Wrocław University
of Science and Technology

Faculty of Computer Science and Management
Field of study: Computer Science
Specialty: —

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 Spławski	....	....
	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 przestawia 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. 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, nor 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 of the thesis

The main goal of this thesis is to design and implement a specialized tool, 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. In order to [**TODO**], the tool must provide a graphical user interface. Additionally, to ensure the hightest degree of accessibility, the tool must be available in the form of an easily accessible web-based application that is accessed through a web page and can run in a browser without the need of installation on the user's device. 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 of the project

The thesis will propose a definition of a grammar in the form of an abstract syntax tree of the Extended Backus-Naur Form. It will describe the process of implementing the business logic of the application in the Rust programming language compiled to WebAssembly. The compiled code is then ran inside the web-based application made with the Svelte framework, which incorporates the markup, CSS styles, and JavaScript scripts in the superset of the HyperText Markup Language (HTML).

The implementation phase will include the process of tokenization — the act of dividing the grammar in a textual form into a sequence of tokens — while taking into account proper interpretation of Unicode graphemes. The whitespace-agnostic tokens will be then combined together to form a previously-defined abstract syntax tree with a technique called *parser combination*. Several smaller helper parsers will be defined, all of which then will be combined into more sophisticated parsers capable of parsing entire terms, productions, and grammars. The implementation phase will also include the definition of an algorithm for handling left recursion in the resulting grammar, as well as a dependency graph reduction algorithm for determining the starting rule of a grammar. Up to this stage, any errors encountered in the textual form of a grammar are going to be reported to the user in a friendly format with exact locations of the errors in the input.

[TODO
- *service workers*
- *wizualizacje, edytor tekstowy i kolorowanie składni*
- *wyjaśnienia zwracane przez checker?*
- *wyrażenia regularne w specjalnych sekwencjach?*

]

The web application will be deployed on the GitHub Pages hosting service for static sites, as well as a standalone desktop application with the use of the Electron framework.

1.4. Glossary

AST Abstract syntax tree — **[TODO]**,

EBNF Extended Backus-Naur Form — **[TODO]**,

parser **[TODO]**,

REPL Read-Eval-Print loop — **[TODO]**.

[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

$$S \underset{2}{\Rightarrow} \underline{abc}$$

where $P \underset{i}{\Rightarrow} Q$ 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

$$\begin{aligned} S &\underset{1}{\Rightarrow} \underline{aBSc} \\ &\underset{2}{\Rightarrow} aB\underline{abc}c \\ &\underset{3}{\Rightarrow} a\underline{aB}bcc \\ &\underset{4}{\Rightarrow} aa\underline{bb}cc \end{aligned}$$

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, possibly followed by a single non-terminal on the right-hand side. Because of their simplicity, regular languages are used for lexical analysis of programming languages [11].

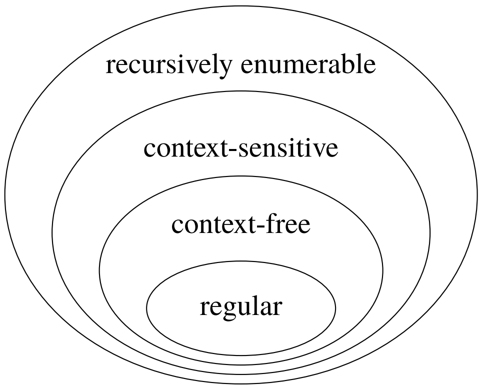

Figure 2.1: The Chomsky Hierarchy visualized

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*]

2.1.3. Parsing Expression Grammars

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

2.2. Why EBNF?

[**TODO**]

2.3. Modifying the specification

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

2.4. Methods of syntactic analysis

2.4.1. Bottom-up parsing

[**TODO**]

2.4.2. Top-down parsing and parser combination

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

```haskell
type Parser a = String -> Maybe (a, String)
```

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

[**TODO** *tabela z potrzebami, cechami i priorytetami*]

4.1.2. Non-functional requirements

[**TODO** *tabela z innymi wymaganiami i ich priorytetami*]

4.2. User stories

[**TODO**]

4.3. Use case specification

4.3.1. Use cases

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

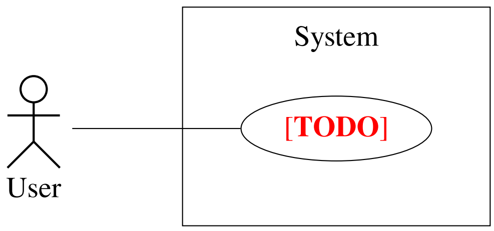

Figure 4.1: The use case diagram

[**TODO** *opisy przypadków użycia*]
[**TODO** *requirements traceability graph — jak łączą się wymagania z historyjkami z przypadkami użycia*]

4.3.2. Use case scenarios

[**TODO** *opisy scenariuszy przypadków użycia*]

4.3.3. Activity diagrams

[**TODO** *diagramy aktywności dla scenariuszów*]

4.3.4. Sequence diagrams

[**TODO**]

4.4. System architecture

4.4.1. Logical architecture

[**TODO** *diagram pakietów, czyli wydzielenie modułów w projekcie — widok i model, gdzie widok podzielony jest na komponenty a model na moduły*]

4.4.2. Physical architecture

[**TODO** *użyte technologie i zwizualizowany stack*]
[**TODO** *Git*] [**TODO** *Rust [12]*] [**TODO** *nom [4]*] [**TODO** *Svelte [9]*] [**TODO** *Rollup*] [**TODO** *WebAssembly*]
[**TODO** *opis technologii*]

4.5. Class diagram

[**TODO** *Diagram "klas"*]

4.6. 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

[**TODO**]

Rollup

[**TODO**]

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

[**TODO**]

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

[**TODO**]

6. Project quality study

6.1. Business logic testing

6.1.1. Unit testing

[**TODO** *cargo test*]

6.1.2. Integration testing

[**TODO**]

6.2. UI testing

[**TODO** *Jest*]

6.3. Benchmarking

[**TODO**]

6.4. Complexity analysis

[**TODO** *clippy*]
[**TODO** *liczba linii kodu*]

7. Deployment

7.1. GitHub Pages

[TODO]

7.2. Electron

[TODO]

Summary

[TODO]

Bibliography

[1] Aho, A. V. Algorithms for finding patterns in strings. In *Algorithms and Complexity*. Elsevier, 1990, pp. 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 (Sept. 1956), 113–124.

[4] Couprie, G. Nom, a byte oriented, streaming, zero copy, parser combinators library in Rust. In *2015 IEEE Security and Privacy Workshops* (May 2015), IEEE.

[5] Dib, F. Regex101 homepage. `https://regex101.com/`. Accessed: 2020-10-24.

[6] Fokker, J. Functional parsers. In *Advanced Functional Programming*. Springer Berlin Heidelberg, 1995, pp. 1–23.

[7] Ford, B. Parsing expression grammars. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '04* (2004), ACM Press.

[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 (Dec. 1968), 805–813.

[12] Klabnik, S., and Nichols, C. *The Rust programming language*. No Starch Press, Inc, San Francisco, 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, Taylor & Francis Group, Boca Raton, 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. In *Language Engineering and Rigorous Software Development*. Springer Berlin Heidelberg, 2009, pp. 252–300.

List of Figures

List of Tables

List of Listings

A. Modified specification

[TODO]

```
character
  = ? any Unicode non-control character ?;
letter
  = ? any Unicode alphabetic character ?;
digit
  = ? any Unicode numeric character ?;
whitespace
  = ? any Unicode whitespace character ?;
comment
  = '(*', {comment | character}, '*)';
gap
  = (whitespace | comment), {whitespace}, {{comment}, {whitespace}};
identifier
  = letter, {{whitespace}, letter | digit};
factor
  = [[gap], digit, {{whitespace}, digit}, [gap], '*'],
    [gap], [(identifier
      | ('[' | '(/'), alternative, (']' | '/)')
      | ('{' | '(:'), alternative, ('}' | ':)')
      | '(', alternative, ')'
      | "'", character - "'", {character - "'"}, "'"
      | '"', character - '"', {character - '"'}, '"'
      | '?', {{whitespace}, character - '?'}, '?'), [gap]];
term
  = factor,
    ['-', ? a factor that could be replaced
      by a factor containing no identifiers ?];
sequence
  = term, {',', term};
alternative
  = sequence, {('|' | '/' | '!'), sequence};
production
  = [gap], identifier, [gap], '=', alternative, (';' | '.'), [gap];
grammar
  = production, {production};
```

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