

# List Resolution Techniques for Grammar Development

Crutcher Dunnavant  
University of Alabama  
crutcher@gmail.com

## ABSTRACT

Most popular parser generation tools work in terms of grammar rules which are not directly capable of providing list semantics. While the basic techniques for realizing list semantics are frequently re-invented, the literature has suffered from a lack of a solid collection of these techniques. This paper presents a collection of techniques for realizing various forms of lists as grammatically re-written forms using AST re-writes compatible with most parser generation environments. Also provided is a set of additional EBNF extension operators for specifying AST re-write actions.

## 1. INTRODUCTION

Using most parser generation tools, matching a list of elements requires the construction of a recursive non-terminal (left-recursive for LALR(1) grammars) which matches the various productions of the list, and has an epsilon production. Thus, in order to describe a list containing some arbitrary number of *As* and *Bs*, representable in ISO EBNF[10] as:

$$L = \{A \mid B\}$$

most parser generation tools would require the re-structured rule:

$$L = L A \mid L B \mid \epsilon$$

Unfortunately, given a symbol stream *A B A A*, this rule would yield the containment  $(L (L (L (L (L) A) B) A) A)$ , rather than the more useful  $(L A B A A)$ . It would likely be necessary to restructure this tree before applying semantic actions; and since matching lists of elements is virtually guaranteed to come up while writing a language evaluator, there is some real value in having a higher level model for dealing with this.

This paper defines transformations upon grammatical lists of various kinds which permit their realization in most parser generation environments. These transformations are themselves defined in terms of 4 simple parse-time AST re-write actions, which should be simple to implement in any parsing environment.

## 2. BACKGROUND

The yacc[3] family of parser generators won the parser generation space. This family is what is taught in compiler classes, and what books are written about. Many projects exist to re-implement the yacc generator in various environments. These include bison[5](C), Yacc++[9](C++), CUP

(or java\_cup)[6](Java), and JavaCC[8](Java). These tools work on input files structured to associate, and partially re-write, action code which is attached to the various rules of the grammar. For example, the rule:

$$Foo = A B C$$

would require a description for most of the yacc-alikes akin to:

```
A B C {
    ASTNode Foo = new ASTNode('Foo');
    Foo.addChild($1); /* For A */
    Foo.addChild($2); /* For B */
    Foo.addChild($3); /* For C */
    $$ = Foo;
    return Foo_symbol;
}
```

which would be re-written by the parser generator to define  $\$1$ ,  $\$2$ ,  $\$3$ , and  $\$\$$ . Working within these constraints, it is often difficult to provide the list semantics desired for a given grammar's AST.

Some of the more advanced parser-generation environments support lists directly. The meta-DSL tool smgn[4] flattens recursive lists, but does not handle delimited lists, and provides no means of controlling the nature of the collapse. SDF[1], which provides parser generation for full context-free languages, supports list constructs; as does GDK[2], which exists for debugging and/or reverse engineering grammars and provides a rich framework of grammar re-write algorithms.

However, most computer programming and configuration languages are not developed in robust grammar transformation environments, but are instead developed with vanilla yacc-alike parser generation tools. In such environments, it is necessary to perform custom grammar tweaks to achieve the list semantics desired.

## 3. AST RE-WRITE ACTIONS

Provided with a sufficiently abstract AST implementation, there are many useful tree re-writes which can be performed at production time. The list resolution transforms will be expressed in terms of four such actions. These actions are here defined as extensions to EBNF, in order to permit an abstract description of the list transforms. Figure 1 provides a graphical account of the functioning of the *preserve*, *token*, and *content* actions.

### 3.1 AST Action: Preserve

The default action taken for a symbol in a production is *preserve*. When a symbol is preserved in a production rule,

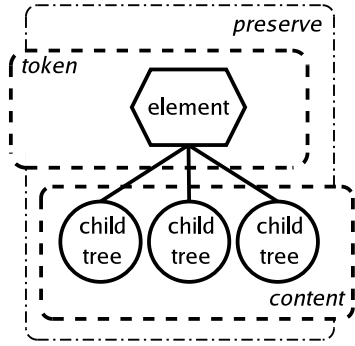


Figure 1: AST Re-Write Actions

the AST sub-tree rooted at that symbol is made a child of the sub-tree produced for the rule's non-term. For example, the rule:

$$Foo = Bar Baz$$

would, upon matching *Bar Baz*, produce a *Foo* node which contained the trees for *Bar* and *Baz* as it's children.

### 3.2 AST Action: Drop

The *drop* action, represented by  $\llbracket Symbol \rrbracket$ , is very simple. When a symbol is dropped in a production rule, the AST sub-tree rooted at that symbol is omitted from the sub-tree produced for the rule's non-term. For example, the rule:

$$Foo = \llbracket Bar \rrbracket Baz$$

would, upon matching *Bar Baz*, produce a *Foo* node which contained only the tree for *Baz* as a child.

### 3.3 AST Action: Token

The *token* action, represented by  $\lfloor Symbol \rfloor$  (as though *Symbol* is to be cut below), is more complex. When a symbol is tokenized in a production rule, only the root of the AST sub-tree rooted at that symbol is included in the sub-tree produced for the rule's non-term, while the symbol's sub-tree's root's children are omitted. For example, the rule:

$$Foo = \lfloor Bar \rfloor Baz$$

would, upon matching *Bar Baz*, produce a *Foo* node which contained as children the root of the tree for *Bar*, and the complete tree for *Baz*. Alternatively, if this rule matched the subtrees (*Bar a b c*) and (*Baz e f g*), it would produce (*Foo Bar (Baz e f g)*).

### 3.4 AST Action: Content

The *content* action, represented by  $\lceil Symbol \rceil$  (as though *Symbol* is to be cut above), is the complement of *token*. When the content action is performed on a symbol in a production rule, the children of the symbol's sub-tree's root are included as children of the sub-tree produced for the rule's non-term, while the symbol's sub-tree root itself is omitted. For example, the rule:

$$Foo = \lceil Bar \rceil Baz$$

would, upon matching *Bar Baz*, produce a *Foo* node which contained as children the root of the tree for *Bar*, and the complete tree for *Baz*. Alternatively, matching (*Bar a b c*) (*Baz e f g*) would produce (*Foo a b c (Baz e f g)*).

## 4. LIST TRANSFORMATIONS

**Note:**  $\{P\}$ — is defined in EBNF[10] as one or more occurrences of *P* (an arbitrary number of *P* except the empty set). This is confusing, and the equivalent form  $P, \{P\}$  will be used here instead.

### 4.1 Non-Delimited Lists

The simplest form of lists are repetitions of the same symbol over and over again without delimiters. There are two variations on this form, lists which may be empty, and lists which may not be. Given *I*, a single grammar symbol, these forms may be handled quite adequately with the following transformations (though for empty lists, *I* cannot itself have an  $\epsilon$  production without producing a shift-reduce conflict).

$$\begin{aligned} L = \{I\} &\Rightarrow L = \epsilon \mid \lceil L \rceil I \\ L = I, \{I\} &\Rightarrow L = I \mid \lceil L \rceil I \end{aligned}$$

Upon matching  $I \ I \ I$ , both of these rules would produce the sub-tree ( $L \ I \ I \ I$ ).

However, if it is necessary for the list to match multiple symbols, or collections of symbols, there are several distinct cases to consider.

#### 4.1.1 Grouped Lists

Often there are several possible matches for a list item, as in:

$$L = \{ Keyword \ equals \ Expression \mid Expression \}$$

and we wish to collect each item up, so that each group which makes up the list is collected together. This is a simple matter of re-writing the list to create an item non-terminal *I* which produces the various options. Letting  $\Phi$  be a series of production alternatives lacking  $\epsilon$ :

$$\begin{aligned} L &= \{\Phi\} \\ &\downarrow \\ L &= \{I\} \\ I &= \Phi \\ &\downarrow \\ L &= \epsilon \mid \lceil L \rceil I \\ I &= \Phi \end{aligned}$$

The non-empty variant of this transform is straightforward:

$$\begin{aligned} L &= \Phi, \{\Phi\} \\ &\downarrow \\ L &= I, \{I\} \\ I &= \Phi \\ &\downarrow \\ L &= I \mid \lceil L \rceil I \\ I &= \Phi \end{aligned}$$

If  $\Phi$  were *Keyword equals Expression*  $\mid$  *Expression*, upon matching (*I Expression*) (*I Expression*) (*I Keyword equals Expression*), these rules would produce the sub-tree ( $L \ (I \ Expression) \ (I \ Expression) \ (I \ Keyword \ equals \ Expression)$ ).

#### 4.1.2 Ungrouped Lists

If there are several possible matches for a list item, as in:

$$L = \{ A \ B \mid C \}$$

but the items are NOT to be grouped, then there are two transformations available, both which produce the same result. Either we re-write the tree at production time, using this transform:

$$\begin{array}{l}
L = \{\Phi\} \\
\downarrow \\
L = \{[I]\} \\
I = \Phi \\
\downarrow \\
L = \epsilon \mid [L] [I] \\
I = \Phi
\end{array}$$

or, letting  $\phi_1 \dots \phi_n$  be the various individual productions of  $\Phi$ , we re-write the grammar, using this transform:

$$\begin{array}{l}
L = \{\Phi\} \\
\downarrow \\
L = \epsilon \mid [L] \phi_1 \mid \dots \mid [L] \phi_n
\end{array}$$

These transforms are roughly equivalent; but the non-empty variant of the first transform:

$$\begin{array}{l}
L = \Phi, \{\Phi\} \\
\downarrow \\
L = [I], \{[I]\} \\
I = \Phi \\
\downarrow \\
L = [I] \mid [L][I] \\
I = \Phi
\end{array}$$

is so much more efficient than that of the second:

$$\begin{array}{l}
L = \Phi, \{\Phi\} \\
\downarrow \\
L = \epsilon \mid \phi_1 \mid \dots \mid \phi_n \mid [L] \phi_1 \mid \dots \mid [L] \phi_n
\end{array}$$

that it is usually wisest to use the first form of both the empty and non-empty list transforms, unless a pressing reason exists not too. If  $\Phi$  were  $A B \mid C$ , upon matching  $A B C$ , all of these rules would produce the sub-tree  $(L A B C)$ .

## 4.2 Non-Empty Delimited Lists

A more complex form of lists are delimited lists. Delimited lists have explicit separators between items. Beginning again with simple cases, let us take  $I$  and  $D$ , singular grammar symbols, and describe a transformation which is sufficient to handle a list of  $I$  delimited by  $D$ :

$$L = I, \{DI\} \Rightarrow L = I \mid [L] D I$$

Note that a shift-reduce conflict will exist if  $I$  and  $D$  both contain  $\epsilon$  productions. Upon matching  $I D I D I$ , this rule would produce the sub-tree  $(L I D I D I)$ .

There are several distinct variants which we may consider for delimited lists.

### 4.2.1 Dropped Delimiters

Suppose that we only wanted to keep the  $I$  symbols? In this case, we would describe the initial list, and the transform, as:

$$L = I, \{[D]I\} \Rightarrow L = I \mid [L] [D] I$$

Which, upon matching  $I D I D I$ , would produce the sub-tree  $(L I I I)$ . Using  $I$  as a list item non-terminal with dropped delimiters is the preferred way to parse most lists in most environments, though it is necessary to examine the empty version of this transform if empty lists are needed.

### 4.2.2 Token Delimiters

Suppose that we wished to keep a marker that a delimiter had been matched, but not it's content. In this case, we would describe the initial list, and the transform, as:

$$L = I, \{[D]I\} \Rightarrow L = I \mid [L] [D] I$$

Which, upon matching  $I (D a) I (D b) I$ , would produce the sub-tree  $(L I D I D I)$ .

### 4.2.3 Delimiters on Ungrouped Non-Empty Lists

When working with ungrouped lists, it is sometimes useful to maintain the delimiters in order to separate the groups at a later time. The following transform provides this:

$$\begin{array}{l}
L = \Phi, \{D, \Phi\} \\
\downarrow \\
L = I, \{D, I\} \\
I = \Phi \\
\downarrow \\
L = [I] \mid [L] D [I] \\
I = \Phi
\end{array}$$

If  $\Phi$  were  $A B \mid C$ , upon matching  $A B D C$ , these rules would produce the sub-tree  $(L A B D C)$ .

### 4.2.4 Variable Delimiters

The same collection technique used for grouping list items should be used in the case that multiple delimiters are permitted for a given list. The various delimiter options should be extracted into their own non-terminal, and appropriate re-write action should be taken on the resultant non-terminal if necessary. Letting  $\Phi$  be a series of production alternatives for the delimiter lacking  $\epsilon$ :

$$\begin{array}{l}
L = I, \{\Delta, I\} \\
\downarrow \\
L = I, \{D, I\} \\
D = \Delta \\
\downarrow \\
L = I \mid [L] D I \\
D = \Delta
\end{array}$$

## 4.3 Empty Delimited Lists

Delimited lists which can be empty require the most complex list transformations to realize. For starters, neither their items nor their delimiters can yield  $\epsilon$  productions without conflicting with the  $\epsilon$  productions of the list itself. Also, if the list is given an  $\epsilon$  production and a recursive production containing the delimiter  $D$ , then it would be possible to match  $D I$ , which is not what is intended. As a result, we will introduce an additional non-terminal,  $\hat{L}$ , which will be processed with the content action wherever it occurs. The following transformation provides for empty delimited lists matching against the item symbol  $I$  and the delimiter symbol  $D$ .

$$\begin{array}{l}
L = (I, \{D I\}) \mid \epsilon \\
\downarrow \\
L = \epsilon \mid [\hat{L}] \\
\hat{L} = I \mid [\hat{L}] D I
\end{array}$$

More complex item and delimiter groups can be realized using techniques from 4.1.1. *Grouped Lists* and 4.2.4. *Variable Delimiters*. All of the usual AST re-writes remain applicable with this transformation situation, and a complete enumeration of the various possibilities is not needed here.

## 4.4 A General Form

There is a general target form which, with the application of different AST re-write actions on  $I$  and  $D$ , is capable of realizing most list semantics. If  $\Phi$  and  $\Delta$  be a series of production alternatives lacking  $\epsilon$ :

$$\begin{array}{l}
L = \epsilon \mid [\hat{L}] \\
\hat{L} = I \mid [\hat{L}] D I \\
I = \Phi \\
D = \Delta
\end{array}$$

Because of its flexibility, this form is a good target form for automated tools which provide list semantics.

## 5. REFERENCES

- [1] Joost Visser, and Jeroen Scheerder *A Quick Introduction to SDF*. 2000.  
<http://homepages.cwi.nl/~jvisser/papers/sdfintro.pdf>
- [2] Jan Kort, Ralf Lämmel, and Chris Verhoef *The Grammar Deployment Kit*. In *Electronic Notes in Theoretical Computer Science 65 No. 3*, 2002.
- [3] John Levine, Tony Mason, and Doug Brown *lex & yacc*. O'Reilly, 2nd Edition, 1992. ISBN: 1-56592-000-7
- [4] Holger M. Kienle *Using smgn for Rapid Prototyping of Small Domain-Specific Languages*. In *ACM SIGPLAN Notices V.36(9)*, September 2001.
- [5] The GNU Project *Bison*.  
[www.gnu.org/software/bison/](http://www.gnu.org/software/bison/)
- [6] Scott E. Hudson *CUP Parser Generator for Java*.  
[www.cs.princeton.edu/~appel/modern/java/CUP/](http://www.cs.princeton.edu/~appel/modern/java/CUP/)
- [7] Don Yessick, and Joel Jones *Reinventing the Wheel Or Not Yet Another Compiler Compiler Compiler*. In *Southeast ACM Conference*, 2002.
- [8] *Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator*.  
[javacc.dev.java.net](http://javacc.dev.java.net)
- [9] *Yacc++ and the Language Objects Library*.  
[world.std.com/~compres/](http://world.std.com/~compres/)
- [10] International Organization for Standards *ISO/IEC 14977 - Extended BNF*. 1996.