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

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

Chapter 8

# Enforcing Static Typing Rules

We derive the meaning of a sentence from both its structure (syntax) and the particular vocabulary symbols it uses. The structure says what to do, and the symbols say what to do it to. For example, in phrase `print x`, the syntax says to print a value, and the symbol `x` says which value to print. Sometimes, though, we write code that make no sense even if the syntax is correct. Such programs violate a language's semantic rules.

Languages typically have lots and lots of semantic rules. Some rules are run-time constraints (*dynamic semantics*), and some are compile-time constraints (*static semantics*). Dynamic semantic rules enforce things like "no division by zero" and "no array index out of bounds." Depending on the language, we can enforce some rules statically such as "no multiplication of incompatible types."

Where to draw the line between static and dynamic rules is up to the language designer. For example, Python is *dynamically typed*, which means that programmers do not specify the types of program values (nor can the compiler infer every type). The Python interpreter enforces all the semantic rules at run-time. C++ is the opposite extreme. Anything goes at run-time, but C++ is *statically typed*. We have to specify the types of all program values. Some languages enforce the same rule statically and then again dynamically to guard against hostile programs. For example, Java does type checking at compile-time as well as at run-time. Both statically and dynamically typed languages are called *type safe* if they disallow operations on incompatible types.

Because statically typed languages are so common, we are going to devote an entire chapter to enforcing static type safety (those readers interested only in implementing dynamically typed languages such as

Python and Ruby can skip this chapter). Here are the patterns we'll discuss:

- Pattern 20, *Computing Static Expression Types*, on page 201. To guarantee type safety, the first thing we've got to do is compute the types of all expressions and expression elements. We assume that the operands of a binary arithmetic operation have the same type. There is no automatic promotion of arithmetic values. Most languages do automatically promote arithmetic values, but technically type computation and promotion are two different operations. That's why we'll look at them separately in this pattern and the next.
- Pattern 21, *Automatic Type Promotion*, on page 210. This pattern demonstrates how to promote operands to have the same or otherwise compatible types. For example, in the expression `3+4.5`, we expect the language to automatically promote integer `3` to a floating-point value.
- Pattern 22, *Enforcing Static Type Safety*, on page 218. Once we know the types of all expressions, we can enforce type safety. This amounts to checking for operand-operator and assignment type compatibility.
- Pattern 23, *Enforcing Polymorphic Type Safety*, on page 225. The notion of type compatibility is a little bit looser in object-oriented languages. We have to deal with polymorphic assignments. We can, for example, assign a `Manager` object reference (pointer) to an `Employee` reference: `e = m;`. Polymorphic means that a reference can point at multiple types. In contrast, assignments in non-object-oriented languages must be between identical types. This pattern explains how to check for polymorphic type compatibility.

Before jumping into the patterns, we need to agree on a specific language that we can use as a common thread throughout this chapter. There's no way we can describe all possible semantic rules for all languages, so we'll have to focus on a single language. Using C as a base is a good choice because it's the progenitor of the statically typed languages commonly in use today (C++, C#, and Java). For continuity, we'll augment our Cymbol language from Chapter 6, *Tracking and Identifying Program Symbols*, on page 148 (with some more operators to make it interesting).

Cymbol has the following features (when in doubt, assume C++ syntax and semantics):

- There are **struct**, function, and variable declarations.
- The built-in types are **float**, **int**, **char**, **boolean**, and **void**. Along with built-in type **boolean**, we have true and false values.
- There are no explicit pointers (except in Pattern 23, *Enforcing Polymorphic Type Safety*, on page 225), but there are one-dimensional arrays: `int a[];`. Like C++, we can initialize variables as we declare them: `int i = 3;`. Also like C++, we can declare local variables anywhere within a function, not just at the start like C.
- There are **if**, **return**, assignment, and function call statements.
- The operators are +, -, *, /, <, >, <=, >=, !=, ==, !, and unary -. Beyond the usual expression atoms like integers and identifiers, we can use function calls, array references, and struct/class member accesses.

We're going to enforce a number of type safety rules. In a nutshell, all operations and value assignments must have compatible operands. In Figure 8.2, on page 219, we see the exact list of semantic type rules. Furthermore, we're going to check symbol categories. The type of expressions on the left of the . member access operator must be of type **struct**. Identifiers in function calls must be functions. Identifiers in array references must be array symbols.

Now we just have to figure out how to implement those rules. All the patterns in this chapter follow the same general three-pass strategy. In fact, they all share the first two passes. In the first pass, a Cymbol parser builds an AST. In the second pass, a tree walker builds a scope tree and populates a symbol table. Pattern 20, *Computing Static Expression Types*, on the following page is the third pass over the AST and computes the type of each expression. Pattern 21, *Automatic Type Promotion*, on page 210 augments this third pass to promote arithmetic values as necessary. We'll assume valid input until Pattern 22, *Enforcing Static Type Safety*, on page 218. In that pattern, we'll add type checking to the third tree pass to enforce our semantic rules.

In practice, you might squeeze the second and third or the first and second passes into a single pass for efficiency. It might even be possible to reduce this to a single pass that parses, defines symbols, computes types, and checks type compatibility. Unless run-time speed is critical, though, consider breaking complicated language applications down into as many bite-size chunks as possible.

Here's a quick summary of when to apply the patterns:

| **Pattern** | **When to Apply** |
|---|---|
| Pattern 20, *Computing Static Expression Types* | This pattern is a component of any type safety checker such as Pattern 22, *Enforcing Static Type Safety*, on page 218 and Pattern 23, *Enforcing Polymorphic Type Safety*, on page 225. |
| Pattern 21, *Automatic Type Promotion*, on page 210 | Automatically promoting types is also really just a component of a type checker. If your language doesn't support automatic promotion (like ML), you don't need this pattern. |
| Pattern 22, *Enforcing Static Type Safety*, on page 218 | You'll need this pattern if you're parsing a non-object-oriented programming language such as C. |
| Pattern 23, *Enforcing Polymorphic Type Safety*, on page 225 | Use this pattern if you're dealing with an object-oriented language such as C++ or Java. |

OK, let's get to it. Don't worry if the process of computing and checking types seems complicated. We'll take it slowly, component by component. In fact, static type analysis for C and its descendents is not too bad. The following patterns break the problem down into easy-to-digest pieces.

## 20 Computing Static Expression Types

□

### Purpose

*This pattern explains how to statically compute the type of expressions in languages with explicit type declarations like C.*

You'll be able to extrapolate from this pattern everything you'd need to build a static type analyzer for C, C++, Java, or C#. Every compiler for

| **Subexpression** | **Result Type** |
|---|---|
| true, false | **boolean**. |
| Character literal | **char**. |
| Integer literal | **int**. |
| Floating-point literal | **float**. |
| *id* | The declared type of the variable identified by *id*. |
| !«*expr*» | **boolean**. |
| -«*expr*» | The result type is the same as «*expr*»'s type. |
| «*expr*».*id* | The declared type of the field identified by *id*. |
| a[«*expr*»] | The declared array element type. For example, a[i] has type **float** if a has declaration float a[]. |
| f(«*args*») | The declared return type of function f. |
| «*expr*» *bop* «*expr*» | Since both operands have the same type, we can simply choose the type of the left operand as a result type; *bop* is in {+, -, *, /}. |
| «*expr*» *relop* «*expr*» | **boolean** where *relop* is in {<, >, <=, >=}. |
| «*expr*» *eqop* «*expr*» | **boolean** where *eqop* is in {!=, ==}. |

Figure 8.1: Cymbol expression type computation rules

those languages implements an extended version of this pattern. So, do static bug analyzers such as FindBugs[1] and Coverity.[2]

## Discussion

Type computation is an extremely broad topic. To make things more concrete, we'll focus on the type computation rules for Cymbol itemized in Figure 8.1.

Computing the type of an expression is a matter of computing the type of all elements and the result type of all operations.

1. http://findbugs.sourceforge.net
2. http://coverity.com/html/prevent-for-java.html

For example, to compute the type of f(1)+4*a[i]+s.x, we proceed as follows:

| **Subexpression** | **Result Type** |
|---|---|
| 1 | **int** |
| f(1) | **int** |
| 4 | **int** |
| i | **int** |
| a[i] | **int** |
| 4*a[i] | **int** |
| f(1)+4*a[i] | **int** |
| s | **struct** S |
| s.x | **int** |
| f(1)+4*a[i]+s.x | **int** |

These computations are pretty dull because we're assuming the operands are all the same type. Technically, we could stop computing the type after encountering first operand: f(1). The entire expression result type has to be integer because f returns an integer. In practice, though, two things can happen: we might need to promote a simpler type like **char** to **int** and sometimes programmers make mistakes (operand types can be incompatible). This pattern just sets up the proper action plan for the next two patterns. We'll graft type promotion and type checking onto this pattern later.

## Implementation

The general strategy we'll use is to parse a Cymbol program into an AST and then walk that tree twice. The first tree walk defines symbols, and the second walk resolves symbols and computes expression types. The first two passes come from Pattern 18, *Symbol Table for Data Aggregates*, on page 178, so we can focus on the final type resolution and computation tree walk.

Once we have an AST and a populated symbol table (courtesy of Cymbol.g and Def.g), we can describe the type computation rules as tree pattern-action pairs. The actions compute types and annotate the AST with them. Using Pattern 13, *External Tree Visitor*, on page 133, we could walk the tree looking for the patterns. We have to be careful, though, how we match expression elements. For example, we have to consider isolated identifiers and identifiers in array references differently. In ANTLR notation, that means we can't simply make a tree pattern rule like this:

```
id : ID {«action»} ;
```

To encode context information, we need Pattern 14, *Tree Grammar*, on page 136 rather than a set of isolated tree patterns. That said, we don't want to resort to a full tree grammar because we care only about expressions in this pattern. To get the best of both worlds, we can use Pattern 15, *Tree Pattern Matcher*, on page 140 to look for **EXPR** root nodes and then invoke a type computation rule to traverse the expression subtree:

Download semantics/types/Types.g

```
bottomup // match subexpressions innermost to outermost
    :   exprRoot // only match the start of expressions (root EXPR)
    ;

exprRoot // invoke type computation rule after matching EXPR
    :   ^(EXPR expr) {$EXPR.evalType = $expr.type;} // annotate AST
    ;
```

This way we only have to specify the type computation rules and can totally ignore the AST structure outside of expressions.

The meat of our implementation is rule **expr**, which computes the subexpression types:

Download semantics/types/Types.g

```
expr returns [Type type]
@after { $start.evalType = $type; } // do after any alternative
    :   'true'      {$type = SymbolTable._boolean;}
    |   'false'     {$type = SymbolTable._boolean;}
    |   CHAR        {$type = SymbolTable._char;}
    |   INT         {$type = SymbolTable._int;}
    |   FLOAT       {$type = SymbolTable._float;}
    |   ID {VariableSymbol s=(VariableSymbol)$ID.scope.resolve($ID.text);
            $ID.symbol = s; $type = s.type;}
    |   ^(UNARY_MINUS a=expr)   {$type=symtab.uminus($a.start);}
    |   ^(UNARY_NOT a=expr)     {$type=symtab.unot($a.start);}
    |   member      {$type = $member.type;}
    |   arrayRef    {$type = $arrayRef.type;}
    |   call        {$type = $call.type;}
    |   binaryOps   {$type = $binaryOps.type;}
    ;
```

The first few alternatives encode the type computation rules with inline actions for the literals and identifiers. The $ID.scope.resolve($ID.text) expression deserves some explanation. $ID.text is the text of the identifier that we need to look up with resolve(). resolve() needs the identifier's context (surrounding scope), which our definition phase conveniently stashed as the **ID** AST node's scope field. Expression $start refers to the first node matched by enclosing rule **expr**.

The tree grammar handles the more complicated patterns via a few small helper rules in SymbolTable such as uminus() and unot():

Download semantics/types/SymbolTable.java

```
public Type uminus(CymbolAST a) { return a.evalType; }
public Type unot(CymbolAST a)   { return _boolean; }
```

Rule **expr** also annotates the root of the subexpression subtree with the type it computes (via $start.evalType = $type;). Because a static type analyzer is normally just a component of a larger language application, we need to store type information somewhere rather than throwing it out. We'll store the type in field evalType of a customized AST node, CymbolAST:

Download semantics/types/CymbolAST.java

```
public class CymbolAST extends CommonTree {
    public Scope scope;   // set by Def.g; ID lives in which scope?
    public Symbol symbol; // set by Types.g; point at def in symbol table
    public Type evalType; // The type of an expression; set by Types.g
```

Continuing on with the type computation rules, here is how to compute the type of a member access operation:

Download semantics/types/Types.g

```
member returns [Type type]
    :   ^('.' expr ID)          // match expr.ID subtrees
        { // $expr.start is root of tree matched by expr rule
        $type = symtab.member($expr.start, $ID);
        $start.evalType = $type; // save computed type
        }
    ;
```

Notice that the left side of the operation can be any expression according to the grammar. This handles cases such as functions that return **struct** values as in f().fieldname. The member() method in the SymbolTable looks up the field within the scope of the expression on the left side:

Download semantics/types/SymbolTable.java

```
public Type member(CymbolAST expr, CymbolAST field) {
    StructSymbol scope=(StructSymbol)expr.evalType; // get scope of expr
    Symbol s = scope.resolveMember(field.getText());// resolve ID in scope
    field.symbol = s;  // make AST point into symbol table
    return s.type;     // return ID's type
}
```

It retrieves the type of the expression via the evalType AST field. evalType is set as a side effect of calling rule **expr** in **member** and must point at a StructSymbol.

The next rule computes types for array references. It also delegates the type computation to SymbolTable (the actions for these rules will get bigger in the following patterns; it's a good idea to tuck them out of the way as methods in another class):

Download semantics/types/Types.g

```
arrayRef returns [Type type]
    :   ^(INDEX ID expr)
        {
        $type = symtab.arrayIndex($ID, $expr.start);
        $start.evalType = $type; // save computed type
        }
    ;
```

The type of an array reference is just the element type of the array (the index isn't needed):

Download semantics/types/SymbolTable.java

```
public Type arrayIndex(CymbolAST id, CymbolAST index) {
    Symbol s = id.scope.resolve(id.getText());
    VariableSymbol vs = (VariableSymbol)s;
    id.symbol = vs;
    return ((ArrayType)vs.type).elementType;
}
```

Function calls consist of the function name and an optional list of expressions for the arguments. The **call** rule collects all this information and passes it to a helper in SymbolTable:

Download semantics/types/Types.g

```
call returns [Type type]
@init {List args = new ArrayList();}
    :   ^(CALL ID ^(ELIST (expr {args.add($expr.start);})*))
        {
        $type = symtab.call($ID, args);
        $start.evalType = $type;
        }
    ;
```

The type of a function call is the return type of the function (we'll ignore the argument types until we do type promotion and type checking):

Download semantics/types/SymbolTable.java

```
public Type call(CymbolAST id, List args) {
    Symbol s = id.scope.resolve(id.getText());
    MethodSymbol ms = (MethodSymbol)s;
    id.symbol = ms;
    return ms.type;
}
```

Finally, we come to the binary operators (binary in the sense that they have two operands). It turns out that we'll ultimately need to deal separately with the arithmetic, relational, and equality operators. For consistency with future patterns, we'll trigger different helper methods:

Download semantics/types/Types.g

```
binaryOps returns [Type type]
@after { $start.evalType = $type; }
    :   ^(bop a=expr b=expr)   {$type=symtab.bop($a.start, $b.start);}
    |   ^(relop a=expr b=expr) {$type=symtab.relop($a.start, $b.start);}
    |   ^(eqop a=expr b=expr)  {$type=symtab.eqop($a.start, $b.start);}
    ;
```

Because we assume that the operand types of arithmetic operators are identical, there is no computation to do. We can just arbitrarily pick the type of the left operand. The relational and the equality operators always yield **boolean** types:

Download semantics/types/SymbolTable.java

```
public Type bop(CymbolAST a, CymbolAST b)   { return a.evalType; }
public Type relop(CymbolAST a, CymbolAST b) { return _boolean; }
public Type eqop(CymbolAST a, CymbolAST b)  { return _boolean; }
```

To put everything together, we need to build an AST and then perform two tree walks:

Download semantics/types/Test.java

```
// CREATE PARSER AND BUILD AST
CymbolLexer lex = new CymbolLexer(input);
final TokenRewriteStream tokens = new TokenRewriteStream(lex);
CymbolParser p = new CymbolParser(tokens);
p.setTreeAdaptor(CymbolAdaptor);  // create CymbolAST nodes
RuleReturnScope r = p.compilationUnit();   // launch parser
CommonTree t = (CommonTree)r.getTree();    // get tree result

// CREATE TREE NODE STREAM FOR TREE PARSERS
CommonTreeNodeStream nodes = new CommonTreeNodeStream(t);
nodes.setTokenStream(tokens);        // where to find tokens
nodes.setTreeAdaptor(CymbolAdaptor);
SymbolTable symtab = new SymbolTable();

// DEFINE SYMBOLS
Def def = new Def(nodes, symtab); // pass symtab to walker
def.downup(t); // trigger define actions upon certain subtrees

// RESOLVE SYMBOLS, COMPUTE EXPRESSION TYPES
nodes.reset();
Types typeComp = new Types(nodes, symtab);
typeComp.downup(t); // trigger resolve/type computation actions
```

After the tree walks, we have annotated all nodes within expressions with two pointers. symbol points at its symbol definition in the symbol table, and evalType points at the node's computed type. To print out our handiwork, we can use Pattern 13, *External Tree Visitor*, on page 133 to trigger a method calledshowTypes() on each expression node. To get a bottom-up, innermost to outermost traversal, we use a postorder walk:

Download semantics/types/Test.java

```
// WALK TREE TO DUMP SUBTREE TYPES
TreeVisitor v = new TreeVisitor(new CommonTreeAdaptor());
TreeVisitorAction actions = new TreeVisitorAction() {
    public Object pre(Object t) { return t; }
    public Object post(Object t)  {
        showTypes((CymbolAST)t, tokens);
        return t;
    }
};
v.visit(t, actions); // walk in postorder, showing types
```

Method showTypes() just prints out subexpressions and their types for nodes with non-null evalType fields:

Download semantics/types/Test.java

```
static void showTypes(CymbolAST t, TokenRewriteStream tokens) {
    if ( t.evalType!=null && t.getType()!=CymbolParser.EXPR ) {
        System.out.printf("%-17s",
                          tokens.toString(t.getTokenStartIndex(),
                                          t.getTokenStopIndex()));
        String ts = t.evalType.toString();
        System.out.printf(" type %-8s\n", ts);
    }
}
```

Let's run the following sample Cymbol file through our test rig:

Download semantics/types/t.cymbol

```
struct A {
  int x;
  struct B { int y; };
  struct B b;
};
int i=0; int j=0;
void f() {
  struct A a;
  a.x = 1+i*j;
  a.b.y = 2;
  boolean b = 3==a.x;
  if ( i < j ) f();
}
```

Here's how to build the test rig and run it on t.cymbol (it's the same for all patterns in this chapter):

```
$ java org.antlr.Tool Cymbol.g Def.g Types.g
$ javac *.java
$ java Test t.cymbol
0                  type int
0                  type int
a                  type struct A:{x, B, b}
a.x                type int
1                  type int
i                  type int
j                  type int
i*j                type int
1+i*j              type int
a                  type struct A:{x, B, b}
a.b                type struct B:{y}
a.b.y              type int
2                  type int
3                  type int
a                  type struct A:{x, B, b}
a.x                type int
3==a.x             type boolean
i                  type int
j                  type int
i < j              type boolean
f()                type void
$
```

This pattern identifies the basic type computations for expression elements and operations. It's fairly restrictive in that operand types within a single operation must be identical such as integer plus integer. Still, we've created the basic infrastructure needed to support automatic promotion such as adding integers and floats. The next pattern defines the rules for arithmetic type promotion and provides a sample implementation. Its implementation builds upon the source code in this pattern.

## Related Patterns

This pattern uses Pattern 18, *Symbol Table for Data Aggregates*, on page 178 to build a scope tree and populate the symbol table. It uses Pattern 13, *External Tree Visitor*, on page 133 to print out type information. Pattern 21, *Automatic Type Promotion*, on the next page, Pattern 22, *Enforcing Static Type Safety*, on page 218, and Pattern 23, *Enforcing Polymorphic Type Safety*, on page 225 build upon this pattern.