Grammlator Manual

Dr. Rudolf Gardill
Bamberg, Germany
2020

Abstract

The concept of using grammar rules as statements, which control the execution flow in a program, has been implemented and published as a theses in 1979 [1] .

Grammlator is an advanced implementation of this concept. It takes a C# code file as input and evaluates special comment lines containing grammar rules. C# enum declarations specify the terminal symbols and C# method declarations are used as semantic methods. The formal parameters of these C# methods implement the strongly typed attributes of the terminal and nonterminal symbols. Conflicts produced by ambiguous grammars can be solved by static priorities or by C# functions at runtime of the generated code.

Grammlator generates well commented C# statements and method calls implementing the respective control flow. Special optimizations reduce the length of the generated code, improve readability and performance. The generated code is inserted in a specific region of the input file. Details of code generation are set by special directives in the source file.

This manual describes the user interface of grammlator, the syntax and semantics of the input files and the C# code generated by grammlator. Examples help to understand the concept and the manifold ways to use grammlator.

Keywords: program generator [2] , preprocessor [3] , syntax directed translation [4] , Extended Backus-Naur Form [5] , bottom-up parsing [6] , LALR parser [7] , C# [8] , MS Visual Studio [9]

Table of Contents

1. Introduction

Welcome to the grammar translator "grammlator". Grammlator is a software tool and assists programmers who want to write applications that analyze input streams. So grammlator is a special "program generator", "a computer program that can be used to help to create other computer programs"[2].

Grammlator may also be of interest to students who want to try out examples of LR-parsing. Typing in a grammar, clicking translate, perhaps correcting some typos, will list the symbols, the conflicts, and the states and parser actions (before and after optimization). And even the optimized code generated by grammlator gives an impression, how LR-parsing can work.

But the main purpose of grammlator is to assist programmers. The programmer describes the structure of the input data using a grammar and applies the principles of "Syntax directed translation"[4] to write a program, that analyzes a stream of data elements. This is done by adding attributes and C#-methods to the grammar rules, embedding this combination as a region into a C# program file and let grammlator add (or replace) the lines of another region of the same file with generated code. The objective is to get a well structured and documented program, which can be maintained using Visual Studio (or other C# development tools) and grammlator.

This version of grammlator produces an LALR(1) parser. Reading this manual you may get an idea of the concept of grammars and of states and actions of an analyzer. The basics and the limitations of LALR(1) parsing are well known and well documented [7] and you will find precise information in manifold literature and in the internet.

Grammlator has some unique features:

- Most of the examples in this manual analyze streams of characters. But grammlator does not contain a concept of "character" and even less a concept of "string". It handles abstract "terminal symbols" and sequences of terminal symbols. The C# implementation of each terminal symbol is a C# constant defined by the programmer. Typically they are elements of a C# enum. The examples show some ways how a programmer can map characters to such constants by simple C# instructions or methods.
- Grammlator does not build syntax trees. It "only" generates a sequence of instructions which test terminal symbols and call the C# methods written by the programmer. The programmer may use this methods to generate syntax trees.
- The syntax of the grammar rules is alike "Extended Backus-Naur Form"[5]. Comments in the grammar part are like C# comments, attributes look like C# formal parameters.
- The semantic attributes, which can be assigned to terminal and nonterminal symbols are strongly typed by using names of C# value or C# reference types. Even user defined types (classes, structs) are allowed. Grammlator does know nothing of these types. It simply checks the equality of the names.
- Grammlator produces C# code. To produce code is common practice in implementing top-down-parsers, but unusual and not trivial in implementing bottom-up-algorithms. (Most implementations of bottom-up-methods use interpreters and tables).
- Grammlator produces optimized code which in simple cases even does not use a state stack.
- As an option grammlator inserts comments (typically the kernel items of the states and the applied grammar rules) into the generated code.
- The programmer may give a static priority (a number) or a dynamic priority (an int method) to each grammar rule. Grammlator uses this priorities to solve shift-reduce and reduce-reduce-conflicts in ambiguous grammars.

- Grammlator has no special options to produce a lexer or a parser. But it offers support to use syntax directed programing when writing a class, which implements special methods (Peek(), Accept() ...) and which can be used by another method to get its terminal symbols.

Next Chapters:

2. C02 The Grammlator Application.md

3. The structure of the grammlator input (C# source file)

```
#region grammar
//| *=;
#endregion grammar
#region grammlator generated
#endregion grammlator generated
```

4. The startsymbol

5. Define terminal symbols by a C# enum

6. Define nonterminal symbols by sequences of terminal symbols

7. Recursion

8. Syntactic sugar: repeat operators

9. Solve Conflicts

10.

---

1. Gardill, Rudolf, 1979, SKOP : Syntaxnotation als Kontrollstruktur für Programmiersprachen. ↩
2. https://www.thefreedictionary.com/program+generator ↩ ↩
3. https://en.wikipedia.org/wiki/Preprocessor ↩
4. https://en.wikipedia.org/wiki/Syntax-directed_translation ↩ ↩
5. https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form ↩ ↩
6. https://en.wikipedia.org/wiki/Bottom-up_parsing ↩
7. https://en.wikipedia.org/wiki/LALR_parser ↩ ↩
8. https://en.wikipedia.org/wiki/C_Sharp_(programming_language) ↩
9. https://en.wikipedia.org/wiki/Microsoft_Visual_Studio ↩