

Dotted Grammars - Generalized Deterministic Parsing

Table of contents

1 Dotted Grammars Home Page.....	2
1.1 Background.....	2
1.2 Phrase structure grammars.....	2
1.3 Unrestricted phrase structure grammars.....	2
1.4 Context sensitive phrase structure grammars.....	2
1.5 Context free phrase structure grammars.....	3
1.6 regular grammars (also called right linear grammars).....	3
1.7 Some referencesbooks.....	3
1.8 Dotted grammars.....	3
1.9 Two example dotted grammars.....	4
1.10 The Tiny parser generator.....	5

1. Dotted Grammars Home Page

You are in the right place for [information](#) on *dotted grammars* and [my generalized deterministic parser generator](#) for these grammars.

1.1. Background

My research in this field started in about 1988. During that period I developed an interest in the machine models which were used to implement software components like parsers, Prolog interpreters, the machine models for C programs and the like.

By studying relevant literature and working example programs I decided that the state of the systems studied were all based on a single runtime stack which was used to store attributes and to guide the flow of control within the system. That's not to say that none of these systems had more than one stack. It's only to say that additional stacks (if present) were not essential.

Somewhere around 1991 I decided I would study the properties of systems which *needed* two runtime stacks to store their internal state. I'd often ask myself: "What new features could I add to a language like C or C++ if there were a second runtime stack available?"

After a few months I came up with what I called "record" and "playback" statements, which would effectively allow a system to generate the code it would execute in the future, based on code it had already executed (within the context of a single program). Recording would of course push information to be executed in the future on the second stack and the playback statement would arrange for the execution of the generated code.

To me it's comforting to have a formal model as the motivation for the way I write software. So I tried to model "recording" and "playing back" with Chomsky's phrase structure grammars. This line of thought lead to the definition of what I call dotted grammars.

1.2. Phrase structure grammars

As I understand it, *phrase structure grammars* were originally introduced by Noam Chomsky as a mathematical tool with which to define the syntax and semantics of natural languages. These grammars are generally divided into the following four classes.

1.3. Unrestricted phrase structure grammars

This is the most general class of grammars. These grammars can be shown to be equal in expressive power with Turing machines and as such are said to generate the set of recursively enumerable languages.

1.4. Context sensitive phrase structure grammars

The expressive power of context sensitive grammars has been shown to equal that of Linear Bounded Automata (nondeterministic Turing machines with a linear bound on the length of their tape). These grammars are said to generate the set of context sensitive languages.

1.5. Context free phrase structure grammars

Context free grammars generate the set of context free languages. Pushdown stack automata are the machines which have been shown to accept the set of context free languages. As it turned out, context free grammars extended with arguments (called attribute grammars, affix grammars) are capable of generating non context free languages. These grammars are used extensively in computer science. Much research has been done in the area of context free grammars and many results are known.

1.6. regular grammars (also called right linear grammars)

Regular grammars are known to many as "regular expressions". These grammars have also found many applications in computer science. The machine model which accept regular grammars are called finite state machines.

1.7. Some referencesbooks

Here are some books (in no particular order) I enjoyed on (the application of) phrase structure grammars and their associated machines.

1. T.A. Sudkamp [1991] "Languages and Machines" Addison-Wesley.
2. A. Aho, R.Sethi, J. Ullman [1986] "Compilers, principles techniques and tools" Addison-Wesley.
3. D.Grune, C. Jacobs [1990] "Parsing Techiques" Ellis Horwood Limited.
4. R. Uzgalis, J. Cleaveland [1977] "Grammars for Programming Languages" Elsevier Computer Science Library.
5. [John Shutt](#) has nice a introduction to Chomsky's grammars on his [Survey of Grammar Models](#) page.

1.8. Dotted grammars

In general it has not proven to be an easy task to define languages using unrestricted phrase structure grammars. Applications of these grammars generally restrict these grammars in some form as to make it possible to efficiently and effectively apply them. Unfortunately such restrictions often also lead to restrictions in the languages accepted by these restricted systems. As far as I am concerned all efficient parsers have the following properties:

1. Given a sentential form of a grammar, they know at which position in this sentential form rewriting should take place.
2. Given the position at which rewriting should take place, they also know how to

efficiently select one and only one production rule with which to rewrite this sentential form.

Dotted grammars trivially modify (and restrict) unrestricted phrase structure grammars in such a way that both properties mentioned above are met, while preserving the ability to generate non context free languages. We simply place a unique nonterminal symbol (a dot) to both the left hand side and the right hand side of all production rules. The dot then identifies the position at which rewriting should take place.

This makes it easy to show that parsers for dotted grammars can efficiently parse all LL(1) en LR(1) grammars. It is also easy to show that linear time parsing of some context sensitive grammars is possible.

Please check out my current paper on dotted grammars available in [pdf](#) format. This paper contains the following:

1. Definition for phrase structure grammars
2. Definition for dotted grammars
3. Parsers for dotted grammars
4. Introduction of recording grammars together with an example grammar and parse table. These grammars allow the concept of "recording" and "playing back" to be maintained in a consistent manner.
5. Generating table driven parsers for dotted grammars
6. Showing that the LL(1) and LR(1) languages are subsets of the languages generated by dotted grammars.
7. A short description of a parser generator for dotted grammars. This program generates C++ code and requires a C++ compiler.
8. A section on generating the Chomsky language classes with deterministic dotted grammars. Please read carefully here, since this section is a bit more theoretical in nature I'm likely to have made at least a few mistakes.

I've also thrown together a first take at a [quick introduction](#) to Generalized Deterministic Parsing based on dotted grammars.

1.9. Two example dotted grammars

Two example dotted grammars which are accepted by the parser generator are given here.

1.9.1. Showing that it is context sensitive

[This example](#) considers a way to define a grammar for the language containing strings of an equal number of a's , b's and c's in that order. This language is often used as an example of a language generated by a context sensitive grammar.

1.9.2. A simple expression grammar

A [simple expression grammar](#) extended with semantic actions.

1.10. The Tiny parser generator

The [Tiny parser generator](#) is a proof of concept implementation of a parser generator for dotted grammars. It is a generalized deterministic parser. The program is written in C++ using the GNU C++ compiler under Linux. It is distributed under the GNU public license.