Abstract Grammar and Abstract Syntax Tree

In programming languages, we prefer a grammar that is close to the language constructs. We call this grammar the abstract grammar. The corresponding derivation tree is called the abstract syntax tree. A node represents a language construct, a leaf represents a token.

$$
\begin{aligned}
E &\rightarrow E + E \\
E &\rightarrow E - E \\
E &\rightarrow num \\
E &\rightarrow id
\end{aligned}
$$

The grammar is not parser friendly.

Concrete Grammar and Parse Tree

Therefore, the grammar is tranformed such that it becomes easy to parse. Sample transformations include removing left recursion, left factoring, etc.

The resulting grammar is called the concrete grammar. The corresponding derivation tree is called the parse tree.

Parse tree: each leaf represents a token, each internal node represents a production rule. It is determined by the grammar.

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +E \mid -E \mid \varepsilon \\
T &\rightarrow num \\
T &\rightarrow id
\end{aligned}
$$

More particularly: we use concrete grammar to construct parsers, but we insert code into parser implementation to explicitly generate abstract syntax tree.

Recall: Representing Abstract Syntax Tree

One abstract class for each nonterminal in the abstract grammar; one sub-class for each production rule.

```
public abstract class Exp {
}
public class PlusExp extends Exp {
   private Exp e1,e2;
   public PlusExp(Exp a1, Exp a2) {
     e1=a1; e2=a2;
   }
 }
public class MinusExp extends Exp {
   private Exp e1,e2;
   public MinusExp(Exp a1, Exp a2) {
     e1=a1; e2=a2;
   }
}
```

```
public class IdExp extends Exp {
   private String f0;
   public IdExp(String n0) {
     f0 = n0;
   }
}

public class IntExp extends Exp {
   private int v;
   public IntExp(int n0) {
      v = n0;
   }
}
```

Abstract Syntax Tree Construction

Assume a recursive descendent parser.

```
Exp parseE() {
   Exp e1=parseT();
   Exp e2=parseEP(e1);
   return e2;
}
Exp parseT() {
   if (next_token()==ID) {
      return new IdExp (eatID());
   } else if (next_token()==NUM) {
      return new NumExp(eatNum());
   } else error();
}
```

```
Exp parseEP(Exp e1) {
  Exp e2;
  if (next_token()=='+') {
     e2=parseE();
     return new PlusExp(e1,e2);
  } else if (next_token()=='-') {
     e2=parseE();
     return new MinusExp(e1,e2);
  } else if (next_token()=='$') {
     return e1;
  } else error();
}
```

Traversing AST:Design One

Nodes are visited in a depth-first order, starting from the root (calling evalVisit()). The behavior of each node varies depending on the node type.

```
 public abstract class Exp {
   public abstract int evalVisit();
}
public class PlusExp extends Exp {
   private Exp e1,e2;
   public PlusExp(Exp a1, Exp a2) {...}
   int evalVisit() {
      return e1.evalVisit() +
             e2.evalVisit();
   }
 }
public class MinusExp extends Exp {
   private Exp e1,e2;
   public MinusExp(Exp a1, Exp a2) {...}
   int evalVisit() {
     return e1.evalVisit()
            - e2.evalVisit();
   }
}
```

```
public class IdExp extends Exp {
   private String f0;
   public IdExp(String n0) { ... }
   int evalVisit() {
     return getVarValue(f0);
   }
}
public class IntExp extends Exp {
   private int v;
   public IntExp (int n0) { ... }
   int evalVisit() {
     return v;
   }
}
```

What if we need to traverse the tree for another purpose?

Traversing AST:Design Two

Group all methods regarding one functionality into one class. The problem is that we don't know the type of a node.

```
public class EvalVisitor {
   public visit(Exp root) {
      int v1, v2;
      if (e instanceof PlusExp) {
         PlusExp t= (PlusExp) e;
 return visit(t.e1)+visit(t.e2);
      } else if (e instanceof MinusExp) {
         MinusExp t= (MinusExp) e;
 return visit(t.e1)-visit(t.e2);
      } else if (e instanceof IdExp) {
         IdExp t=(IdExp) e;
         return retrieveVal(t.f0);
      } else if (e instanceof NumExp) {
         NumExp t= (NumExp) e;
 return t.v;
      }
   }
}
```

The instanceof and braches are not OO style.

Design Three: Visitor Pattern

The key: do a little handshake and let the node itself tell its type.

```
 public abstract class Exp {
   public abstract int accept (Visitor v);
}
public class PlusExp extends Exp {
   public Exp e1,e2;
   public PlusExp(Exp a1, Exp a2) ...
   int accept(Visitor v) {
     return v.visit(this);
   }
 }
public class MinusExp extends Exp {
   public Exp e1,e2;
   public MinusExp(Exp a1, Exp a2) ...
   int accept(Visitor v) {
     return v.visit(this);
   }
}
```

```
public class EvalVisitor
       implements Visitor {
   public int visit(PlusExp n) {
       return n.e1.accept(this)
             +n.e2.accept(this)
   }
   public int visit(MinusExp n) {
       return n.e1.accept(this)
              -n.e2.accept(this);
   }
   ...
}
```

A very important design pattern, strongly suggest to use it in the project.