Extracted from:

Language Implementation Patterns

Create Your Own Domain-Specific and General Programming Languages

The Pragmatic Programmers

Language Implementation Patterns

Create Your Own Domain-Specific and General Programming Languages

Terence Parr

Edited by Susannah Davidson Pfalzer

With permission of the creator we hereby publish the chess images in Chapter 11 under the following licenses:

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-45-X

ISBN-13: 978-1-934356-45-6

Printed on acid-free paper.

P1.0 printing, December 2009

Version: 2010-1-13

```
Token two = new Token(Token.INT,"2");
AST root = new AST(plus);
root.addChild(new AST(one));
root.addChild(new AST(two));
System.out.println("1+2 tree: "+root.toStringTree());

AST list = new AST(); // make nil node as root for a list
list.addChild(new AST(one));
list.addChild(new AST(two));
System.out.println("1 and 2 in list: "+list.toStringTree());
```

Here is a sample session:

```
$ java Test
1+2 tree: (+ 1 2)
1 and 2 in list: 1 2
$
```

The next pattern, Pattern 10, *Normalized Heterogeneous AST*, is an extension to this pattern that allows multiple node types while retaining the normalized child list.

Related Patterns

Pattern 10, *Normalized Heterogeneous AST* uses normalized lists of children as well but allows nodes to have different class types.

10 Normalized Heterogeneous AST

Purpose

This pattern implements an abstract syntax tree (AST) using more than a single node data type but with a normalized child list representation.

Discussion

This pattern is a variation on Pattern 9, *Homogeneous AST*, on page 111. All we're doing differently is distinguishing between nodes with our implementation language's type system. Because this pattern also uses a normalized child list, we can derive heterogeneous nodes using AST from Pattern 9, *Homogeneous AST*, on page 111 as a base class.

This pattern makes the most sense when we need to store node-specific data and plan on using Pattern 13, *External Tree Visitor*, on page 133. The normalized child list makes it much easier to build external visitors. If you need lots of node-specific methods or plan on using Pattern

12, *Embedded Heterogeneous Tree Walker*, on page 130, use Pattern 11, *Irregular Heterogeneous AST*, on page 116 instead. (An embedded walker has walking methods distributed across the heterogeneous node type definitions.)

Let's flesh out some of the heterogenous node details from Section 4.2, *Implementing ASTs in Java*, on page 98. We added a field, evalType, to track expression type information (see also Pattern 20, *Computing Static Expression Types*, on page 201). evalType tracks the type of the value computed by the expression. For example, the type of 1+2 is integer. We can put this field into an abstract class:

Download IR/Normalized/ExprNode.java

```
public abstract class ExprNode extends AST {
    public static final int tINVALID = 0; // invalid expression type
    public static final int tINTEGER = 1; // integer expression type
    public static final int tVECTOR = 2;  // vector expression type
    /** Track expression type (integer or vector) for each expr node.
     *  This is the type of the associated value not the getNodeType()
     *  used by an external visitor to distinguish between nodes. */
    int evalType;

    public int getEvalType() { return evalType; }
    public ExprNode(Token payload) { super(payload); }
    /** ExprNode's know about the type of an expresson, include that */
    public String toString() {
        if ( evalType != tINVALID ) {
            return super.toString()+"<type="+
                   (evalType == tINTEGER ? "tINTEGER" : "tVECTOR")+">";
        }
        return super.toString();
    }
}
```

Rather than creating a generic node and then adding children to form a + (addition) subtree, we can use AddNode's constructor:

Download IR/Normalized/AddNode.java

```
public class AddNode extends ExprNode {
    public AddNode(ExprNode left, Token addToken, ExprNode right) {
        super(addToken);
        addChild(left);
        addChild(right);
    }
    public int getEvalType() { // ...
```

Note that it's still a good idea to track the + token in the AST node. This helps with a number of things including producing better error messages.

Operand node types for integer and vector literals are straightforward subclasses of ExprNode:

Download IR/Normalized/IntNode.java

```
public class IntNode extends ExprNode {
    public IntNode(Token t) { super(t); evalType = tINTEGER; }
}
```

Download IR/Normalized/VectorNode.java

```
import java.util.List;
public class VectorNode extends ExprNode {
    public VectorNode(Token t, List<ExprNode> elements) {
        super(t); // track vector token; likely to be imaginary token
        evalType = tVECTOR;
        for (ExprNode e : elements) { addChild(e); } // add as kids
    }
}
```

The following test code creates and prints an AST for 1+2.

Download IR/Normalized/Test.java

```
Token plus = new Token(Token.PLUS,"+");
Token one = new Token(Token.INT,"1");
Token two = new Token(Token.INT,"2");
ExprNode root = new AddNode(new IntNode(one), plus, new IntNode(two));
System.out.println(root.toStringTree());
```

Here is a sample session:

```
$ java Test
(+ 1<type=tINTEGER> 2<type=tINTEGER>)
$
```

The serialized tree output indicates that the 1 and 2 children have type tINTEGER. Naturally, the result of the addition operation is also an integer, so the root should have type tINTEGER. In Chapter 8, *Enforcing Static Typing Rules*, on page 198, we'll figure out how to do this computation properly. We'll leave it blank for now.

Related Patterns

This pattern defines node types that subclass AST from Pattern 9, *Homogeneous AST*, on page 111. The next pattern, Pattern 11, *Irregular Heterogeneous AST*, on the next page, uses an irregular child list rather than a normalized list like this pattern.

11 Irregular Heterogeneous AST

Purpose

This pattern implements an abstract syntax tree (AST) using more than a single node data type and with an irregular child list representation.

Discussion

This pattern only differs from Pattern 10, *Normalized Heterogeneous AST*, on page 113 in the implementation of its child pointers. Instead of a uniform list of children, each node data type has specific (named) child fields. In this sense, the child pointers are irregular. In some cases, named fields lead to more readable code. For example, methods can refer to left and right instead of, say, children[0] and children[1].

When building trees from scratch, most programmers follow this pattern. It's very natural to name the fields of a class, in this case naming the children of a node. The big downside to using nodes with irregular children is that it's much less convenient to build tree walkers (such as Pattern 13, *External Tree Visitor*, on page 133). This pattern is fine for small projects where the extra gain in readability is worth the small bit of extra work to implement visitors. Larger projects tend to do so much tree walking, though, that the irregular children prove to be a big hassle.

To see where the pain comes from, look again at thetoStringTree() tree printing method shown in Pattern 9, *Homogeneous AST*, on page 111. Because the children of each node look the same, a single toStringTree() works for all nodes. With irregular children, each node has to have its own toStringTree(). There is no way to access a node's children generically. That means duplicating essentially the same logic just to use different field names. In the source code directory, you'll see that both ListNode.java and AddNode.java have node-specific toStringTree() implementations.

Since each node defines its own child fields, the abstract base class HeteroAST doesn't have a normalized list of children:

Download IR/Hetero/HeteroAST.java

```
public abstract class HeteroAST {   // Heterogeneous AST node type
    Token token;                    // Node created from which token?
```

Node type AddNode is a typical irregular heterogeneous AST implementation. It has specific named child fields and node-specific methods. In this case, there are methods for printing out the tree structure and computing expression value result types:

Download IR/Hetero/AddNode.java

```
public class AddNode extends ExprNode {
    ExprNode left, right; // named, node-specific, irregular children
    public AddNode(ExprNode left, Token addToken, ExprNode right) {
        super(addToken);
        this.left = left;
        this.right = right;
    }
    public String toStringTree() {
        if ( left==null || right==null ) return this.toString();
        StringBuilder buf = new StringBuilder();
        buf.append("(");
        buf.append(this.toString());
        buf.append(' ');
        buf.append(left.toStringTree());
        buf.append(' ');
        buf.append(right.toStringTree());
        buf.append(")");
        return buf.toString();
    }
}
```

The other node type definitions are available in the source directory; they differ only in the child field definitions. There is also a test file in Test.java.

Related Patterns

See Pattern 10, *Normalized Heterogeneous AST*, on page 113.

Up Next

This completes the last tree construction pattern. The next chapter explores how to build tree walkers for these data structures.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Language Implementation Patterns' Home Page
http://pragprog.com/titles/tpdsl
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/tpdsl.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)