



Engineering Thesis

**FORMAL GRAMMAR
PRODUCTION RULE PARSING TOOL**

Karol Belina

keywords:

Parser combinators, context-free grammars,
Extended Backus-Naur Form

short summary:

The thesis documents the process of designing and implementing a tool for parsing the production rules of context-free grammars in a textual form. It discusses the choice of Extended Backus-Naur Form notation over the alternatives and provides a mathematical model for parsing such a notation. The implemented parser can turn a high-level specification of a grammar into a parser itself, which in turn is capable of constructing a parse tree from arbitrary input provided to the program with the use of parser combinators.

Supervisor	dr inż. Zdzisław Sławski
	Title/degree/name and surname	grade	signature

The final evaluation of the thesis

Head of the examination commission
	Title/degree/name and surname	grade	signature

*For the purposes of archival thesis qualified to:**

a) category A (perpetual files)

b) category BE 50 (subject to expertise after 50 years)

** delete as appropriate*

stamp of the faculty

Wrocław 2020

Abstract

The thesis presents the design and implementation of a context-free grammar parsing tool with real-time explanations and error detection. It discusses the choice of Extended Backus-Naur Form notation over the alternatives and provides a mathematical model for parsing such a notation. For this purpose, the official specification of the EBNF from the ISO/IEC 14977 standard has been examined and transformed into an unambiguous form. A definition of a grammar is proposed to act as a result of the syntactic analysis phase formed with a technique called *parser combination*. A method of testing an arbitrary input against the language generated by the constructed grammar is described. The thesis shows the process of creating a simple command line REPL program to act as a basic tool for interfacing with the grammar parser and checker, but in order to efficiently use the library, a web-based application is designed on top of that to serve as a more visual, user-friendly and easily accessible tool. It describes the deployment of the application on a static site hosting service, as well as a cross-platform desktop application. The designed and implemented system gives the opportunity to extend it with other grammar specifications.

Streszczenie

Praca przedstawia proces projektowania i implementacji narzędzia służącego do analizy syntaktycznej gramatyk bezkontekstowych z naciskiem na obsługę błędów i wyjaśnień w czasie rzeczywistym. Omawia wybór rozszerzonej notacji Backusa-Naura i przedstawia matematyczny model do analizy takiej notacji. W tym celu przeprowadzono analizę i przekształcenie w jednoznaczną formę oficjalnej jej specyfikacji zdefiniowanej w standardzie ISO/IEC 14977. Zaproponowana zostaje definicja gramatyki tej notacji, która jest tworzona w wyniku analizy syntaktycznej za pomocą techniki zwanej *kombinacją parserów*. Opisana zostaje metoda sprawdzania dowolnego ciągu znaków pod kątem języka generowanego przez analizowaną gramatykę. Praca przedstawia stworzenie prostego programu działającego z poziomu wiersza poleceń, który jest podstawowym narzędziem do analizy gramatyk, jednak by móc efektywnie korzystać ze stworzonej biblioteki, zaprojektowana zostaje aplikacja webowa, która służy za bardziej wizualne, przyjazne i łatwo dostępne dla użytkownika narzędzie. Praca opisuje wdrażanie aplikacji na usługę hostingową dla statycznych stron, a także jako wieloplatformowej aplikacji. Zaprojektowany i wdrożony system daje możliwość rozszerzenia go o inne specyfikacje gramatyk.

Contents

1	Problem analysis	1
1.1	Description and motivation	1
1.2	Goal	1
1.3	Scope	2
1.4	Glossary	2
2	Theoretical preliminaries	3
2.1	Formal grammars	3
2.1.1	Introduction to formal grammars	3
2.1.2	The Chomsky Hierarchy	4
2.1.3	Parsing Expression Grammars	4
2.2	Why EBNF?	4
2.3	Specification	4
2.4	Syntactic analysis	4
2.4.1	Methods	4
2.4.2	Parser combination	5
3	Analysis of similar solutions	7
4	Design of the project	9
4.1	Requirements	9
4.1.1	Functional requirements	9
4.1.2	Non-functional requirements	9
4.2	Use cases	9
4.3	The architecture	9
4.3.1	Used technologies	9
4.3.2	Class diagram	9
4.4	Interface prototype	9
5	Implementation of the project	11
5.1	Software environment	11
5.2	Business logic	11
5.2.1	Grammar definition	11
5.2.2	Lexical analyser	11
5.2.3	Syntactic analyser	11
5.2.4	Left recursion handling	11
5.2.5	Dependency graph reduction	11
5.2.6	Grammar processing	12
5.3	Command line application	12
5.4	Web-based application	12

5.4.1	Linking the business logic	12
5.4.2	Text editor	12
5.4.3	Visualizations	12
6	Testing	13
6.1	Business logic testing	13
6.1.1	Unit testing	13
6.1.2	Benchmarking	13
6.2	UI testing	13
7	Deployment	15
7.1	GitHub Pages	15
7.2	Electron	15
	Summary	17
	Bibliography	19
	List of Figures	21
	List of Tables	23
	List of Listings	25
A	Modified specification	27

1. Problem analysis

1.1. Description and motivation

Programming language theory has become a well-recognized branch of computer science that deals with the study of programming languages and their characteristics. It is an active research field, with findings published in various journals, as well as general publications in computer science and engineering. But besides the formal nature of Programming language theory, many amateur programming language creators try their hand at the challenge of creating a programming language of their own as a personal project. It is certainly relevant for a person to write their own language for educational purposes, and to learn about programming language and compiler design. However, the language creator must first of all make some fundamental decisions about the paradigms to be used, as well as the syntax of the language.

The tools for aiding the design and implementation of the syntax of a language are generally called *compiler-compilers*. These programs create parsers, interpreters or compilers from some formal description of a programming language (usually a grammar). The most commonly used types of compiler-compilers are *parser generators*, which handle only the syntactic analysis of the language — they do not handle the semantic analysis, not the code generation aspect. The parser generators most generally transform a grammar of the syntax of a given programming language into a source code of a parser for that language. The language of the source code for such a parser is dependent on the parser generator.

Most such tools, however, [TODO offer] too much complexity and generally have a steep learning curve for people inexperienced with the topic. Limited availability makes them less fitted for prototyping a syntax of a language — they often require a complex setup for simple tasks, which is not welcoming for new users [TODO and may lead to...?]. The lack of visualization capabilities shipped with these tools makes them less desirable for teachers in the theory of formal languages, who often require such features for educative purposes in order to present the formulations of context-free grammars in a more visual format.

1.2. Goal

The main goal of this thesis is to design and implement a specialized tool in the form of an easily accessible web-based application, that serves teachers, programmers and other kinds of enthusiasts of the theory of formal languages in the field of discrete mathematics and computer science, in order to formulate and visualize context-free grammars in the form of the Extended Backus-Naur Form. The thesis itself will document the entire process of creating such a project.

[TODO jak projekt pomoże w powyższych problemach?]

In order to achieve the general goal, several sub-goals have been distinguished, all of which contribute to the main objective as a whole

- analysis of existing solutions and applications,

- presentation of the theoretical preliminaries of the project,
- definition of the outline of the project, including a description of the functional and non-functional requirements, the use case diagram, use case scenarios, the class diagram, and the user interface prototype,
- description of technologies used in the implementation,
- implementation of the project,
- description of the testing and deployment environments.

1.3. Scope

1.4. Glossary

*[**TODO** słownik wybranych pojęć]*

AST	Abstract syntax tree — [TODO] ,
EBNF	Extended Backus-Naur Form — [TODO] ,
parser	[TODO] ,
REPL	Read-Eval-Print loop — [TODO] .

2. Theoretical preliminaries

2.1. Formal grammars

2.1.1. Introduction to formal grammars

Formal grammar of a language defines the construction of strings of symbols from the language's *alphabet* according to the language's *syntax*. It is a set of so-called *production rules* for rewriting certain strings of symbols with other strings of symbols — it can therefore generate any string belonging to that language by repeatedly applying these rules to a given starting symbol [14]. Furthermore, a grammar can also be applied in reverse: it can be determined if a string of symbols belongs to a given language by breaking it down into its constituents and analyzing them in the process known as *parsing*.

For now, let's consider a simple example of a formal grammar. It consists of two sets of symbols: (1) set $N = \{S, B\}$, whose symbols are *non-terminal* and must be rewritten into other, possibly non-terminal, symbols, and (2) set $\Sigma = \{a, b, c\}$, whose symbols are *terminal* and cannot be rewritten further. Let S be the start symbol and set P be the set of the following production rules:

1. $S \rightarrow aBSc$
2. $S \rightarrow abc$
3. $Ba \rightarrow aB$
4. $Bb \rightarrow bb$

To generate a string in this language, one must apply these rules (starting with the start symbol) until a string consisting only of terminal symbols is produced. A production rule is applied to a string by replacing an occurrence of the production rule's left-hand side in the string by that production rule's right-hand side. The simplest example of generating such a string would be

$$S \Rightarrow_2 \underline{abc}$$

where $P \Rightarrow_i Q$ means that string P generates the string Q according to the production rule i , and the generated part of the string is underlined.

By choosing a different sequence of production rules we can generate a different string in that language

$$\begin{aligned} S &\Rightarrow_1 \underline{aBSc} \\ &\Rightarrow_2 aB\underline{abcc} \\ &\Rightarrow_3 aa\underline{Bbcc} \\ &\Rightarrow_4 aabb\underline{cc} \end{aligned}$$

After examining further examples of strings generated by these production rules we may come into a conclusion that this grammar generates the language $\{ a^n b^n c^n \mid n \geq 1 \}$, where x^n is a string of n consecutive x 's. It means that the language is the set of strings consisting of one or more a 's, followed by the exact same number of b 's, then followed by the exact same number of c 's.

Such a system provides us with a notation for describing a given language formally. Such a language is a usually infinite set of finite-length sequences of terminal symbols from that language.

2.1.2. The Chomsky Hierarchy

In [3] Chomsky divides formal grammars into four classes and classifies them in the now called *Chomsky Hierarchy*. Each class is a subset of another, distinguished by the complexity.

Type-3 grammars generate the so-called *regular languages*. As described in [1], regular languages can be matched by *regular expressions* and decided by a *finite state automaton*. They are the most restricting kinds of grammars, with its production rules consisting of a single non-terminal on the left-hand side and a single terminal on the right-hand side. Because of their simplicity, regular languages are used for lexical analysis of programming languages [11].

Type-2 grammars produce *context-free languages* and can be represented as a *push-down automaton* which is an automaton that can maintain its state with the use of a stack.
[TODO jak w stosie wygląda pamięć]

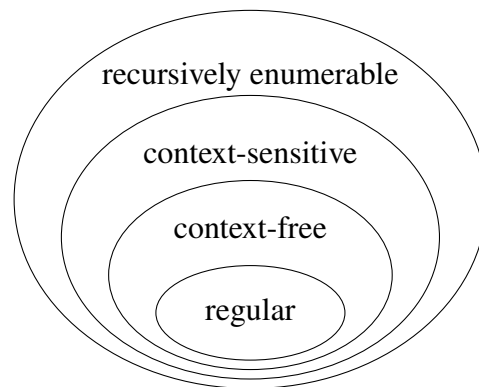


Figure 2.1: The Chomsky Hierarchy visualized

2.1.3. Parsing Expression Grammars

[TODO https://en.wikipedia.org/wiki/Parsing_expression_grammar]
[TODO [7]]

2.2. Why EBNF?

2.3. Specification

[TODO analiza i zmodyfikowanie oficjalnej specyfikacji EBNF]
See appendix A.

2.4. Syntactic analysis

2.4.1. Methods

[TODO o różnych metodach i podejściach do parsowania]

2.4.2. Parser combination

[TODO] *opisanie parser combinatorów w Haskellu [17] [13] [6]*

3. Analysis of similar solutions

Coco/R

[TODO [15]]

ANTLR

[TODO [16]]

Bison

[TODO [8]]

PLY

[TODO [2]]

Regex101

[TODO [5]]

4. Design of the project

4.1. Requirements

4.1.1. Functional requirements

4.1.2. Non-functional requirements

4.2. Use cases

[**TODO** *diagram UML*] [**TODO** *scenariusze przypadków użycia*]

4.3. The architecture

4.3.1. Used technologies

[**TODO** *Git*] [**TODO** *Rust [12]*] [**TODO** *nom [4]*] [**TODO** *Svelte [9]*] [**TODO** *Rollup*]
[**TODO** *WebAssembly*]

4.3.2. Class diagram

[**TODO** *Diagram “klas”*]

4.4. Interface prototype

[**TODO** *obrazki*]

5. Implementation of the project

5.1. Software environment

Visual Studio Code

[TODO konfiguracja, rozszerzenia]

Git and GitHub

[TODO w jaki sposób używam Gita, GitHuba, jak używam branchy, issues, PR, projektów]

Cargo

[TODO konfiguracja Cargo, Clippy]

npm

Rollup

5.2. Business logic

5.2.1. Grammar definition

[TODO opis]

5.2.2. Lexical analyser

[TODO krótko o “algorytmie” tokenizacji]

5.2.3. Syntactic analyser

[TODO zdefiniowanie ważnych parserów dla EBNF]

5.2.4. Left recursion handling

[TODO przedstawienie algorytmu do usuwania lewej rekurencji i wyjaśnienie po co]

5.2.5. Dependency graph reduction

[TODO przedstawienie algorytmu do wyszukania reguły początkowej]

5.2.6. Grammar processing

*[**TODO** opisanie sposobu na sprawdzenie czy wejście należy do języka generowanego przez gramatykę]*

5.3. Command line application

5.4. Web-based application

5.4.1. Linking the business logic

*[**TODO** jak się kompiluje Rusta do WebAssembly, czyli wasm-pack]*

5.4.2. Text editor

*[**TODO** CodeMirror]*

5.4.3. Visualizations

6. Testing

6.1. Business logic testing

6.1.1. Unit testing

[TODO *Cargo test*]

6.1.2. Benchmarking

6.2. UI testing

[TODO *Jest*]

7. Deployment

7.1. GitHub Pages

7.2. Electron

Summary

Bibliography

- [1] AHO, A. V. Algorithms for finding patterns in strings. *Algorithms and Complexity* (1990), 255–300.
- [2] BEAZLEY, D. PLY homepage. <https://www.dabeaz.com/ply/>. Accessed: 2020-10-24.
- [3] CHOMSKY, N. Three models for the description of language. *IEEE Transactions on Information Theory* 2, 3 (1956), 113–124.
- [4] COUPRIE, G. Nom, a byte oriented, streaming, zero copy, parser combinators library in Rust. *2015 IEEE Security and Privacy Workshops* (2015).
- [5] DIB, F. Regex101 homepage. <https://regex101.com/>. Accessed: 2020-10-24.
- [6] FOKKER, J. Functional parsers. *Advanced Functional Programming Lecture Notes in Computer Science* (1995), 1–23.
- [7] FORD, B. Parsing expression grammars. *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL 04* (2004).
- [8] FREE SOFTWARE FOUNDATION. GNU Bison homepage. <https://www.gnu.org/software/bison/>. Accessed: 2020-10-24.
- [9] HARRIS, R. Svelte API documentation. <https://svelte.dev/docs>. Accessed: 2020-10-24.
- [10] ISO/IEC. *ISO/IEC 14977:1996(E) — Information technology, syntactic metalanguage, Extended BNF*. Geneva, 1996.
- [11] JOHNSON, W. L., PORTER, J. H., ACKLEY, S. I., AND ROSS, D. T. Automatic generation of efficient lexical processors using finite state techniques. *Communications of the ACM* 11, 12 (1968), 805–813.
- [12] KLABNIK, S., AND NICHOLS, C. *The Rust Programming Language*. No Starch Press, USA, 2018.
- [13] LEIJEN, D., AND MEIJER, E. Parsec: Direct style monadic parser combinators for the real world.
- [14] MEDUNA, A. *Formal Languages and Computation: Models and Their Applications*. CRC Press, 2014.
- [15] MÖSSENBÖCK, H., AND LÖBERBAUER, M. The Compiler Generator Coco/R homepage. <http://www.ssw.uni-linz.ac.at/Coco/>. Accessed: 2020-10-24.

- [16] PARR, T. ANTLR homepage. <https://www.antlr.org/>. Accessed: 2020-10-24.
- [17] SWIERSTRA, S. D. Combinator parsing: A short tutorial. *Language Engineering and Rigorous Software Development Lecture Notes in Computer Science* (2009), 252–300.

List of Figures

2.1	The Chomsky Hierarchy visualized	4
-----	--	---

List of Tables

List of Listings

A.1	Modified version of the EBNF language specification defined in [10]	27
-----	---	----

A. Modified specification

```
1 character
2   = ? any Unicode non-control character ?;
3 letter
4   = ? any Unicode alphabetic character ?;
5 digit
6   = ? any Unicode numeric character ?;
7 whitespace
8   = ? any Unicode whitespace character ?;
9 comment
10  = '(*', {comment | character}, '*)';
11 gap
12  = (whitespace | comment), {whitespace}, {{comment}, {whitespace}};
13 identifier
14  = letter, {{whitespace}, letter | digit};
15 factor
16  = [[gap], digit, {{whitespace}, digit}, [gap], '*'],
17    [gap], [(identifier
18      | ('[' | '(/', alternative, (']' | '/)')
19      | ('{' | '(:', alternative, ('}' | ':)')
20      | '(', alternative, ')')
21      | '"', character - '"', {character - '"'}, '"'
22      | "'", character - "'", {character - "'"}, "'"
23      | '?', {{whitespace}, character - '?', '?'), [gap]]];
24 term
25   = factor,
26     ['-', ? a factor that could be replaced
27       by a factor containing no identifiers ?];
28 sequence
29   = term, {' ', term};
30 alternative
31   = sequence, {'|', sequence};
32 production
33   = [gap], identifier, [gap], '=', alternative, (';' | '.'), [gap];
34 grammar
35   = production, {production};
```

Listing A.1: Modified version of the EBNF language specification defined in [10]