Recursive Descent Parsing

Ernest Kirstein

October 17, 2014

Historically, recursive descent parsers have been coded manually or else generated from a context free grammar specification. [4, 5] Coding a RDP manually is tedious, error prone, and difficult to maintain. Programatically generating a RDP from a grammar still isn't ideal - converting a context free grammar into a recursive decent parsable form may not preserve the *strong equivalence* of the grammar. And as a result, parse trees generated by that RDP will not be in the same form as they might appear in the initial (non-RD-parsable) grammar, which is often a more natural representation of the desired language [7].

In this paper, I hope to describe a useful variation on recursive descent parseing which addresses these problems. My system compiles a context free grammar specification into a RD-parseable grammar. A parser is generated from the RD-parseable gramar, and used to parse a input streams. The resulting parse trees are then transformed back into equivalent parse trees under the original grammar. (Figure 1)

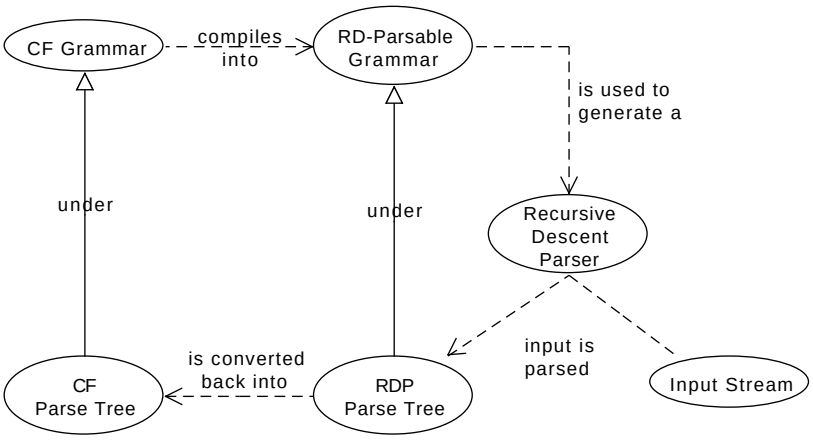

Figure 1: A high level overview of the process.

1 Recursive Descent without Syntax Diagrams

Introductory material by Dr. Lewis [4] describes a RDP as a piece of software which takes a sentence and turns it into a parse tree by performing a 'depth-first' search. But a search of what? One might try to call it a search of the parse tree, but that's not exactly right. The 'recursive descent' name comes from the way that a RDP traverses through a syntax diagram of a context free grammar [7]. The recursive descent algorithm in this paper does not use sytax diagrams (in hindsight, this was a questionable design choice). This section will describe how recursive descent parsing works without using syntax diagrams.

Consider a context-free grammar with the following production rules:

$$S \to aS \tag{1}$$
$$S \to bS \tag{2}$$
$$S \to \epsilon \tag{3}$$

And the following string which we will attempt to parse: "ab"

We know, right off the bat, that the start symbol S will be the root of any parse tree created under this grammar, by virtue of it being the start symbol. (Figure 2)

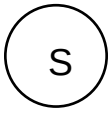

Figure 2: Parse Tree 0

The next parse tree we should consider is the parse tree that is generated when we follow the first production rule. (Figure 3) Notice that, in this instance, the parse tree does not conflict with the string we are trying to parse - i.e. regardless of the production rules we follow the string that is produced from any further production rules we follow will start with "a".

For the next parse tree, we will try to repeat are last action (following the first possible production rule). (Figure 4) This parse tree conflicts with the string we are trying to produce since any string produced by further production rule applications will produce a string starting with "aa". In this

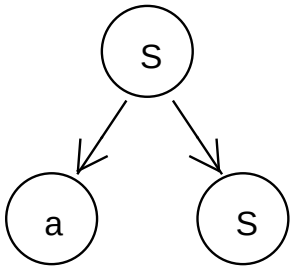

Figure 3: Parse Tree 1 - Valid

case, we go back to the previous parse tree and try a different production rule.

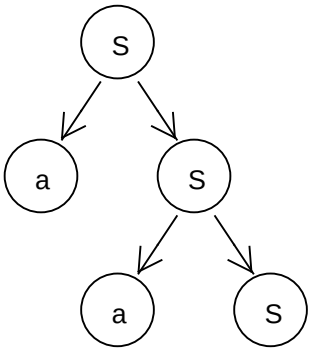

Figure 4: Parse Tree 2 - Invalid

In figure 5, we've followed the second production rule and our new parse tree fits with the input string, so we can continue our descent.

The next two parse trees (Figure 6), created by applying the first and second production rules to Parse Tree 3, are both invalid because they extend past the length of our input string.

In the last step, by applying the third production rule to Parse Tree 3, we have a parse tree which terminates and produces the desired input string, "ab". (Figure 7)

Finally, let's diagram our traversal through the possible parse trees (Fig-

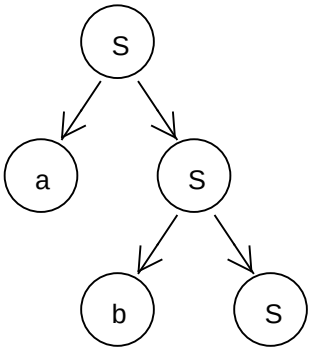

Figure 5: Parse Tree 3 - Valid

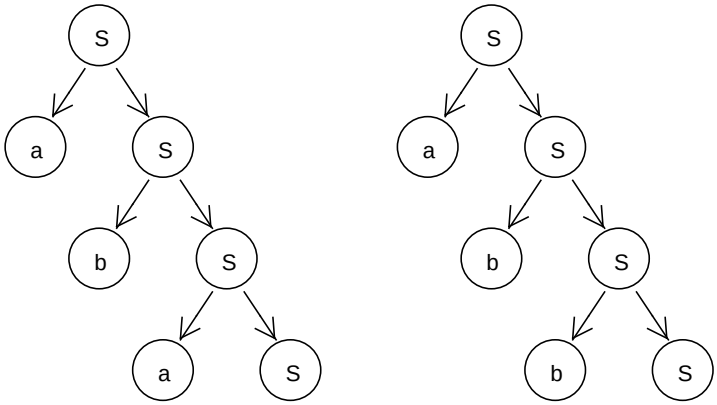

Figure 6: Parse Trees 4 and 5 - Both Invalid

ure 8). Shown this way, one can notice a pattern in our attempt to build the tree. The recursive descent parser performs a depth first search of the graph of possible parse trees, looking for a parse tree which fits the input string. The child nodes from each PT (Parse Tree) node in the graph are PT nodes generated by applying each of the production rules to the first (left, deepest) nonterminal symbol in that PT node. The depth of each node in the PT tree corresponds with the number of production rules that have been applied.

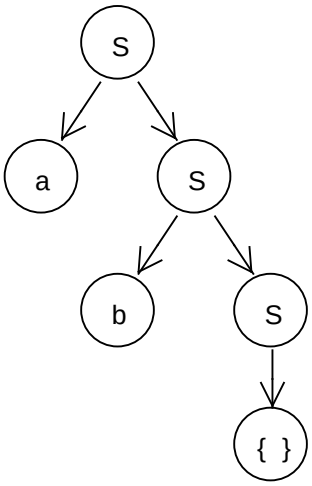

Figure 7: Parse Tree 6 - Complete

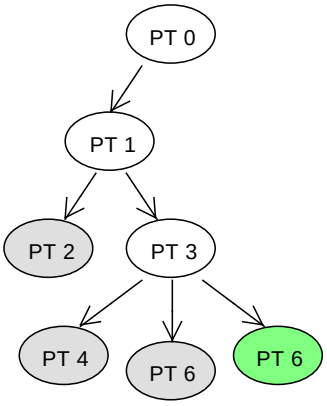

Figure 8: Parse Tree Search Progression

2 Effect of Grammar Transformations on Parse Trees

Left recursion is a problem for recursive descent parsers because it causes them to go into an infinite loop. Using the model described in section 1: when a PT node is reach where the first non-terminal symbol has a production rule with left recursion, it's child node will have the same non-terminal symbol so it will produce a child node with the same non-terinal symbol ad infinitum - and none of those children will consume any terminals from the input stream so the parser will not proceed.

So, removing left recursion from context free grammars is a necessary evil for recursive descent parsing. It just takes two transformations to turn any context free grammar with left recursion into a *weakly equivalent* grammar with only right recursion. These two transformations are direct left recursion elimination, $DLRE(G; R_\alpha, R_\beta) \rightarrow G'$ and substitution, $Sub(G; r_\alpha, R_\beta) \rightarrow G'$, which is used to remove indirect left recursion. [1, 4] These sections will describe how these transformations work on the grammar and on their parse trees.

2.1 Direct Left Recursion Elimination

A left recursive grammar, G, has rules for some non-terminal A of the form $A \rightarrow A\alpha_i$ and $A \rightarrow \beta_j$, $i \in [1, m]$, $j \in [1, n]$. Let R_α and R_β represent the sets of those rules respectively where the notation $R_\alpha(1)$ represents $A \rightarrow A\alpha_1$. Such a grammar represents a language which contains strings of the form $\beta_y\alpha_{x_1}\alpha_{x_2}...\alpha_{x_p}...\alpha_{x_{k-1}}\alpha_{x_k}$ where $y \in [1, m]$ and each $x_p \in [1, n]$. However, to produce such a string, the alpha rules need to be followed in reverse order:

$$R_\alpha(x_k), R_\alpha(x_{k-1}), ...R_\alpha(x_p), ...R_\alpha(x_2), R_\alpha(x_1), R_\beta(y)$$

The transformation [1] $DLRE(G; R_\alpha, R_\beta) \rightarrow G'$ changes each rule in R_α to the form $A' \rightarrow \alpha_i A'$, each rule in R_β to the form $A \rightarrow \beta_j A'$, and adds an additional rule $R_\epsilon = A' \rightarrow \epsilon$. Consequently, the order of the followed production rules in G' results in alpha rules being followed in forward order:

$$R_\beta(y), R_\alpha(x_1), R_\alpha(x_2), ...R_\alpha(x_p), ...R_\alpha(x_{k-1}), R_\alpha(x_k), R_\epsilon$$

The transformation effects the parse trees by flipping and reversing A chains, replacing lower A nodes with A' nodes, moving the β up, and adding an A' node and ϵ node to the end of the chain. (Figure 9) The inverse transformation removes last A' and ϵ nodes, moves the β back down to the

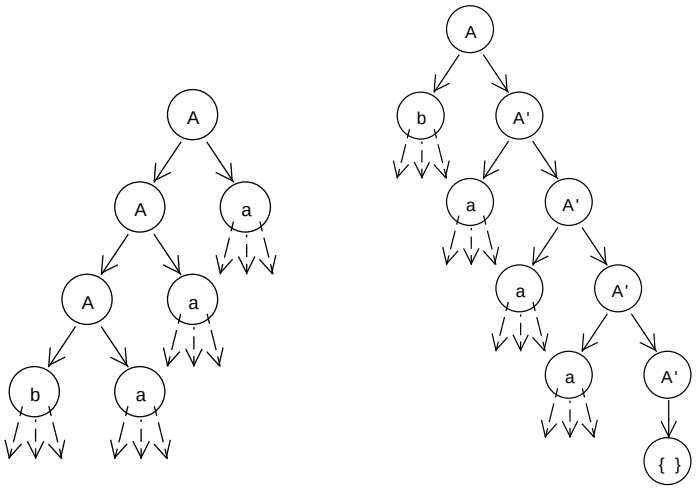

Figure 9: Original and Transformed Parse Trees

end of the chain, and changes the A' nodes back into A nodes, then flips and reverses the A chain. Note that the α and β nodes are arbitrary strings, so in reality they might be multiple nodes which might have any number of children.

More formally, for each A node in the RD-parsable grammar, the inverse transformation $DLRE^{-1}(T'; A, A') \rightarrow T$ modifies chains of A/A' nodes where the children of the A node are C_A, of all but the last A'_{x_p} node are $C_{A'_{xp}}$, and the last A'_ϵ node has only the child ϵ. From the $DLRE$ transformation rules we know that the C_A will be of the form $\beta_y A'_{x_1}$. We can also conclude from the $DLRE$ tranformation that each $C_{A'_{xp}}$ will be of the form $\alpha_p A'_{x_{p+1}}$ when $p < k$ and A'_ϵ when $p = k$.

The inverse transformation first removes the A'_ϵ node. Then it changes each remaining A'_{x_p} node into an A_{x_p} node with the same children. Next the A_{x_p} are restructured such that each $C_{A_{xp}}$ is equal to $A_{x_{p-1}}\alpha_p$ where $1 < p$ and $\beta_y \alpha_k$ where $p = 1$. The top node of the chain, A, is replaced with A_{x_k}. And this process is repeated for each chain.

2.2 Substitution

TODO

3 Compiling a Context Free Grammar for RD Parsing

It is important to notice that not all context free grammars can be directly parsed by recursive descent. Some context free grammars require a bit of maniuplation to remove left recursion (direct or otherwise) [7]. This process of converting a context free grammar into a *weakly equivalent* recursive descent parsable grammar shall be refered to as *compiling* the grammar.

In this section, a grammar will be define from an ordered collection of production rules. My parser uses context-free grammar rules, which are comprised of a 'head' (the single-symbol left hand side of the production rule), and a 'tail' (one or more symbols comprising the right hand side of the production rule).

These grammars may be 'compiled' using the four procedures: factoring, substitution, removing left recursion, and removing useless rules. Let 'decision list' define an ordered list of production rule choices which produces a parse tree. As each of these four procedures produces a weakly equivalent grammar, there exists a mapping for any decision list in a compiled grammar back into a same-terminal-producing decision list in the pre-compiled (parent) grammar. My parser keeps track of these inverse transformation rules as performs it's compilation procedure so that a compiled grammar's decision list can be easily converted to the initial grammar's equavalent decision list.

Take this simple grammar for example:

$$S \to AB \tag{1}$$
$$A \to a \tag{2}$$
$$A \to SA \tag{3}$$
$$B \to b \tag{4}$$
$$B \to SB \tag{5}$$

It compiles into the weakly equivalent grammar:

$$Z \to \epsilon \tag{1}$$
$$B \to b \tag{2}$$
$$S \to aBS' \tag{3}$$
$$S' \to \epsilon \tag{4}$$
$$A \to aZ \tag{5}$$
$$B \to aBS'B \tag{6}$$
$$S' \to aZBS' \tag{7}$$
$$Z \to bS'A \tag{8}$$
$$Z \to aBS'BS'A \tag{9}$$

So when the terminal stream "aabb" is parsed in the compiled grammar to the decision list $[3, 6, 2, 4, 2, 4]$ it can be transformed into the parent-grammar-equivalent decision list: $[1, 2, 5, 1, 2, 4, 4]$.

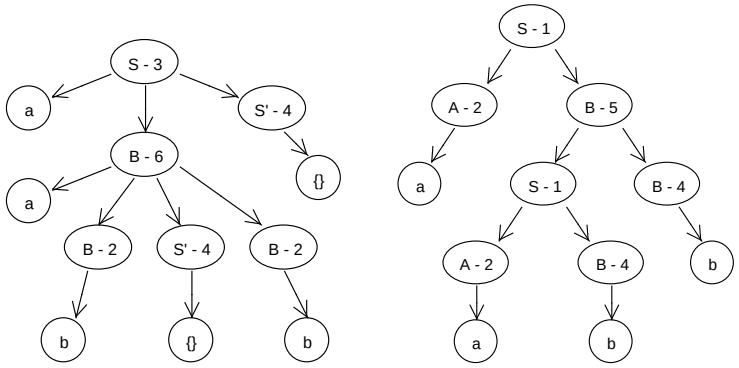

Figure 10: Compiled Grammar Parse Tree (left) Parent Grammar Parse Tree (right)

Terms

- **Grammar**: a phrase-structure grammar is defined by a finite vocabulary (alphabet), a finite set of initial strings, and a finite set of rules... [2] (see Production Rule)
- **Context-Free Grammar**: a context free grammar is one which only has production rules whose head is a single non-terminal symbol. [3, 5, 7]
- **Production, Production Rule, Rewrite Rule**: rules of the form $X \rightarrow Y$ where X and Y are strings in [a grammar] [2]; define the nonterminal symbols by sequences of terminals and nonterminal symbols [7]; rules which specify how nonterminal symbols may be expanded into new sequences of symbols (terminal or otherwise).
- **Head (Production Rule)**: the left hand side of a production rule
- **Tail (Production Rule)**: the right hand side of a production rule
- **Parse Tree**: an ordered, rooted tree whose nodes are symbols in a context-free grammar where the children of each brach node correspond to the tail of some production rule in said grammar; a tree-representation of the grammatical structure of [an input stream] [3]
- **Weakly Equivalent (Grammar)**: two grammars are [weakly] equivalent if they define the same language. [6]
- **Strongly/Structurally Equivalent (Grammar)**: two grammars are strongly or structurally equivalent if they are weakly equivalent and can assign any sentence the same parse tree. [6]

References

[1] Alphred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Pearson Education, Inc., 1986.

[2] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[3] Alice E. Fischer and Frances S. Grodzinsky. *The Anatomy of Programming Languages*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.

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

[5] Peter Linz. *An Introduction to Formal Languages and Automata.* Jones and Bartlett Publishers, Inc., Sudbury, MA, 2001.

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

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