Notes

Ernest Kirstein

October 13, 2014

1 Recursive Descent Parsing

Historically, recursive descent parsers have been coded manually or else compiled from a context free grammar specification. [6] Coding a RDP manually is tedious and error prone. Compiling a context free grammar into a recursive descent parser may not preserve the *strong equivalence* of the grammar since some grammars need to be manipulated before they can be parsed by a RDP. This makes any manipulation of the resulting parse trees much more difficult.

In this paper, I will describe a novel approach to recursive descent parseing which solves both problems by first compiling a grammar into a recursive descent parseable grammar, then decompiling the parse trees generated from the RDP back into the initial grammar.

1.1 What does a Recursive Descent Parser Do?

The introductory material by Dr. Lewis [3] describes a RDP as a piece of software which takes a sentence and puts it into a parse tree by performing a 'depth-first' search. But a search of what? Dr. Lewis called it a search of the derivation (parse) tree, but that's not exactly right. It turns out to be a supprisingly difficult question to answer. This section will explain what, exactly, a recursive descent parser recursively descends.

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

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

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 1)

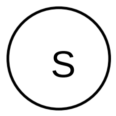


Figure 1: Parse Tree 0

The next parse tree we should consider is the parse tree that is generated when we follow the first production rule. (Figure 2) 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".

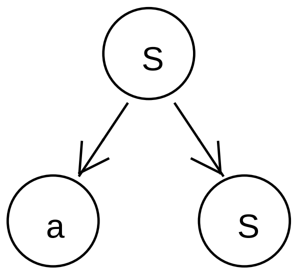


Figure 2: Parse Tree 1 - Valid

For the next parse tree, we will try to repeat are last action (following the first possible production rule). (Figure 3) 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 case, we go back to the previous parse try and try a different production rule.

In figure 4, 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 5), created by applying the first and

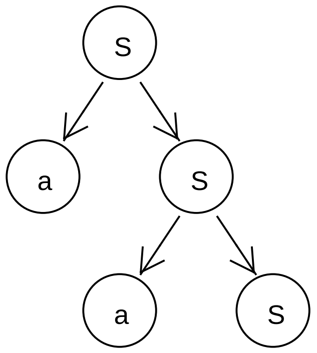


Figure 3: Parse Tree 2 - Invalid

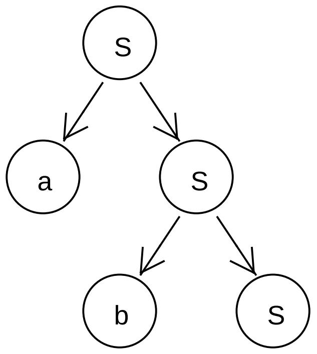
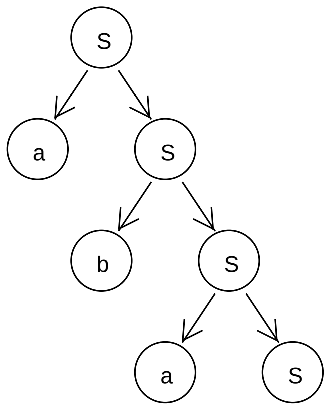


Figure 4: Parse Tree 3 - Valid

second production rules to Parse Tree 3, are both invalid because they extend past the length of our input string.

At 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 6)

Finally, let's diagram our traversal through the possible parse trees (Figure 7). Shown this way, it becomes quite clear that what a recursive descent parser does - it does a depth first search of the graph of possible parse trees, looking for a parse tree which fits the input string, where the child nodes from each PT node in the graph are PT nodes generated by applying each



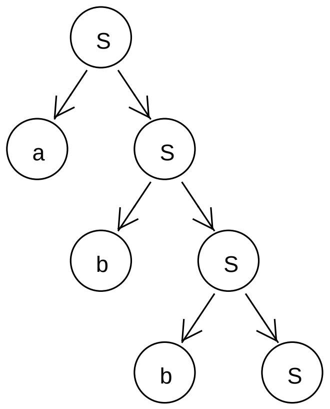


Figure 5: Parse Trees 4 and 5 - Both Invalid

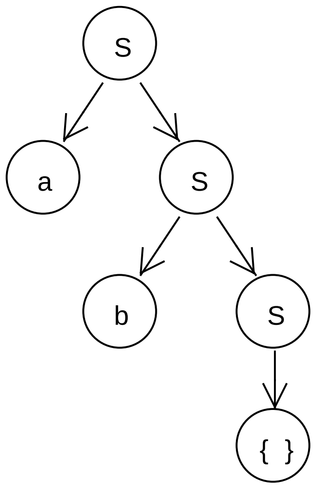


Figure 6: Parse Tree 6 - Complete

of the production rules to the first nonterminal symbol in that PT node. The depth of each node in the PT graph corresponds with the number of terminal symbols (including the empty string) that have been parsed.

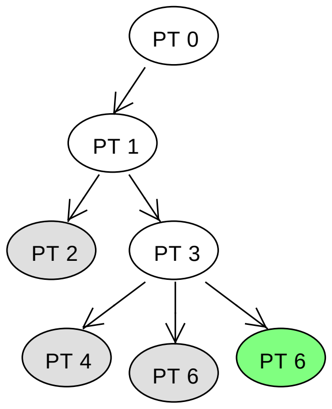


Figure 7: Parse Tree Search Progression

1.2 Compiling and Decompiling a Context Free Grammar for RDP

The core rules for my parser are built off of Dr. Lewis's notes [3]. A grammar is define as 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: $[1, 2, 5, 1, 2, 4, 4]$.

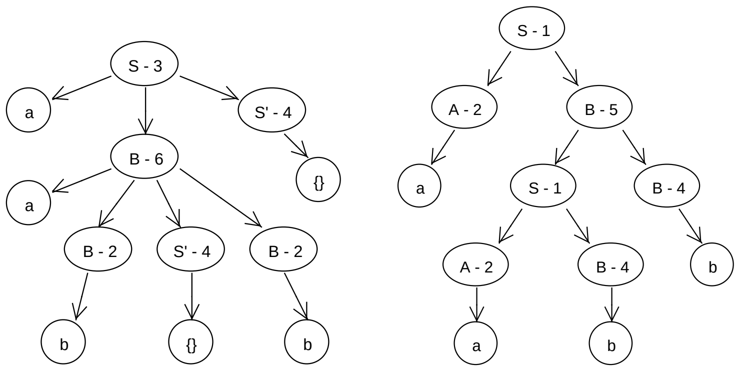


Figure 8: Compiled Grammar Parse Tree (left) Parent Grammar Parse Tree (right)

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... [1] (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. [2, 4, 6]
- **Production, Production Rule, Rewrite Rule**: rules of the form $X \rightarrow Y$ where X and Y are strings in [a grammar] [1]; define the nonterminal symbols by sequences of terminals and nonterminal symbols [6]; 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] [2]
- **Weakly Equivalent (Grammar)**: two grammars are [weakly] equivalent if they define the same language. [5]
- **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. [5]
- **Recursive Descent Parser**: (See Section 1) a parser which performs a depth first search of all potential parse trees of a [3, 6]

References

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

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

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

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

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

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