tree right) {
      this.Op= Op; this.right= right; this.left= left;}
    public void print() {
      System.out.print("(");
      left.print();
      printOp(Op);
      right.print();
      System.out.print(")");
  }
…
```

OO Decomposition

```
…
    public int eval() {
      int l = left.eval();
      int r = right.eval();
      switch(Op) {
        case TOKENS.PLUS: return l + r;
        case TOKENS.MINUS: return l - r;
        case TOKENS.TIMES: return l * r;
        case TOKENS.DIV: return l / r;
        default: throw new InternalError();
      }
    }
  }
}
```


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 is more common?
  - New nodes ~ changes in the language.
  - New methods ~ added optimizations.
- Which is easier to do?
  - New nodes: add a new subclass.
  - New methods: add processor method 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 – it is easy to share common code and data.
- A visitor object contains, for each kind of node N, a method called caseN that can process trees of that kind.
- The tree contains only a simple generic processor method which applies a given visitor object.

Example of Visitable Tree

```
public abstract class Tree {
  public abstract void apply(Visitor v);
  public static class Lit extends Tree {
    int num;
    public Lit(int num) {this.num = num;}
    public void apply(Visitor v) { v.caseLit(this); }
  }
  public static class Op extends Tree {
    Tree left, right; int Op;
    public OP(int Op, Tree left, Tree right) {
      this.Op = Op; this.right = right; this.left = left;}
    public void apply(Visitor v) { v.caseOp(this); }
  }
}
```


The Visitor

```
public interface Visitor {
  void caseLit(Lit tree);
  void caseOp(Op tree);
}
```


A Print visitor

```
public class Printer implements Visitor {
  public static void print(Tree tree) {
    tree.apply(new Printer());
  }
  public void caseLit(Tree.Lit tree) {
      System.out.print(" " + tree.num);
  }
  public void caseOP(Tree.Op tree) {
      System.out.print("(");
      print(tree.left);
      printOp(Op);
      print(tree.right);
      System.out.print(")");
  }
}
```

Coding with Visitors

- Make the tree ready for visits:
  - Write an `apply` method for each node.
  - Write an interface declaration for the visitor.
- Writing visitors:
  - Write the caseN method for each node type N.
  - Write one convenience routine, which can be called from the outside and for recursion. (in the example `print`).

An Evaluation Visitor

- The apply method has the signature `void apply(Visitor)` so we can not return the result of the evaluation to the caller of `apply`.
- Instead we keep the result in a local variable.

An Evaluation Visitor

```
public class Evaluator implements Visitor {
  int val;
  public static int eval(Tree tree) {
    Evaluator ev = new Evaluator();
    tree.apply(ev);
    return ev.val;
  }
  public void caseLit(Tree.Lit tree) {
      val = tree.num;
  }
…
```


An Evaluation Visitor

```
…
  public void caseOP(Tree.Op tree) {
    int lval = eval(tree.left)
    int rval = eval(tree.right);
    switch(tree.Op) {
      case TOKENS.PLUS:  val = lval + rval; break;
      case TOKENS.MINUS: val = lval - rval; break;
      case TOKENS.TIMES: val = lval * rval; break;
      case TOKENS.DIV:   val = lval / rval; break;
      default: throw new InternalError();
    }
  }
}
```


Which solution is better?

- Extensibility
  - OO Decomposition makes adding new kinds of nodes easy.
  - Visitors makes adding of new processors easy.
- Modularity
  - OO allows sharing of data and code in a tree node between phases.
  - Visitors allows sharing of data and code between methods of the same processor.
- Programming in a project:
  - Is one person responsible for one kind of node or is one person responsible for one tree processor?

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

Extensibility

- Compiler
  - Operations
    - Type-check
    - Pentium back-end
    - SPARC back-end
    - optimize
    - ...
  - Kinds
    - Id
    - Lit
    - Exprs
    - Statements
- GUI
  - Operations
    - redisplay
    - move
    - iconize
    - highlight
  - Kinds
    - Scrollbar
    - Menu
    - Canavas
    - Dialogbox
    - ...

Optimization: Reusing the Visitor

- Creating a new visitor object for every invocation is expensive.
- One routine is globally available and creates a new visitor.
- Another is local and reuses the visitor.
- More efficient and allows visitor global data.

Reusing the Visitor

```
public class Printer implements Visitor {
  public static void print(Tree tree){
    tree.apply(new Printer()); }
  protected void printRec(Tree tree) { tree.apply(this);}
  public void caseLit(Tree.Lit tree) {
    System.out.print(" " + tree.num); }
  public void caseOP(Tree.Op tree) {
    System.out.print("("); printRec(tree.left);
                           printOp(Op);
                           printRec(tree.right);
    System.out.print(")");
  }
}
```


Visitor Global Data

```
public class Printer implements Visitor {
  PrintStream p;
  public Printer(PrintStream p) {this.p = p;}

  public static void print(Tree tree, PrintStream p) {
    tree.apply(new Printer(p)); }
  protected void printRec(Tree tree){ tree.apply(this);}

  public void caseLit(Tree.Lit tree){p.print(" "+tree.num);}
  public void caseOP(Tree.Op tree) {
    p.print("("); printRec(tree.left);
                  printOp(Op,p);
                  printRec(tree.right); p.print(")");
  }
}
```


Summary

- In an OO setting there are basically two ways of encoding operations on a tree:
  - Object-oriented Decomposition.
  - Visitors.
- For compilers visitors are the better choice.