Aster: Automatic Abstract Syntax

Shaon Barman
Supervising Professor: William R. Cook
Department of Computer Sciences
University of Texas at Austin

May 8, 2009

Abstract

Modern parser generators typically require two components: an abstract syntax data structure and a concrete syntax grammar specification. Usually the abstract syntax and the concrete syntax are closely related but are specified independently because it is difficult to distinguish what parts of the concrete syntax are associated with the abstract syntax. There are a few add-ons to parser generators, such as ANTLR, which allow the programmer to specify this relationship through annotations in the concrete syntax, but their scope is limited. In this honors thesis, I propose a new way of generating the abstract syntax data structure. The concrete syntax grammar is used as a base for the abstract syntax data structure. The grammar undergoes a series of transformations, where each transformation analyzes the grammar and removes excess information, such as precedence, associativity, or literals used only to disambiguate potential parsings. The final result is an abstract syntax data structure. In addition, since each of these transformations can be linked together, parser actions can be inserted into the original grammar so that an abstract syntax tree is created when the generated parser is run.

1 Introduction

In creating a new language, programmers must typically define a concrete syntax, an abstract syntax, and a parser to recognize concrete syntax and build the corresponding abstract syntax tree. Parser generators are frequently used to build the parser and provide support for specifying the concrete syntax. Programmers add actions to the concrete syntax to create the corresponding abstract syntax, which is defined externally. Creating the abstract syntax and especially the actions, is often tedious and error-prone. Below is an example of the process a programmer would go through.

```
void ForStatement() : {}
{
  "for" "(" ( ForInit() )? ";"
  ( Expression )? ";" ( ForUpdate )? ")"
  Statement
}
```

The concrete syntax grammar for a *for statement* in JavaCC format.

```
public class ForStatement {
  public ForInit init;
  public Expression expression;
  public ForUpdate update;
  public Statement statement;
}
```

The Java class which represents the abstract syntax for a *for statement*.

```
ForStatement ForStatement() :
  {ForStatement f = new ForStatement(); ForInit fi;
   Expression e; ForUpdate f;  Statement s;}
{ "for" "(" ( fi=ForInit() {f.init = fi} )? ";"
  ( e=Expression() {f.expression = e;} )? ";" ( f=ForUpdate() {f.update = f;} )? ")"
  s=Statement {f.statement = s; return f;}
}
```

The concrete syntax grammar for a *for statement* in JavaCC format with actions inserted.

The actions embedded into the concrete syntax grammar will create the abstract syntax tree. The goal of this project is to automatically create the Java classes for the abstract syntax given a concrete syntax grammar without actions. Additionally, the grammar actions to generate the abstract syntax tree will also be automatically inserted into the concrete syntax grammar.

As can be seen in the above grammar, the concrete syntax has terminals in it which allows it to build a parser that can parse the input. The abstract syntax Java class just contains the nonterminals found in the production. Also, the optionals are removed in the abstract syntax.

There are several existing systems for automating the process. ANTLR supports annotations that indicate what parts of the concrete structure should be included in an abstract structure. It uses a generic AST format, and the annotations are limited. Some language environments have extended grammar notations that specify both concrete and abstract syntax. Wile considered the general problem, but did not formalize his rules or generate actions [2].

In this paper we describe a process that automatically derives abstract syntax from a concrete syntax, and also generates the actions necessary to create abstract syntax during parsing. The intuition behind our approach is that a logical abstract syntax can be derived from concrete syntax by removing all the extra information associated with concrete representations, especially precedence, associativity and syntactic markers of structure. This process has been implemented in Aster. In particular, Aster uses JavaCC grammars to automatically create an abstract syntax data structure and to output a JavaCC grammar complete with actions.

The difficulty in removing all of the extra information is distinguishing the use of the parts of the production. We address this through a series of transformations; each targets a specific type of information found in the grammar.

2 Grammars and Abstract Syntax

Grammars are given in the following form:

$$\begin{aligned} Grammar &:= \overline{n := p} \\ p \in Pattern &:= t \parallel p* \parallel p \mid p \parallel p? \parallel p+ \parallel p_1 p_2 \\ t \in Terminal & \\ n \in Name & \end{aligned}$$

A grammar is composed of a list of productions. $\overline{n := p}$ indicates a list of $n := p$, each of which is a production or rule. A production is a name and an associated pattern. There are several forms of patterns. t indicates a terminal symbol, such as ";", $p*$ indicates a repetition of zero or more times of another pattern, $p \mid p$ indicates a choice between two patterns, $p?$ indicates zero or one instances of the pattern, $p+$ indicates a repetition of one or more times and $p_1 p_2$ indicates a sequence of patterns. Also because we are defining a grammar for grammars, we must distinguish between metasymbols and terminals in the grammar. $\parallel$ is a metasymbol signifying a choice in the grammar for grammars.

We will be using a small grammar throughout this paper to demonstrate how the transformations work. The grammar in its original form is given below. Notice that the grammar carries information such as precedence and grouping in it.

```
ExpressionList := (Expression ';')*
Expression := Term ('+' Term)*
Term := Factor ('*' Factor)*
Factor := ID | NUM | '(' Expression ')'
```

We will also introduce the notion of *holes* in order to define the transformations. A hole is any pattern with exactly one subpattern missing. $H[x]$ is defined as the pattern with the missing subpattern filled in by x. By defining a hole, it is possible to distinguish all specific patterns within any grammar.

$$H \in Hole := \bullet \parallel H* \parallel H \mid p \parallel p \mid H \parallel H? \parallel H+ \parallel pH \parallel Hp$$

2.1 Actions

Actions are a way of interleaving native code in the generated parser. When a parser matches the pattern in a production with an input sequence, it also runs the associated actions in that pattern. Actions can be composed of any code, but they are unique in that they can capture the return values of nonterminals and lexical tokens. Parser actions are embedded throughout each production so that the abstract syntax tree is created as the parser goes through the input.

In Aster, actions are represented as Java code since the generated parser is written in Java. To simplify the renaming of variables, we use a simplified abstract syntax of Java to represent an action. An action can be placed before and after every pattern. In addition, nonterminals and tokens can have assignment actions placed on them, which will capture the return value of the nonterminal or token into a Java variable. Variables in one action can be referenced in other actions, as long as the variable remains in scope.

2.2 Normal Form and Classes

In order to create Java classes from a grammar, we define a normal form for grammars. The Java classes created from a grammar will form an abstract syntax data structure for that grammar. Any grammar can be expressed in normal form, and we give a transformation below which converts any grammar into normal form.

Normal form also gives an easy way to create the classes that form the abstract syntax data structure. There are three types of productions: sequences, choices of terminals and choices of nonterminals. A sequence production corresponds to a Java class being created in the abstract syntax data structure, with each element in the sequence becoming a field in the class. A choice of terminals production corresponds to an enum. And a choice of nonterminals corresponds to an interface, with each of the possible nonterminals implementing that interface.

$$
\begin{aligned}
Normalized &:= \overline{n := p} \\
p \in Pattern &:= e \parallel s \parallel c \\
e \in Enum &:= t_1 \mid ... \mid t_k \\
s \in Superclass &:= n_1 \mid ... \mid n_k \\
c \in Class &:= f_1 ... f_k \\
f \in Field &:= x \parallel x? \parallel x* \parallel x+ \\
x \in Type &:= n \parallel t \\
t \in Terminal & \\
n \in Name &
\end{aligned}
$$

3 Transformations

Each of the following transformations analyzes the concrete syntax to remove excess information. Since the transformations modify the grammar by removing information, the parsers that are generated from them cannot parse the same input as the original grammar. This is irrelevant since the transformed grammars are only used to create the abstract syntax data structure. The final grammar with actions will be based on the original grammar.

The order of the transformations can be changed except for the normalization transformation which must be done last. This requirement is placed because normalization puts the grammar into a form that is hard to analyze, although the normalization process can be reversed.

3.1 Remove Excess Literals

Some literals in the concrete grammar are only used to disambiguate parsings. These literals may be obvious to the programer, but are difficult for the transformations to recognize. In order to allow the other transformations to function correctly, Aster requires the programmer to indicate which literals should be ignored. Usually, these include literals like '(', ')', '[', ']', etc.

Example grammar after removing excess literals.

```
ExpressionList := ( Expression )*
Expression := Term ( '+' Term )*
```

```
Term := Factor ( '*' Factor )*
Factor := ID | NUM | Expression
```

3.2 Recursion Elimination

In order to obtain the optimal abstract syntax, we must remove the explicit precedence and associativity in the grammar, since they will be encoded implicitly in the abstract syntax tree itself. The key insight is that precedence and associativity are created when productions refer to each other in a cyclic manner. To find such productions, we create a production graph, where the edge (A, B) is present if A can go to B (ie A is a choice with B as a potential choice or A is a sequence and all other elements in A can go to nothing). Once the cycles are found, one production in the cycle is chosen to represent the cycle. That production then replaces the nonterminals for all other productions in the cycle and that production becomes a choice, with all of the patterns of the other productions in the cycle becoming the possible choices.

$$\frac{(n_0 = p_0 \ldots n_k = p_k) \in G \qquad derives(n_i, n_{(i+1)\%k}) \qquad n \text{ fresh}}{G \Rightarrow [n_i \mapsto n]G}$$

where substitution is defined as,

$$\begin{aligned}
[n \mapsto n']n'' &= n'' \\
[n \mapsto n']n &= n \\
f(n := p) &= f(n) := f(p) \\
f(\{a_0, \ldots, a_n\}) &= \{f(a_0), \ldots, f(a_n)\} \\
f(A \mid B) &= f(A) \mid f(B) \\
f(A*) &= f(A)* \\
f(AB) &= f(A)f(B) \\
f(A?) &= f(A)?
\end{aligned}$$

Example grammar after eliminating recursion.

```
ExpressionList := ( Factor )*
Expression := Factor ( '+' Factor )*
Term := Factor ( '*' Factor )*
Factor := ID | NUM | Expression | Term
```

3.3 Remove Repetitions

One pattern which is commonly used is $A := AB*$. This pattern can be used to express lists and operators. But in this form, it is difficult to extract information. So we convert this form into a form which parses the same language, but is easier to extract information from.

$$\frac{n = n(p_1*) \in G}{G \Rightarrow G - \{n = n(p_1*)\} + \{n = np_1\}}$$

$$\frac{n = (p_1*)n \in G}{G \Rightarrow G - \{n = (p_1*)n\} + \{n = p_1 n\}}$$

Example grammar after replacing repetitions.

```
ExpressionList := ( Factor )*
Expression := Factor '+' Factor
Term := Factor '*' Factor
Factor := ID | NUM | Expression | Term
```

In the example above, it may not seem at first that the production matches the required pattern since *Expression* and *Factor* are different, and similarly *Term* and *Factor* are different. But since *Factor* can go to *Expression* without consuming and lexical tokens, we can say that they are equivalent. Likewise *Factor* is equivalent to *Term* and therefore both productions match the required patterns.

3.4 Normalization

Normalization takes any grammar and modifies it into a standardized form. From this standardized form, abstract syntax classes will be created. Normalization also plays an important part in the transformation process because many of the other transformations require the input grammar to be in normal form.

$$\frac{(n = p) \in G \qquad p = H[p'] \qquad p' \rightarrow p'', R}{G \Rightarrow G - \{n = p\} + \{n = H[p'']\} + R}$$

$$\frac{p_1 \notin N \qquad n \text{ fresh}}{p_1 \mid p_2 \rightarrow n \mid p_2, n = p_1}$$

$$\frac{p_1 \notin N \qquad n \text{ fresh}}{p_1 p_2 \rightarrow n p_2, n = p_1}$$

$$\frac{p \notin N \qquad n \text{ fresh}}{p* \rightarrow n*, n = p}$$

$$\frac{p \notin N \qquad n \text{ fresh}}{p? \rightarrow n?, n = p}$$

Example grammar after normalization.

```
ExpressionList := ( Factor )*
Expression := Factor '+' Factor
Term := Factor '*' Factor
Factor := Factor1 | Factor2 | Expression | Term
Factor1 := ID
Factor2 := NUM
```

4 Code Generation

The Java code which defines the abstract syntax is created from a normalized grammar. There are three forms of productions: sequences, choices of terminals and choices of nonterminals. A sequence production is transformed into a class, with each element in the sequence becoming a field in the class. A choice of terminals production is transformed into an enum. And a choice of nonterminals is transformed into an interface, with each of the possible nonterminals implementing that interface. In addition, if a production simply goes to a single nonterminal, repetition, or optional, we consider that to be a sequence and construct a new class from that production with the given nonterminal, repetition, or optional being the only field in that class.

5 Inserting Initial Actions

Once the grammar is in a normal form, actions can be easily inserted into the normalized grammar. The classes, interfaces and enums that are created during the code generation phase are used during this phase. Each production is given the return type of the class, interface or enum that was created from it. For sequence productions, an instantiation action is placed at the beginning of the pattern. Each member of the sequence is either a nonterminal, terminal, or repetition. For nonterminals, a variable declaration action is placed before the pattern, a capture action saves the return value to a variable, and an assignment action stores the variable in one of the fields in the class. For a nonterminal choice production, a variable declartion action is placed before the choice. For each nonterminal in the choice, a variable capture action saves the return value and an assignment action saves the nonterminal value as the variable which is to be returned by the production. Similar code is inserted for the enum, except no capture actions are needed, only the action to return the selected enum is inserted.

Example production with initial actions.

```
ExpressionList ExpressionList() :
{
  { ExpressionList el1 = new ExpressionList();
    List<Factor> alf = new ArrayList<Factor>(); }
  (
    { Factor f1; }
    f1=Factor()
    { alf.add(f1); }
  )*
  { el1.f = alf;
    return el1; }
}
```

6 Reconstructing Actions

The final output of Aster is the original grammar with parser actions inserted. This grammar can generate a parser which will create the abstract syntax tree using the classes created during the

code generation phase. The grammar with actions is created by propagating the actions backward from the normalized grammar into the original grammar.

Once the actions are placed in the normalized grammar, they must be propagated to previous grammars. This is done using pointers created during each transformation between the pre-transform grammar and the post-transform grammar. Each transformation updates the pointers so that each pattern keeps a pointer to its equivalent pattern in the transformed grammar. When the actions need to be propagated backwards, a traversal is made of the pre-transform grammar. For each pattern in the grammar, the equivalent pattern is checked in the post-transform grammar. Since the post-transform grammar already contains actions, the pattern will copy over any actions associated with the equivalent pattern. For most transformations, the copying of actions is straightforward. But for the normalize and replace repetitions transformations, the propagation of actions is not one-to-one because these transformations change the structure of the patterns.

6.1 Normalization

Since new productions are created during normalization, the names of variables in the original production and the created production will not match. So, once the actions are propagated, a renaming of variables occurs.

6.2 Recursion Elimination

For those productions that are not modified, the actions are propagated through to the pre-transform grammar. For those actions that were edited, the actions are rearranged so that a new abstract syntax node is created in the repetition, and the old node is stored as a field.

6.3 Remove Circular Dependencies

The actions are passed to the pre-transform grammar using the pointers created during the transformation. But since each of the productions in the cycle can go to each other without using any terminals, each production needs to be able to pass through the correct nonterminal. This is done by adding conditional statements before the return value. These conditional statements check to see which variables have been set to a value and pass through a value if needed.

7 Evaluation

7.1 Infix Expression Grammar

The example grammar used through this paper is an infix arithmetic expression grammar with only two operators, multiplication and addition. When given into Aster, the loop from Expression ⇒ Factor ⇒ Term ⇒ Expression is identified so each production returns an object which inherits from interface Factor. The grammar given below is able to parse an input string with correct precedence and associativity, but the precedence and associativity are stored by the structure of the abstract syntax tree.

```
ExpressionList ExpressionList() :
{ ExpressionList el = null; java.util.ArrayList<Factor> alf = null; Factor f1 = null; }
{
```

```
  { el = new ExpressionList(); }
  { alf = new java.util.ArrayList<Factor>(); }
  (
    f1=Expression()
    { alf.add(f1); }
    ";"
  )*
  { el.f = alf; }
  { return el; }
}

Factor Expression() :
{}
{
  { Factor f1 = null; }
  f1=Term()
  (
    { Expression e = null; e = new Expression(); }
    "+"
    { Factor f3 = null; }
    f3=Term()
    { e.f1 = f3; }
    { e.f = f1; f1 = e; }
  )*
  { return f1; }
}

Factor Term() :
{}
{
  { Factor f5 = null; }
  f5=Factor()
  (
    { Term t = null; t = new Term(); }
    "*"
    { Factor f7 = null; }
    f7=Factor()
    { t.f1 = f7; }
    { t.f = f5; f5 = t; }
  )*
  { return f5; }
}

Factor Factor() :
{ Factor f8 = null; Factor1 f24 = null; Token t2 = null;
  Factor2 f23 = null; Token t4 = null; Factor f14 = null; }
{
```

```
  (
    { f24 = new Factor1(); }
    t2=<ID>
    { f24.id = t2.image; f8 = f24; }
  |
    { f23 = new Factor2(); }
    t4=<NUM>
    { f23.num = t4.image; f8 = f23; }
  |
    "(" f14=Expression()
    { f8 = f14; }
    ")"
  )
  { if(f14 != null) { return f14; } return f8; }
}
```

7.2 Java

Aster has successfully been run on a concrete syntax grammar for Java 1.1.

Table 1: Abstract Syntax for Java 1.1 Grammar

	Classes	Interfaces	Enums
Aster	119	24	18
Original*	135	23	18

*The original grammar has only gone through normalization

8 Related Work

8.1 Popart automatically generated ASTs

David Wile implemented a similar project using his Popart's language processing system [2]. Although his work deals with transforming grammars into an abstract syntax, he uses a heuristic based approach to this. In addition, the transformations he provides are not formally defined. Aster also deals with how to automate the introduction of parser actions into the original grammar, which is a topic not discussed in Wile's paper.

8.2 Antlr automatically generated ASTs

Antlr provides the ability to annotate the concrete syntax grammar in order to automatically generate the abstract syntax. This method allows the abstract syntax tree to be created, but can only specify a simple translation between concrete syntax and abstract syntax. In addition, adding the concrete syntax annotations is another burden for the programer.

8.3 Rats automatically generated ASTs

The Rats parser generator also allows for automatic abstract syntax tree generation [1]. Unlike Antlr, no user annotations are required. But the conversion from concrete syntax to abstract syntax is simple and does not try to take advantage of the excess information that Aster tries to remove.

9 Future Work

9.1 Token Elimination and Merging

To create a more compact grammar, tokens can be deleted and similar rules can be merged together. These two processes may seem distinct, but are inherently connected. A token is any literal or concrete symbol used in the concrete syntax grammar. We eliminate the token from the pattern if removing the terminal allows for the pattern to be distinguished amongst all other patterns that are linked to it through a choice that has the pattern as one of its options. Likewise, we merge two patterns together if their structure is equivalent except for the terminals, i.e. when we remove the terminals, the two patterns are the same. Because of this similarity the merge rule and the token elimination rule can be run simultaneously.

Provided below is the formalization of the rule and what the example grammar would be if it went through this transformation. I was unable to implement this transformation because of time contraints.

$$\frac{R = \{n_1 = p_1, n_2 = p_2\} \subset G \qquad p_1 = H[t_1] \qquad p_2 = H[t_2] \qquad insomealt(n_1, n_2, G) \qquad x \notin N}{G \Rightarrow G - R + \{n = H[x], x = t_1 \mid t_2\}}$$

$$\frac{\forall n_1, n_2 s.t. R = \{n_1 = p_1, n_2 = p_2\} \subset G \qquad p_1 = H_1[t_1] \qquad p_2 = H_2[t_2] \qquad H_1 \neq H_2 \qquad insomealt(n_1, n_2, G) \qquad x \notin N}{G \Rightarrow G - R + \{n = H[\epsilon]\}}$$

Example grammar if token elimination and rule merging was run on it.

```
ExpressionList := ( Factor )*
Expression := Factor ( '+' | '*' ) Factor
Factor := Factor1 | Factor2 | Expression
Factor1 := ID
Factor2 := NUM
```

10 Conclusions

Aster provides a way for programmers to automatically generate an abstract syntax data structure from a concrete syntax grammar specification. In most cases, the abstract syntax and concrete syntax are created together. But there are cases in which a programer is given a concrete syntax grammar and must create the abstract syntax. Aster provides a structured way for programmers to automatically generate the abstract syntax. In addition, Aster will insert the parser actions into

the concrete syntax grammar. By automating this process, the programmer is saved the tedious and error-prone process of manually inserting the actions into the concrete syntax grammar.

Aster could be further extended by creating more transformations which could minimize the grammar. One way of doing this is by creating a grammar transformation platform which would take in a series of grammar transformations. Grammar transformations can be represented as pattern matches and how the pattern should be changed, similar to a regular expression find and replace. The transformations in Aster could be written for this platform, along with new transformations. Having such a platform would allow for rapid development of new transformations and would allow for more complicated abstract syntax data structures to be created.

References

[1] R. Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM.

[2] D. Wile. Abstract syntax from concrete syntax. pages 472–480, May 1997.