ATTRIBUTE GRAMMARS AND THE TEACHING OF
COMPILER DESIGN AND IMPLEMENTATION

David G. Wonnacott
davew@cs.haverford.edu
http://www.cs.haverford.edu/people/davew
Haverford College, Haverford PA
June 20, 2006

ABSTRACT

Attribute grammars can help separate issues of language semantics from issues of compiler implementation. We have found that a judicious use of attribute grammars fits very well with our goals for our Compiler Design course. This paper describes the goals of our course and provides a detailed look at how attribute grammars help us to achieve these goals.

1 INTRODUCTION: WHY TEACH COMPILER DESIGN?

Most computer scientists and programmers will probably never need to build (or even understand) a compiler. We therefore see the primary goal of our Compiler Design course to be the teaching of “larger lessons” about computer science. The focus on one large evolving lab project (the compiler) lets us emphasize elements of software engineering, such as the value of precise specifications, careful design, attention to invariant properties, modularity, assertion checking, code clarity, and careful testing, and the importance of using libraries rather than doing everything from scratch.

In addition, the history of advances in compiler design over the decades lends an unusual perspective on software engineering issues that might otherwise not be experienced in a single-semester course. The introduction of automata for lexical analysis and parsing in the 1960’s and early 1970’s [1, Bibliographic Notes for Chapters 3 and 4] provides insight into the interplay between theory

---

1. This is an extended version of a paper published at CCSCE 2006, and copyright Consortium for Computing Sciences in Colleges (CCSC); this version is available from the author’s web site.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to publish, requires a fee and/or specific permission.
and practice in computer science. The use of BNF to describe grammars (dating from the same era) reinforces one of the larger lessons from our programming languages course, “If your idea cannot be expressed well in an existing language, invent a language for it and build a translator”. These issues are important in the construction of large-scale software, and also serve to illustrate other principles, such as the relationship between clarity and efficiency (in some cases, such as scanning, increased clarity and precision come without significant run-time cost [2, Chapter 2]).

1.1 Why (And How) Do We Teach Attribute Grammars?

After our discussion of scanning, parsing, automata, and BNF, it seems that we have a choice: We could continue our use of special-purpose languages by introducing attribute grammars [1, Chapter 5] and using a tool such as TAG [10, Chapter 8] to build later stages of our compiler, or we could turn to traditional software engineering topics by coding later stages of compilation as modules in a general-purpose language such as C++ or Java.

We have chosen a “middle ground” between these two options. We discuss attribute grammars in lecture and use them to specify the results of later stages of compilation. However, we let students implement these passes in C++ (typically by evaluating attributes via general-purpose traversal functions, some of which they must build themselves). This lets us discuss large-scale programming in a general-purpose language without giving up our use of special-purpose languages. It also lets us choose from a wide variety of textbooks and compiler projects — we currently use Andrew Appel’s textbook and Tiger language [2]. While Appel’s text does not emphasise attribute grammars, it does provide a concise treatment of parser generators and good coverage of dynamic programming for instruction selection and fixed-point calculations for dataflow analysis. Furthermore, the Tiger language forces us to address various issues in ways that fit well with our sequence of lab assignments.

The remainder of this paper illustrates our approach, focusing on our use of attribute grammars for symbol table construction. Section 2 covers our use of attribute grammars to present concepts during lecture. Section 3 outlines the tasks required of the students in lab and presents several implementation strategies. Section 4 contrasts our approach with other work. Section 5 concludes by enumerating the other uses of attribute grammars in our course and giving our overall reflections about our approach.

2 ATTRIBUTES, SPECIFICATIONS, AND TRAVERSALS

Attribute grammars provide a vehicle for separating the new ideas in a compiler course from the details of programming and data structure design that the students have seen in prior courses. For example, we introduce lexical scoping and type checking through a series of lectures and in-class exercises
involving attribute grammars. This approach provides a precise specification and gives the students an opportunity to “try out” various definitions, without getting distracted by issues such as tree traversal order.

Consider the pseudo-code shown in Figure 1, in which the initialization of the variable a refers to b, and that of b refers to a. Languages may allow references only backward within a scope, as in C++, in which case the first initialization is illegal (unless there is another b in an outer scope, of course); they may not allow any references within a block of declarations, as in Common Lisp’s let [12, Section 7.5], in which case both initializations are illegal; or they may allow arbitrary references within a set of declarations, in which both initializations are legal.

```
let
    var a : int := b + 1
    var b : int := a
in
    a + b
end
```

**Figure 1. Example for Discussing Scope Rules**

Students can get a feel for these options for scope rules by manually labeling a few abstract syntax trees (AST’s). In lecture, we present the C++-style scoping rules in the YACC-like notation [7] shown in Figure 2 (the Symbol_Table functions create empty or single-entry symbol tables, fuse joins two disjoint tables or gives an error for name collisions, and shadow combines symbol tables for inner and outer scopes). We use these rules to guide students through the process of labeling the AST for the program in Figure 1. We then give students time to work through more complex examples before continuing the lecture. Attribute grammars provide precise rules for the students to perform these labelings, just as they provide precise rules for compiler-generation tools to compute attribute values.

Once our students are comfortable with the attribute grammar for C++-style scoping, we illustrate Lisp-style scoping by changing the definition of $2.avail$ to be $$.avail$ for the production decs : dec decs, and ask the students to re-label their AST’s. Finally, we illustrate the bi-directional option by defining both $2.avail$ and $4.avail$ to be shadow($2.decl, $$.avail$) in the production exp : LET decs IN exp, and both $1.avail$ and $2.avail$ to be $$.avail$ in the production decs : dec decs.

By comparing the results of the various attribute grammars, the students can develop a feel for different semantic rules before they begin the task of implementation. Without this separation, students have a harder time understanding the impacts of these choices for scoping rules, and have more trouble with the principle that semantics can (and should) be defined without reference to a particular implementation.
program : exp

exp : INT { }
  | ID { if (!find($1, $$.avail))
      EM_error("undeclared variable"); }
  | exp + exp { $1.avail = $3.avail = $$.avail; }
  | LET decs IN exp { $4.avail = shadow($2.decl, $$.avail);
                    $2.avail = $$.avail; }

decs : /* epsilon */ { $$.decl = Symbol_Table(); /* empty */ }
  | dec decs { $$.decl = fuse($1.decl, $2.decl);
             $1.avail = $$.avail;
             $2.avail = shadow($1.decl, $$.avail); }

dec : VAR ID : type := exp { $$.decl = Symbol_Table(ID, type); }

Figure 2. Finding Undeclared Variables via Synthesized and Inherited Attributes

Once students understand the possibilities for scoping rules and get a feel for where information is flowing in the AST, they are usually quick to grasp the implications for tree traversals: The C++ and Lisp scoping rules can each be processed in a single compiler pass (since they are expressed with L-attributed grammars [1, Section 5.4]); for the bi-directional scoping rule, we must make the full set of names declared in a given \texttt{let} available in that same \texttt{let}'s initialization expressions, for example by performing one traversal of each set of declarations (or of the whole AST) just to compute the \texttt{decl} attribute, and a later traversal to compute \texttt{avail}.

We continue to use attribute grammars as our primary instructional tool when we move on from scoping rules to type checking. We extend our grammar to allow literal string expressions and string-typed variables, introduce a \texttt{type} attribute for each declared variable and expression, and let the students try out the definition by hand to get a feel for the semantics. We then consider the implications of these definitions for our AST traversal, particularly in light of the fact that Tiger combines static type checking with implicit variable types in declarations (the code in Figure 1 would be legal if we had initialized \texttt{a} to \texttt{1} rather than \texttt{b-1}, even without the \\
``: int'' type specifications). If all declarations included an explicit type, we could perform three separate traversals of the AST, computing variables \texttt{declared} (including their types), variables \texttt{available} (including types), and \texttt{types} of expressions, in that order. However, for an implicitly-typed variable, we must know the \texttt{type} of the initializing expression to compute the type of a variable.

This observation serves as a springboard to discuss more interesting options for tree traversals. One approach (which we recommend for the student lab projects) involves a single traversal of the AST to compute several attributes (\texttt{decl}, \texttt{avail}, and \texttt{type}), since Tiger does not allow forward references to variables. A more general option would be to restructure the entire attribute traversal system so that the values of attributes on individual AST nodes are computed only as they are needed, and the AST traversal order is implicit in the attribute definition rules. This leads to a discussion of dynamic and static checks for circularity
of attributes [1, Chapter 5.10].

Once again, attribute grammars have let us introduce fundamental concepts in a way that is abstract enough to keep the students’ attention on semantics, but still concrete enough to let them try out each idea by hand. Once the students are equipped with both the formal definition and the intuition that comes from trying it on some examples, we can move on to discuss implementation.

3 ATTRIBUTE GRAMMARS IN THE LAB

Students begin their lab assignments by extending the attribute grammars discussed in Section 2. This is done in response to complexities of Tiger that we omit during lecture — Tiger uses two name spaces, one for variables and functions, and the other for named types. This can easily be addressed by creating two sets of attributes (one for declared and available variables, and the other for declared and available types). Tiger’s scoping rules are also more subtle than any of the three options described above: Variables can be used from the point of declaration onward, but functions and types allow forward references within an uninterrupted sequence of functions or of types. To implement this, students must produce a hybrid between the C++-style and bi-directional scope rules.

Once the students have extended their grammars, they must produce their implementations. Our main goal in these labs is to illustrate general lessons of software engineering as the students bridge the gap between attribute grammars and a general-purpose language. The remainder of this section gives a detailed description of the object-oriented approach we use with C++ in our labs, and briefly identifies issues relevant in other paradigms and languages (we discuss these briefly in lecture).

3.1 Object-Oriented Implementation In C++

Since our students have had the most experience with C++, and our faculty with the Bison [3] and Flex [5] tools, we use C++, Bison, and flex for lab projects (along with the Eclipse IDE [4] and its CDT plug-in). We provide students with C++ versions of many of Appel’s C files for the Tiger projects, plus a set of classes implementing an abstract syntax tree with a collection of node types (e.g., nodes for expressions such as let, variable uses, and arithmetic operations; nodes for declarations; etc.) that are all derived in various ways from a superclass AST_node_.

Since we wish to illustrate the value of code reuse, we encourage students to re-use a symbol table class they created in a prior course or choose one from a library, rather than writing one from scratch. Appel’s files also provide a data structure for representing types, so the implementation in the lab focuses almost entirely on creating functions to orchestrate the evaluation of attributes on the AST using the operations provided for symbol tables and types.

Within our hierarchy of AST node types, each attribute can be implemented as a virtual member function that computes the attribute value and, if necessary, stores it in the node. For example, returning to the simplified attribute grammar of Figure 2, the decl attribute could be implemented by creating a decl protected data member in class A_dec_ (a superclass of variable declarations and declaration lists) and providing the code shown in Figure 3.
Unfortunately, such code highlights the weaknesses of the object-oriented paradigm, not its strengths. For most nodes, most attribute-calculating functions will be essentially the same, simply passing information up or down the tree without changing it. Thus, the object-oriented style seems to force the programmer to create a large number of almost identical functions, and an overly verbose program results from an extremely tedious programming exercise. For example, we must create `synthesize_decl` functions for about thirty other classes in our AST hierarchy, and almost all of these simply call `synthesize_decl` on their children to make sure any declarations contained in an enclosed `let` expression are processed. This weakness is discussed in Appel’s Java edition [2, Chapter 4.2], where Appel explicitly rejects the use of the object-oriented style in favor of functions that select appropriate code for each kind of AST node via Java’s `instanceOf` method.

However, it is possible to retain the advantages of the object-oriented approach (such as ease of extension with new classes) and produce a few concise functions only for those attributes where interesting processing occurs, if we create a few general-purpose tree traversal functions. For example, to process synthesized attributes (i.e., those that involve information flowing only toward the root of the AST), we create a general-purpose `synthesize` function that takes two parameters describing the default action to take on an internal tree node (one to synthesize the child’s attributes and one to combine the results of children) and one parameter to describe the default result for a leaf node (a simple value). Figure 4 shows the definition of an example `synthesize` operation, for the AST node corresponding to arithmetic operations, and Figure 5 shows the use of `synthesize` to ensure that AST node types that do not override `synthesize_decl` will either return an empty symbol table (for leaf nodes)
or combine their children's symbol tables with the fuse operation (for internal nodes) — the latter makes the second function of Figure 3, and the tedious functions that traversed through nodes without decl attributes, unnecessary.

```cpp
some_type A_opExp_::synthesize(
    some_type (AST_node_::*get_what_from_children)(),
    some_type (*combine)(const some_type &, const some_type &),
    const some_type &default_value_for_leaves)
{
    some_type l_result = (_left->*get_what_from_children)();
    some_type r_result = (_right->*get_what_from_children)();
    return (*combine)(l_result, r_result);
}
```

**Figure 4.** Example synthesize Function

```cpp
Symbol_Table AST_node_::synthesize_decl()
{
    return synthesize(&AST_node_::synthesize_decl,
                      &fuse,
                      Symbol_Table());
}
```

**Figure 5.** Using synthesize to Implement synthesize_decl

Note that the synthesize operations would most naturally be expressed as templated virtual functions in C++, but lack of support in C++ for this combination of features has led us to use `#define` and `#include` instead. However, the largest macro we create is a three-line function declaration — all function bodies are in a “.cc” file that is #included once for each instantiation. So far, this approach has spared us the difficulties associated with debugging function bodies defined in long macros, that characterized the macros that were used to simulate templates in C++ before they were added to the language.

The synthesize and synthesize_decl functions are, of course, very similar to the visitor design pattern [6], and these labs can be used to lead into a general discussion of design patterns and object-oriented design.

For their lab projects, we provide students with the synthesize functions described above, as well as a collection of functions named pinherit for attributes in which information is inherited only from parent nodes (i.e., flows only away from the root of the AST), for all AST nodes except the one corresponding to the Tiger for loop. In one lab assignment, the students must identify undeclared variables, types, and functions. This can be accomplished by completing synthesize and pinherit functions for the A_forExp_ class and transforming the attribute definitions into a few calls to synthesize and pinherit. In the next lab assignment, students must perform type checking. For partial credit, this can be done in with straightforward uses of synthesize.
and `pinherit`, as long as the programs to be compiled do not use implicit type checking; for full credit, the students must build a traversal function of their own to handle implicitly typed variable declarations.

### 3.2 Aspect-Oriented Implementation

The implementation described above largely eliminates the need for redundant code that focuses on traversal of nodes where no interesting processing is occurring. However, as an approach to large-scale software design, it is still somewhat less than optimal, as the addition of the functions and data elements for each new attribute requires editing of the AST classes and re-compilation of everything that includes them. The need to edit existing source code is eliminated with a full implementation of the visitor design pattern, but this requires a static enumeration of all node types in each visitor class, and thus once again can limit our ability to add new node classes.

These limitations are not present in a language that fully supports Aspect-Oriented Programming [8]. This paradigm allows the definition of “aspects” such as our attributes in a way that provides dynamic dispatch without requiring changes to the node classes. However, time and space constraints limit us to this brief mention in both our course and this paper.

### 3.3 Imperative Implementation

We could also evaluate attributes without editing the AST classes by writing each attribute evaluator as a function that checks the type of the AST node and selects appropriate code with `if` or `switch`. We could either retain separate classes and use run-time type information (e.g. Java’s `instanceOf` method is recommended by Appel), or write old-style C code with a single class (or `struct`) with a `kind` field of an enumerated type listing all kinds of nodes.

However, this approach would still force us to edit the AST class(es) if we wish to add new attributes that are stored in the AST nodes, and of course the static enumeration of the kinds of attributes in each function would make it difficult to add new node classes.

Our students try out this approach and an object-oriented approach without general-purpose traversal functions in the Programming Language Concepts course that serves as a prerequisite for Compiler Design. Thus, by the end of the two-course sequence, they have ample experience on which to base an informed comparison of these approaches.

### 3.4 Pure Functional Implementation In Haskell

Attribute grammars can be translated into lazy function definitions in a way that is straightforward for those who are comfortable with extensive use of lazy evaluation, as shown by Turner [13] and summarized and translated into Haskell by one of our students [9]. With this approach, the functional language system automatically produces the demand-driven traversal to compute the attribute values on the AST, as discussed at the very end of Section 2. The simplicity with which this approach solves this otherwise complex problem can be
seen as a strong argument for the adoption of pure functional programming. Unfortunately, our students have not had sufficient experience with writing and debugging such code, so we have not chosen to pursue this approach in lab (though it is discussed in lecture).

4 RELATED WORK

Many compiler textbooks discuss attribute grammars; [1] is a well-known example. However, almost all limit their use of attribute grammars either to only synthesized attributes, which are well supported by compiler construction tools such as YACC and Bison, or to L-attributed grammars, which are supported to a lesser degree. Other flow of information around the AST is done with ad-hoc traversal code.

We know of only one compiler textbook that integrates more general attribute grammars throughout the course: Pittman and Peters’ “The Art of Compiler Design” [10]. However, this text does not emphasize the implementation of attribute grammars in traditional languages (except in the last chapter), and thus would not motivate many of the discussions we wish to have in this course (as listed in Section 3).

5 CONCLUSIONS

We have found that the use of attribute grammars in lectures, combined with implementation in a traditional language during lab projects, reinforces many of the central lessons we wish to teach in our Compiler Design course. The introduction of attribute grammars in lecture continues our focus on the use of special-purpose languages and specifications, which began as we studied the interplay of theory and practice in scanning and parsing. Student lab exercises combine opportunities to use this concise high-level framework with more traditional lessons in large-scale programming in a familiar language. Perhaps most importantly, the students spend a large fraction of their time learning to make connections between concise high-level descriptions and their programs.

This paper has focused on the examples that we use to introduce non-trivial attribute grammars, but there are opportunities to employ this approach (and the general-purpose traversal functions we provide) throughout the later labs, including those covering Sethi-Ullman register allocation [11] and stack frame layout. Pittman and Peters [10] also discuss the natural connection between dataflow analysis for structured programs and attribute grammars.

The files we have created to support this project are available from the author’s web site.

Bibliography


