Abstract Syntax Tree Implementation Idioms

Joel Jones
jones@cs.ua.edu
Department of Computer Science
University of Alabama

1 Context

You are implementing a language, and the language is sufficiently complex that a direct source-to-source translation is not desirable. An appropriate form for representing the essential characteristics of the structure of the input is needed.

A language defines a mapping from symbols to meaning. Many software implementation tasks are aided by specifying and implementing a domain-specific language. For example, a common solution to distributing logic across disparate environments is to encode the logic as a program in a domain-specific language and provide code generators to translate the logic to the different environments [12], [13].

Language translators are typically described using "T" diagrams. A source language *S* is translated into a target language *T* by an *S*-into-*T* translator expressed in implementation language *L*. The problem these idioms address is how the source program should be represented inside the translator, by using the constructs of the implementation language. These representations are constructed as a result of parsing the source language.

2 Forces

- The target language (*T*) has been chosen. For example, C can be used if access to low-level features is needed.
- The source language (*S*) has been partially designed and the complexity of the translation process is already such that a direct translation from the source language to the target language is not desired.

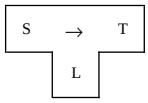

Figure 1: T diagram

- An implementation language (L) for the translator has been chosen.
- As much as possible, the implementation language's compiler should catch errors in the implementation of the translator.
- Introducing new language elements in the source language should be easy.

3 Solution

Implement abstract syntax trees (ASTs) using implementation language specific idioms. In the sections below, the language specific idioms are presented. In this section, language independent information is given.

An abstract syntax tree (AST) captures the essential structure of the input in a tree form, while omitting unnecessary syntactic details. ASTs can be distinguished from concrete syntax trees by their omission of tree nodes to represent punctuation marks such as semi-colons to terminate statements or commas to separate function arguments. ASTs also omit tree nodes that represent unary productions in the grammar. Such information is directly represented in ASTs by the structure of the tree.

ASTs can be created with hand-written parsers or by code produced by parser generators. ASTs are generally created bottom-up.

When designing the nodes of the tree, a common design choice is determining the granularity of the representation of the AST. That is, whether all constructs of the source language are represented as a different type of AST nodes or whether some constructs of the source language are represented with a common type of AST node and differentiated using a value. One example of choosing the granularity of representation is determining how to represent binary arithmetic operations. One choice is to have a single binary operation tree node, which has as one of its attributes the operation, e.g. "+". The other choice is to have a tree node for every binary operation. In an object-oriented language, this would results in classes like: AddBinary, SubtractBinary, MultiplyBinary, etc. with an abstract super class of Binary. The second form is preferred if there will be behavior associated with the tree nodes.

There are systems which will automatically generate AST implementations from a little language, such as Zephyr ASDL [17] or by an integrated specification with a parser generator, Eli [10]. In such systems, a simple specification of the tree-structure of the AST is given and a program-generator reads this specification and generates code for representing and creating the AST in some high-level language.

4 Resulting Context

With the implementation of the AST components complete, the subsequent phases of the compiler can be implemented. In the simplest translations, the next stage would be to traverse the ASTs and generate the target language. If the translation is more complicated, then further decisions must be made as to which code transformations should be performed on the ASTs and which should be done on some other intermediate form

(IR). Such code transformations may have the purpose of improving the run-time performance of the target program or to ease the translation into the target language. Large compilation systems typically perform semantic analysis on the AST, then translate the AST into some other intermediate form immediately. These IRs would typically be either some kind of register-transfer language or static-single assignment form. In some translators, transformations may be performed on the ASTs first, performing an AST to AST translation. Some examples would be loop transformations in FORTRAN compilers or refactoring using tools such as the Refactoring Browser[16].

If transformations are performed on the ASTs, then a decision has to be made as to whether to implement readers and writers for the AST. A human-readable form of the AST can be useful for debugging or certain forms of program analysis.

A useful feature for an AST implementation is to have an instantiation of the Builder pattern[8] with a function that will take strings to ASTs. These are useful for generating test cases before the grammar is completed or when new language features are being experimented with. One example is in the Marion system [4], where a `tree` function was used to allow for the easy generation of function epilogues, using a format string as the first argument, followed by a varying number of subsequent arguments. By nesting calls to `tree` as arguments to `tree`, arbitrarily complex ASTs could be generated.

5 Imperative Languages

5.1 Context

Strictly imperative languages, like C or Pascal, provide the least support for implementing ASTs.

* * *

Avoid these languages if possible. If this isn't possible, then implement the AST in a disciplined fashion, making representation and naming choices uniformly.

* * *

5.2 Solution

GENERATORS

Use generator tools to create an AST implementation from a simple specification. The input language for an AST generator will typically contain the name of the generic node's type, the name of all specific node's types, and member names and types for the specific nodes. One such tool is the generator for Abstract Definition Language (ASDL) [17] which includes C as one of its output languages. It is also not hard to build a simple version of such a tool using a scripting language such as AWK or Perl. In addition to generating the data type declarations, an AST code generator should also create "constructor" functions which take as arguments the children of the node and return initialized instances of the specific nodes of the AST. For an example of an ASDL specification, see the Appendix.

NODES AS VARIANT RECORDS

To represent the nodes of the AST in these languages, some form of variant record must be used. In Pascal, the variant record is supported directly. In C, variant records can be emulated using a struct with an enum member and a union member. The enclosing record (struct) represents the generic node of the AST. The embedded records (members of the union) represent the various specific nodes of the AST. For an example, see the Appendix.

COMPILER CHECKED CASE STATEMENT DISPATCH

When code is written to dispatch on the type of an AST node, write this as a case or switch statement, rather than as a sequence of cascading if statements. This allows for the specification of a compile time check of the implementation code. Most Pascal and C compilers have a compiler option that causes a message to be generated if a multi-way branch on a variant record tag or enumerated type does not cover all possible values. For instance, to insure that a C switch statement over a enum type is complete, the gcc compiler has the `-Wswitch` option.

6 Functional Languages

6.1 Context

Functional programming languages provide good support for implementing ASTs.

* * *

The appropriate features differ between Scheme and ML. The use of user-defined recursive datatypes are preferred. Scheme doe not support "datatypes" ("types") as ML (Haskell) does.

* * *

6.2 Solution

NODES AS DEFINE OVER LET OVER DEFINE

In Scheme, use closures to provide run-time checks of coverage. To represent ASTs and the functions on them in Scheme, use define-over-let-over-define[1, pp. 167–175]. This style embeds local state and the associated functions inside a constructor function definition. For the local state, contained in the let, define a binding for every child. Any attempt to execute a function that isn't supported will generate an error, rather than failing silently.

NODES AS DATATYPES

In ML use a datatype declaration, which provides for compile-time checks [15]. The datatype represents the general AST node type and each of the constructors represent

the specific AST node type. For each constructor, list the type of the children as one of the constructor arguments.

ARGUMENT PATTERN MATCHING

In ML, when defining functions over the ASTs, use function argument pattern matching over the AST datatype. The ML compiler will give a compile time warning if any of the functions are not exhaustive over the AST constructors. See the Appendix for an example.

7 Object-oriented Languages

7.1 Context

Object oriented languages provide features that are better suited for representing ASTs than do imperative languages like C and Pascal. These features should be used in preference to enumerated types, unions, etc.

* * *

When implementing ASTs using an object-oriented language, polymorphism, rather than switch statements, should be used to dispatch on the type of the elements of the tree.

* * *

7.2 Solution

IMPLEMENT TREE NODES AS CLASSES

Implement the elements of the tree as instances of classes. Instance variables of the AST classes are used to represent the children of the node. Instance variables of leaf nodes are used to hold information about the node's value, e.g. literal values, references into a symbol table, etc. All of the AST node classes are directly or indirectly derived from an abstract base class. Identify classes that share common attributes and use the "Extract Superclass" refactoring to create a common abstract base class[7].

AST CLASS NAMES

In naming the various classes, first give precedence to any local coding conventions. In the absence of their direction, the following guidelines should be followed. First, if there is only one AST in the program, name the abstract base class that is the root of the AST node inheritance hierarchy "AST." If there are multiple AST hierarchies, then name them "*Foo*AST", "*Bar*AST", etc. Name other abstract base classes based upon the common feature that they represent, e.g. `Declaration` The classes derived from the other base classes should be named by the concatenation of their specific feature and their immediate base class, e.g. `VarDeclaration`.

SMART NODES/DATA-ONLY NODES

There are two alternatives in designing AST nodes—"smart" objects or "dumb" objects[3]. If there is only one client for the AST, the smart object approach is used. There, the nodes of the AST provide a method for performing the work the client requires. If there are multiple clients, then dumb objects should be used. With dumb objects, there are few or no methods implemented by the AST nodes.

AST VISITOR

To perform work with the ASTs, the VISITOR pattern should be used. To instantiate the VISITOR pattern for the Implement AST pattern, a `visit` method is defined in each AST. The `visit` method has a parameter for the Visitor object. The Visitor object might do type-checking, code-generation, etc.

PROGRAMMATIC AST CLASS GENERATION

There exist parser generators that will produce class definitions for the abstract syntax tree from the parser specification, in addition to generating the parser. For C++, there is YACC++ [6], which generates C++ code. For Java, there is SLY [18], which generates Java.

8 Other Details

Obviously, these idioms are embedded in the rich context of compilers and other translation techniques. Just as automating the process of generating AST implementations is a useful time saver, parser generators are another. Many of these exist for imperative languages, such as yacc[14], and bison[14], as well as those for object-oriented languages, t-gen[9], JavaCup[11], and SmaCC[5], and for functional languages, such as ml-yacc[2]. To generate ASTs using these parser generators, calls to the AST constructors are inserted into the action code section of the parser-specification.

9 Related Patterns

These idioms elaborate Interpreter [8] by giving directives on how to create abstract syntax trees in various different languages.

These idioms are similar to Composite [8] in that they specify how to build tree-like structures. However, Composite is oriented towards composition of leaf nodes with there being only a single container class, whereas ASTs tend to have mostly container classes. Furthermore, a generic child iteration pattern is rarely useful, as the children of a node have a much more constrained relationship to their parent.

The Visitor [8] pattern is commonly used to implement traversals when using dumb objects.

The Tabular Code [12] pattern can be used to alleviate the tedium of coding constructors, destructors and accessors directly.

10 Acknowledgments

Thanks go to the shepherd, Alejandra Garrido, for helping to improve the presentation and pointing me to several works that I was previously unfamiliar with. Thanks also go to Don Yessick for several (heated) discussions on the proper role of inheritance in implementing ASTs in object-oriented systems.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, USA, 1985.

[2] Andrew Appel and David R. Tarditi. Ml-yacc user's manual version 2.3. http://www.cs.princeton.edu/ appel/modern/ml/ml-yacc/manual.html, 1994.

[3] Andrew W. Appel. *modern compiler implementation in Java*. Cambridge University Press, 1998.

[4] David Gordon Bradlee. Retargetable instruction scheduling for pipelined processors. Technical Report TR 91-08-07, University of Washington, 1991.

[5] John Brant and Don Roberts. SmaCC (Smalltalk compiler-compiler). http://www.refactory.com/Software/SmaCC/.

[6] Compiler Resources Inc's. Yacc++® and the language objects library. http://world.std.com/ compres/, 2003.

[7] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994. ISBN 0-201-63361-2.

[9] J. O. Graver. The evolution of an object-oriented compiler framework. *Software—Practice and Experience*, 22(7):519–535, July 1992.

[10] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, February 1992.

[11] Scott Hudson. Java cup LALR parser generator for Java. http://www.cs.princeton.edu/~appel/modern/Java/CUP/, 1999.

[12] Joel Jones. Tabular code generation: Write once, generate many. Pattern Languages of Program Design, 2002.

[13] Sam Kamin. Language implementation via lightweight embedded program generators (extended abstract). http://citeseer.nj.nec.com/11897.html.

[14] John Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly, 2nd edition, 1992.

[15] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, New York, NY, second edition, 1996.

[16] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[17] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In USENIX, editor, *Proceedings of the Conference on Domain-Specific Languages, October 15–17, 1997, Santa Barbara, California*, pages ??–??, Berkeley, CA, USA, 1997. USENIX.

[18] Don Yessick and Joel Jones. Reinventing the wheel or not yet another compiler compiler compiler. In *Southeast ACM Conference*, 2002.

A Appendix

The following sections give examples for the languages discussed in the body. The example is taken from the discussion of Interpreter from [8].

A.1 Zephyr ASDL

```
booleanexp = Variable(identifier id)
           | Constant(boolean b)
           | OrExp(booleanexp left, booleanexp right)
           | AndExp(booleanexp left, booleanexp right)
           | NotExp(booleanexp exp)
```

A.2 C

The "`.h`" file:

```
typedef struct booleanExp *booleanExp_ty;
typedef char* identifier;
typedef int boolean;

enum booleanExp_type {
        VARIABLE, CONSTANT, OREXP, ANDEXP, NOTEXP
} ;

struct booleanExp {
        enum booleanExp_type kind;
        union {
            struct { identifier id; } variable;
            struct { boolean b; } constant;
```

```
            struct {
                booleanExp_ty left;
                booleanExp_ty right;
            } orExp;
            struct {
                booleanExp_ty left;
                booleanExp_ty right;
            } andExp;
            struct { booleanExp_ty exp; } notExp;
        } u;
};
```

The "`.c`" file:

```
#include "booleanExp.h"
booleanExp_ty
mkVariable(identifier id) {
        booleanExp_ty p;

        p = (booleanExp_ty) malloc(sizeof(*p));
        p->kind = VARIABLE;
        p->u.variable.id = id;
        return p;
}
booleanExp_ty
mkConstant(boolean b) { /* ... */ }
/* ... */
```

A.3 ML

```
type identifier = string;
datatype booleanExp = Variable of identifier
                    | Constant of bool
                    | OrExp of (booleanExp * booleanExp)
                    | AndExp of (booleanExp * booleanExp)
                    | NotExp of booleanExp;
```

A.4 Java

```
abstract class AST {
}

public class Variable extends AST {
    public Identifier id;
    public Variable(Identifier id) {
        this.id = id;
    }
```

```
}
public class Constant extends AST { /* ... */ }
public class OrExp extends AST { /* ... */ }
```