The Reuse of Grammars with Embedded Semantic Actions

Terence Parr
University of San Francisco
San Francisco, CA USA 94117
parrt at cs.usfca.edu

Abstract

Reusing syntax specifications without embedded arbitrary semantic actions is straightforward because the semantic analysis phases of new applications can feed off trees or other intermediate structures constructed by the pre-existing parser. The presence of arbitrary embedded semantic actions, however, makes reuse difficult with existing mechanisms such as grammar inheritance and modules. This short paper proposes a mechanism based upon prototype grammars that automatically pushes changes from prototypes to derived grammars even in the presence of semantic actions. The prototype mechanism alone would be unsuitable for creating a new grammar from multiple pre-existing grammars. When combined with grammar composition, however, the prototype mechanism would improve grammar reuse because imported pre-existing grammars could be altered to suit each new application.

1. Introduction

Formal grammars lie at the heart of many important applications, including source code rewriting systems and program comprehension tools. Because many applications deal with the same language, being able to reuse a common syntax specification with different semantics provides a number of advantages. All applications would recognize exactly the same language, bug fixes to the language specification would propagate automatically to all dependent applications, and of course development and testing effort would decrease with reuse.

While the advantages are obvious, the mechanism for grammar reuse is not so clear. To go beyond syntax checking, grammars must have some way to specify the translation or interpretation logic (the semantics). Unfortunately, the act of specifying the semantics can lock a grammar into one specific application since the grammar is often modified to suit (e.g., programmers often want to embed unrestricted semantic actions). Reusing this grammar then becomes a serious problem even though it might be needed verbatim in another application but with different semantics.

In principle, the best solution is simply to decouple the syntax and semantics specifications so programmers can treat syntax specifications just like libraries. Many source-to-source translation tools take this approach. Term rewriting tools such as SDF+ASF [9] and Stratego [1] work with concrete syntax rewrite patterns feeding off of an internal intermediate representation implicitly-generated by a parser derived from the syntax specification. Stratego also allows tree rewrite rules that feed off of abstract syntax trees (ASTs) constructed via AST construction instructions embedded in the syntax grammar. Given an AST-producing parser, an application can also isolate all of the semantics in one or more tree walking phases. ANTLR [8] provides tree grammars with rewrite rules and integrates the StringTemplate [7] template engine.

This decoupled approach only supports verbatim syntax grammar reuse when the new application needs the exact same language (and same AST structure, if using AST rewrites or walking). To support changes to the recognized syntax and associated AST structure, tool metalanguages must support something akin to grammar includes, inheritance, or modules. For example, ANTLR v2 supported single grammar inheritance where rules in a *subgrammar* could override the equivalent definitions in a *supergrammar* to alter the recognized input language and resulting AST.

Unfortunately, a programmer's world is not always so "clean." Programmers often want to embed arbitrary semantic actions all over an existing grammar or tweak the actions already there. Existing mechanisms for grammar reuse do not deal well with arbitrary semantic actions. Consequently, programmers often copy the pre-existing grammar wholesale and then make their changes. The only thing wrong with this approach is that bug fixes to the original grammar are not propagated to the derived grammar.

Some tools avoid the issue of reusing grammars containing arbitrary semantic actions by simply disallowing arbitrary actions altogether. That is fine for pure source-to-source rewrite applications but is not suitable for applica-

tions or application components whose end goal is an internal data structure. Consider the humble task of reading in a configuration file to initialize objects in memory or the complicated task of reading in a program to build indexes and other program comprehension data structures. There is no escaping the need to perform semantic actions in a general purpose programming language. Programmers often rely on less formal systems such as ANTLR because they can embed arbitrary actions within grammars, at the cost of entangling syntax and semantics specifications.

This paper proposes a prototype-grammar mechanism for reusing grammars in the face of arbitrary actions and describes ANTLR's simple grammar composition mechanism that allows extremely large grammars to be broken into reusable, manageable chunks.

2. How Programmers Use ANTLR

Before discussing grammar reuse issues, a brief summary of ANTLR and how programmers use it is in order. ANTLR is an $LL(*)$ recursive-descent parser generator that accepts lexer, parser, and tree grammars. A tree grammar describes the well-formedness of the trees emanating from a parser. ANTLR has a unified metalanguage and uses the same code generation templates for all three grammars. The only difference between a lexer, parser, and tree walker is the kind of input symbol (character, token, or tree node).

ANTLR's $LL(*)$ parsers replace the k-symbol lookahead prediction mechanism in a conventional LL parser (acyclic DFAs recognizing fixed length lookahead languages) with cyclic DFAs capable of seeing past any common left-prefixes of alternative productions. For example, the following grammar fragment's alternative productions have arbitrarily-long left-prefixes in common.

```
// simplified Java declaration rule; e.g.,
// "public static int i;"
// "public static int f() {...};"
decl : variable_modifier+ variable
     | function_modifier+ function
     ;
```

Rule `decl` is not $LL(k)$ for any fixed k because of the positive closure ("+"). An $LL(*)$ parser predicts the alternatives using a DFA that spins past the modifiers and identifier to the symbol beyond (";" or "{"), which uniquely predicts whether a variable or function is approaching. Surprisingly, there is also no strict ordering between $LL(*)$ and $LR(k)$ since `decl` is $LL(*)$ but not $LR(k)$. An $LR(k)$ parser would have a reduce-reduce conflict between rule `variable_modifier` and `function_modifier` upon seeing any modifier in common.

For non-$LL(*)$ grammars, the backtracking option allows ANTLR to accept any non-left-recursive grammar, giving it the power of a PEG [4]. Partial parse result memoization guarantees linear parse times.

Programmers typically use ANTLR to generate one or more components of a larger application rather than viewing an ANTLR specification as stand-alone end-to-end solution. For simple applications such as configuration or data file processing, a straightforward combined parser and lexer grammar with embedded actions usually works well. For more complicated tasks, the parser builds an AST that is then walked by either a recursive hand-built algorithm or an ANTLR tree grammar. Often tasks are broken down into multiple tree walks that annotate the tree, populate ancillary data structures such as symbol tables or use-def chains, and possibly tweak the tree structure itself. The phases share a common tree grammar but need different semantic actions.

3. How Programmers Reuse Grammars

"*Is there an ANTLR grammar for language X?*" So begins one of the most common posts to the ANTLR interest mailing list. Clearly programmers reuse grammars, but how exactly do they benefit from pre-existing grammars? What precisely do they do with them? From watching the mailing list over the years, programmers are:

- cobbling together new grammars by combining pieces of other grammars; e.g., copying rules for arithmetic expressions or copying common lexer rules for identifiers, whitespace, and numeric literals
- combining complete grammars to embed one grammar in another; e.g., Java within HTML or SQL within C
- deriving new variants of an existing language by modifying a pre-existing grammar; e.g., adding GCC extensions to C or vendor specific statements to SQL
- traversing trees generated by pre-existing parser; e.g., program comprehension tools or `lint`-like program analyzers
- copying and modifying semantics of a pre-existing grammar, leaving the language unchanged; e.g., changing symbol table management actions to record more information about the program entities or tweaking a pretty printer

It's clear that there is a lot of cut-and-paste going on. The grim reality is that there are few opportunities to reuse grammars verbatim in non-rewrite systems. Because the result of recognition is a set of internal data structures, there must be actions within the parser or tree grammar. These actions cause trouble for existing grammar reuse mechanisms relying on grammar metalanguage constructs.

For example, ANTLR's first attempt to formally support the reuse of grammars (with and without semantic actions) was a grammar inheritance mechanism, in v2, implemented as an "include." While grammar inheritance works well in a few circumstances there are two problems with it: the subgrammar often requires changes to supergrammar rules and inheritance proves to be a blunt instrument for altering actions interspersed among the rules of the supergrammar.

In discussions with Ari Steinberg, an ANTLR user at Embarcadero Technologies building very large SQL grammars, he pointed out that supergrammar refactoring is sometimes necessary and doing so is painful. Paraphrasing our conversation, the initial structure of the supergrammar rules may be different than what the subgrammar needs to override later. Sometimes cutting and pasting is preferable to altering the supergrammar. Fine-grained control through rule inheritance often forces the introduction of very small and unnatural grammar rules. Morever, with lots of subgrammar overrides, the overall behavior of the generated parser can be difficult to figure out. Changing the supergrammar also risks silently altering the language recognized by the application.

Semantic actions also present problems for inheritance. In discussions with Chris Recoskie, IBM Eclipse C/C++ development tools team lead, he said:

> *The one thing that sort of got in the way of [grammar reuse] in ANTLR 2 was that if you overrode a rule, you had to override the action as well in its entirety.*

Recoskie suggested a nice way to refer to actions inherited from rules in the supergrammar but identifying which action within a rule would require labeling all of them. Also, it is unclear whether that would provide enough precision to properly alter the semantics of a supergrammar. One could, for example, need to tweak the inside of an action from the supergrammar rather than replace or reuse it.

Grammar reuse in the presence of semantic actions is not a lost cause, however. The next section proposes a solution.

4. Prototype-based Grammar Reuse

The key to grammar reuse may be building tools for manipulating grammars rather than designing metalanguage constructs such as inheritance. Consider Recoskie's ideal mechanism:

> *Ideally I'd like someone to have my C99 grammar sitting in their workspace, and if ever I bug fix my grammar, they just automagically pick up all of those changes when they build their grammar again, including all the semantic actions.*"

The philosophy of the ANTLR project is to formalize and automate what programmers do naturally and to present solutions that fit the way programmers think about a particular problem. In this case, being able to pick up changes from a grammar stored in a central repository smacks of source code revision control.

Reusing grammars based upon a revision control model would introduce the notion of a *prototype grammar*. New grammars would be derived from the prototype just like creating a new development branch in a revision system. Changes to the prototype would be merged into the derived grammar just like bug fixes merged from a trunk into a development branch. This approach simply co-opts the diff and merge tools of a revision control system to implement a kind of "live" cut-and-paste. Rather than recording the history of changes to a single file, this mechanism tracks changes between a prototype and any derived grammars.

This prototype grammar mechanism is not perfect, but would work well in three common situations: adding actions to a standard action-free grammar, tweaking actions in a pre-existing grammar to create a slightly different application, and using multiple versions of the same action-free tree grammar but with different actions.

All we need are two tools: `gderive` and `gsync`. `gderive` would make a copy of a prototype grammar and remember from which prototype the new grammar was derived. `gsync` would perform a three-way diff to discover how to alter the derived grammar to suit changes in the prototype grammar. For example, to begin a project the programmer would make a copy of the prototype via:

```
gderive Java.g MyJava.g
```

and later pick up changes from the prototype grammar via:

```
gsync MyJava.g
```

Grammar rules would be compared separately from the actions. One could opt to update just the rules or both rules and actions. As long as the derived grammar had not drifted too far away from the prototype, even merging actions would be manageable, just as it is when merging differences in a source code file.

To illustrate one of the best opportunities for prototype grammar reuse, consider ANTLR v3 itself. ANTLR's metalanguage parser constructs an AST representing the input grammar and then builds internal data structures using three passes over the AST: assigning token types, defining rules, and building NFAs (for use in constructing lookahead sets). An optional tree walk can print out the grammar without actions and a final code generation pass emits source code for the recognizer. In total, there are five essentially identical tree grammars, all with different semantic actions.

During initial development and when extending ANTLR's metalanguage, changes to the AST structure

required changes to all five tree grammars. These changes were propagated manually, often introducing tree grammar errors or slight inconsistencies. Using a single prototype tree grammar, changes could have been propagated quickly and correctly via:

```
gsync types.g define.g nfa.g print.g codegen.g
```

`gsync` would examine the grammar underneath the actions and, because the prototype grammar has no actions in this case, it could have easily folded in changes without disturbing actions. That is not to say that changes in a grammar cannot affect the actions. A change in rule element order could break actions so programmers would still need to peruse updated grammars and rerun unit tests.

Repeatedly making changes to those five tree grammars was very frustrating and, if the prototypical grammar concept has merit, why didn't we simply use `diff3`? The reason is that, fundamentally, we need to compare grammar structures (ordered trees) not unstructured plain text. Plain text oriented tools cannot separate grammars from the embedded actions. For example, `ID+` in a prototype grammar might be altered in a derived grammar to have an action:

```
( ID {names.add($ID.text);} )+
```

A plain text diff tool would announce a difference here, but a tool that compared structure could ignore the action and see `ID+` and `(ID)+` as identical in structure.

Research colleague Kay Röepke is working on a tree diff and merge tool inspired by Tancred Lindholm's 3DM tool for XML [6]. Lindholm builds off of the ordered tree difference work of Chawathe *et al* [2] among others. `gderive` and `gsync` are straightforward once we can compare metalanguage structures. The tree diff algorithm would not necessarily even need to be super efficient as grammar bug fixes and other updates occur infrequently.

The prototype grammar mechanism described in this section is only applicable when deriving a new grammar from a single prototype grammar. What happens when an application needs multiple derivations levels (analogous to multiple levels of grammar inheritance), needs to derive a grammar from multiple prototypes, or needs to break up a single extremely large grammar into more manageable chunks? Addressing these issues requires an additional mechanism, one that uses the grammar metalanguage, as discussed in the next section.

5. ANTLR Grammar Composition

Extremely large grammars are unwieldy just like any overly large single file a source code. Consider Steinberg's Oracle grammars written using the single inheritance model of ANTLR v2. The 8.1.6 grammar is 9400 lines, the 9i subgrammar is 4000 lines, and the 10g subsubgrammar is another 2000 lines. The 8.1.6 grammar is difficult to comprehend all at once. Worse is the fact that ANTLR generates a shocking 129k line Oracle 10g recursive-descent parser. This is too big for Eclipse [3] to load and is way beyond the limit of 64k lines imposed by Java debuggers. ANTLR's strength that it produces human readable parser becomes a serious weakness for very large grammars. Steinberg stressed in personal communications that the size of generated parsers is his primary issue with ANTLR v2's include-based grammar inheritance model.

One solution would be to have ANTLR generate multiple class definitions (instead of a single huge class) that, together, implement the parser. Each generated class would be small enough for Eclipse and the Java debugger to handle. One problem with this scheme is that semantic actions within the rules would likely break because they would no longer coexist in the same generated class. Instance variables and methods specified in a semantic action would no longer be visible to rules factored into other classes. It would be better to allow the programmer to break up the source grammar itself into logical units. ANTLR would then generate a separate class for each logical unit. In this way, both grammar and generated code would appear in manageable chunks.

Once a grammar is broken down into logical chunks, variations on the same language can share common rules more easily. Steinberg said that, for a single vendor's SQL, they could definitely organize grammars into several categories such as "*expressions, functions, DML statements, DDL statement, control statements, etc...*". Across vendors, unfortunately, he said that SQL syntax is different enough that coding differences between them is more complicated than simply copying what they need and starting fresh. Steinberg indicated that only lexer rules are shared between vendors; vendor specific lexers extend a common lexer grammar.

The upcoming ANTLR v3.1 introduces a grammar composition mechanism to simultaneously allow the logical organization of large grammars, provide more opportunities for grammar reuse, and allow the programmer to control the size of the generated classes.

A *root* grammar imports dependent grammars that it needs and, like traditional inheritance, the root grammar imports only those rules that are not overridden in the root grammar. In contrast, include-style mechanisms usually require programmers to manually specify what not to include in order to avoid rule definition collisions. Duplicate rule definitions imported from multiple delegate grammars get resolved in favor of the rule imported first. Imported grammars may, in turn, import their own grammars. ANTLR's composition mechanism is similar to that of PEG parser

generator *Rats!* [5]. *Rats!*'s composition is more general than ANTLR's as it allows parameterized modules, for example, but *Rats!* does not allow actions with arbitrary side effects.

ANTLR's composition mechanism behaves like multiple inheritance and is implemented with a delegation model. Given root grammar R and imported grammars A and B, ANTLR generates classes R, R_A, and R_B. Class R has two delegate pointers to instances of type R_A and R_B so that rules in R can access the imported rules. The delegate classes have delegator pointers back to the root so that, with a little bit of indirection, any rule can get to any other rule. There is no need to copy any rules from the delegate grammars into the root grammar.

Besides controlling the size of the generated parser, ANTLR must implement grammar composition with delegation because of semantic actions. Just as ANTLR could not automatically pull apart a single grammar for fear of breaking semantic actions, it cannot combine semantic actions outside of rules from multiple grammars. These actions could define class members with conflicting names; the resulting merged class would not compile. Using delegation allows ANTLR to avoid having to merge actions.

Every import of a delegate grammar results in the generation of a delegate parser not just the root parser. At first this might seem a waste, because if two root grammars import A, ANTLR will generate two different classes from A. In general, though, ANTLR must do this because a rule overridden in the root grammar often changes the prediction lookahead sets of rules in the imported grammar. Additionally, all token types must be consistent across rules from all delegate grammars. There is little choice but to regenerate classes for delegate grammars. Reuse is at the source grammar level not at the compiled binary level. Consider the following grammar for simplified Java declarations.

```
parser grammar JavaDecl;
type : 'int' ;
decl : type ID ';'
     | type ID init ';'
     ;
init : '=' INT ;
```

Another grammar, `Java`, can import `JavaDecl` to override rule `type`, thus, altering the prediction decision in rule `decl`:

```
parser grammar Java;
import JavaDecl;
prog : decl ;
type : 'int' | 'float' ;
```

From the root grammar, ANTLR generates code akin to the following (using the Java target).

```
class Java extends Parser {
  JavaDecl gJavaDecl; // delegate pointer
  void prog() { decl(); }
  void type() { match int or float }
  void decl() { gJavaDecl.decl(); }
  void init() { gJavaDecl.init(); }
}
```

and generates the following from the imported grammar.

```
class JavaDecl extends Parser {
  public Java gJava; // delegator pointer
  void decl() {predict alts using Java.type}
  void init() { }
}
```

There is no method associated with rule `type` in the `JavaDecl` delegate because the root grammar overrides it. More to the point, the lookahead DFA for rule `decl` differs because `type` has been overridden. The transition from s0 to s1 in Figure 1 reflects `Java`'s definition of `type` not `JavaDecl`'s definition. Notation "s3=>1" indicates that accept state s3 predicts alternative production 1.

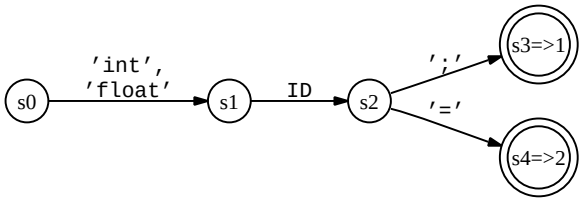

Figure 1. DFA predicting `decl` productions

Judicious use of delegate pointer indirection ensures that rule references are polymorphic: references to rule r in delegate grammar A see $R.r$ not $A.r$ if r is overridden in the root R. In this case, rule `decl` from grammar `JavaDecl` invokes rule `type` from grammar `Java` via the delegator pointer: `gJava.type()`.

As a more realistic test of grammar composition, the standard ANTLR Java grammar was broken into five subgrammars subsequently imported into a root grammar:

```
grammar Java;
options {backtrack=true; memoize=true;}
import JavaDecl, JavaAnnotations, JavaExpr,
       JavaStat, JavaLexerRules;
compilationUnit
    :  annotations? packageDeclaration?
       importDeclaration* typeDeclaration* ;
```

Instead of a single Java file with 21,591 lines generated from a single grammar, the composite grammar generates six smaller files, the largest of which is 5,287 lines.

Grammar composition allows programmers to break up a large grammar into manageable pieces or reuse multiple

pre-existing grammars, but composition does not obviate the need to alter pre-existing grammars to suit new applications. As discussed in Section 4, verbatim reuse of grammars is rarely possible. Programmers need to add rules, change rules, or add actions. The prototype grammar mechanism neatly solves many of these problems, but is unsuitable when multiple grammars must be combined. By combining the prototype grammar mechanism with grammar composition, programmers can simply derive new grammars to make the necessary refinements and then import them as delegates into a new composed grammar. Composed grammars are kept in sync by syncing any derived delegates from their respective prototypes and then running ANTLR again on the root grammar.

6. Summary

By decoupling the semantics specification of an application from the syntax specification, multiple applications can reuse all or part of the syntax specification. The syntax specification constructs a general AST which is then walked one or more times by an application to handle the semantics. This ideal situation does not always occur. Sometimes there is no pre-existing grammar for the desired language or a grammar exists but has arbitrary embedded semantic actions. Currently programmers cut, paste, and modify what they need from existing grammars. We need a formal way to deal with entangled syntax and semantic specifications.

This paper proposes a prototype grammar approach that operates in a manner similar to a revision control system. Programmers derive new grammars from existing grammars and then tweak the rules or actions. Changes and fixes to the prototype grammar are merged into the derived grammar using a tool based upon a structured tree diff. Prototype-based derivations would work well in situations that normally call for specialization of a supergrammar using single inheritance. It would also work extremely well for propagating AST structure changes to all dependent tree grammars, even in the presence of semantic actions. The prototype approach is not an earth shattering new formal mechanism, but has the advantage of being familiar to developers and, because it does not require metalanguage constructs, is applicable to any language tool.

Prototype-based grammar reuse breaks down when a grammar needs to pull elements from multiple grammars. ANTLR v3.1 introduces grammar composition that simulates multiple grammar inheritance implemented via delegation. By combining grammar prototypes and composition, programmers could more easily reuse pre-existing grammars because they could be altered before inclusion. The functional requirements and implementation of these mechanisms evolved through discussions with ANTLR users and observations from the ANTLR interest list.

7. Acknowledgments

Ralf Lämmel provided excellent feedback on a draft of this paper. Chris Recoski, Ari Steinberg, and Mike Kucera provided valuable information about the reuse of grammars in industry.

References

[1] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16. a language and toolset for program transformation. *Science of Computer Programming*, 2007. (To appear).

[2] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *ACM SIGMOD'96 International Conference on Management of Data*, pages 493–504, 1996.

[3] Eclipse homepage. http://www.eclipse.org.

[4] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL'04*, pages 111–122. ACM Press, 2004.

[5] R. Grimm. Better extensibility through modular syntax. In *PLDI'06*, pages 38–51. ACM Press, 2006.

[6] T. Lindholm. A 3-way merging algorithm for synchronizing ordered trees — the 3DM merging and differencing tool for XML. Master's thesis, Helsinki University of Technology, Dept. of Computer Science, Sept. 2001. http://www.cs.hut.fi/ ctl/3dm/thesis.pdf.

[7] T. J. Parr. Enforcing strict model-view separation in template engines. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 224–233, New York, NY, USA, 2004. ACM.

[8] T. J. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, 2007. ISBN 0-9787392-5-6.

[9] M. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of Compiler Construction 2001 (CC 2001)*, LNCS. Springer, 2001.