Transforming Grammers and the Effects on Parse Trees

Ernest Kirstein

April 3, 2015

Implementing a parser often requires a grammar with certain properties. [1, 3] There are also special properties that a grammer can have which will make parsing more efficient. [2, 3]

But modifying a grammar can be expensive from a software engineering perspective. Using a non-intuitive form of the grammer will make implementing compilation more difficult since the parse trees produced using the modified grammer will be different than those one would expect from the natural, unmodified form of the grammar.

An especially expensive engineering problem would be modifying an already-implemented compiler to use a newly discovered grammar property. Implementing that change would also require developers to change both the parser and the code generator which uses the output of the parser.

I propose a radical change - a way to avoid that highly coupled design. Let parsers use a special modified grammer and let the compiler use the more natural form of the grammar. Impossible? No. Impractical? Maybe.

Condensed Version

Definition 1. *A* symbol *is an abstract building block used in formal grammar definitions.*

Definition 2. *A* string *is an ordered list of symbols.*

Definition 3. *A* context free production rule *describes how a symbol in a string can be replaced with another string.*

I will refer to the symbol which is replaced as the 'head' of the production rule. And I'll refer to the string which the 'head' is replaced with as the 'tail' of the production rule.

For example, a production rule $S \to abc$ might be applied to the string "$xvStnB$" to form the string "$xvabctnB$". It's 'head' would be the symbol S and it's tail would be the string "abc".

Definition 4. *A symbol may be* terminal *if it is not the head of any production rule.*

In practical examples, terminal symbols often correspond to tokens or printable characters, while non-terminal symbols correspond to more abstract concepts. I may refer to a string as terminal, in which case I mean string which is composed entirely of terminal symbols.

Definition 5. *A* context-free grammar is an order list of context free production rules.

Context free grammars, CFGs, can get quite complicated. Here's a simple example:

$$S \to a \tag{1}$$
$$S \to aS \tag{2}$$

Definition 6. *A* parse tree *is an ordered, rooted tree who's nodes correspond to the following of production rules in a context-free grammar. The leaves of a parse tree form the string which would be produced by following the production rules in any decending order of their application in the layers of the parse tree.*

Notation 1. *For brevity, nodes which correspond to a symbol which has not been transformed by a production rule will be writen using the symbol alone. Non-leaf nodes will be written using an ordered pair of symbol and corresponding set of child nodes. For example* $(S, \{a, b, c\})$.

Consider the example grammar (above). We can start a parse tree rooted at S: $t_0 = S$. Applying the second production rule to this root node, we get the parse tree: $t_1 = (S, \{a, S\})$ which corresponds to the string "aS". An application of the first production rule to t_1 would yield the tree $t_2 = (S, \{a, (S, \{a\})\})$ which produces the string "aa".

I may refer to parse tree's as terminal, in which case I mean that the parse tree produces a terminal string.

Definition 7. *The* language *of a context free grammar is the set of terminal strings can be produced by following rules in that grammar. If G is a CFG, then $L(G)$ shall refer to it's language.*

Theorem 1. *For every string in a CFG's language there exists some terminal parse tree which produces that string.*

Proof. Since every string in the language is produced by following the set of rules in some CFG, then it follows that the order in which those rules are followed may be used to construct a parse tree (by definition). And since that string is, by definition, terminal then so is that parse tree. □

Notation 2. *If G is a CFG then let $T(G)$ refer to the set of terminal parse trees that may be produced using the rules of that grammar.*

Definition 8. *If two CFG's produce the same language, then they shall be called* weakly equivalent*; noted by $G \approx G'$.*

Definition 9. *If two parse trees produce the same string, they shall be called* weakly equivalent*; noted by $t \approx t'$.*

Theorem 2. *If G and G' are weakly equivalent grammars then there exists a function, which shall be called a* tree-mapping function*, $f : T(G) \to T(G')$ such that $\forall t \in T(G) : t \approx f(t)$.*

Proof. Let G and G' be weakly equivalent grammars. For all strings, s, in $L(G)$ there exists a tree $t \in T(G)$ which produces s (by Theorem 1). Since $G \approx G'$ then $L(G) = L(G')$ so $s \in L(G) \implies s \in L(G')$. Since $s \in L(G')$ it must also be true that there exists a tree $t' \in T(G')$ which produces s. Then for every parse tree, $t \in T(G)$ there exists a $t' \in T(G')$ such that $t \approx t'$. Let f be the function which maps each of those t to that t'. □

Conclusion

Thus, if we have some calculable function which transforms a CFG G into some weakly equivalent CFG, G' and we can parse a string s into it's corresponding parse tree $t' \in T(G')$ then we know there exists a function which mapps t' into some tree $t \in T(G)$. If we can calculate that tree-mapping function of G' to G we can parse strings in G indirectly (via G'), without being able to directly parse strings in G.

References

[1] F. D. Lewis. Recursive descent parsing. `http://www.cs.engr.uky.edu/~lewis/essays/compilers/rec-des.html`, 2002.

[2] Stefano Crespi Reghizzi. *Formal Languages and Compilation.* Springer-Verlag London Limited, Italy, 2009.

[3] William M. Waite and Lynn R. Carter. *An Introduction to Compiler Construction.* HarperCollins College Publishers, New York, NY, 1993.