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	....
	Title/degree/name and surname

The final evaluation of the thesis

Chairman of the Diploma Examination Committee	....	....	....
	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. 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. Zaproponowano definicję gramatyki tej notacji, która jest tworzona w wyniku analizy syntaktycznej za pomocą techniki zwanej *kombinacją parserów*. Opisano metodę 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, zaprojektowano aplikację webową, która służy za bardziej wizualne, przyjazne i łatwo dostępne dla użytkownika narzędzie. Praca opisuje wdrażanie aplikacji na usługę hostingową dla statycznych stron. Zaprojektowany i wdrożony system daje możliwość rozszerzenia go o inne specyfikacje gramatyk.

Contents

Thesis structure

[TODO]

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, suffer from 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. 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?*]

[**TODO** *wspomnieć o nazwie Parser-parser gdzieś*]

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, 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 detecting left recursion in the resulting grammar [**TODO** *na pewno?*], as well as a dependency graph reduction algorithm for determining the starting rule of a grammar [**TODO** *na pewno?*]. 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. The scope of the thesis includes the implementation of a simple command line REPL program for interfacing with the grammar parser and checker.

[**TODO**
- *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.

1.4. Glossary

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

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

parser **[TODO]**,

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

DFA **[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 [21]. 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 [4] 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.

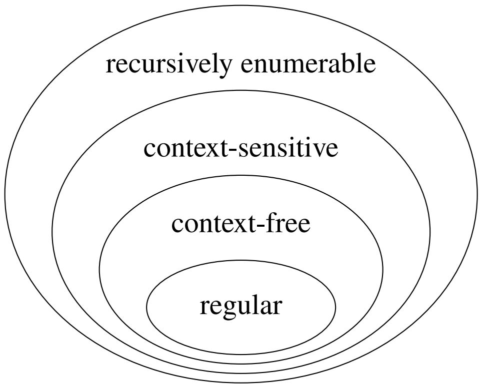

Figure 2.1: The Chomsky Hierarchy visualized.

Type-3 grammars generate the so-called *regular languages*. As described in [2], 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 [18].

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*]

[**TODO** *[16, 27]*]

2.1.3. Parsing Expression Grammars

[**TODO** `https://en.wikipedia.org/wiki/Parsing_expression_grammar`]
[**TODO** *[13]*]

2.2. Why EBNF?

[**TODO**]

2.3. Modifying the specification

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

2.4. Lexical analysis

Table 2.1: [**TODO**]

<table>
<tr><th>Token name</th><th>Normal representation</th></tr>
<tr><td>Non-terminal</td><td>[TODO]</td></tr>
<tr><td>Terminal</td><td>[TODO] surrounded by either “'”s or “"”s</td></tr>
<tr><td>Special</td><td>[TODO]</td></tr>
<tr><td>Integer</td><td>[TODO]</td></tr>
<tr><td>Concatenation</td><td>“,”</td></tr>
<tr><td>Definition</td><td>“=”</td></tr>
<tr><td>Definition separator</td><td>“|”, “/”, or “!”</td></tr>
<tr><td>End group</td><td>“)”</td></tr>
<tr><td>End option</td><td>“]” or “/)”</td></tr>
<tr><td>End repeat</td><td>“}” or “:)”</td></tr>
<tr><td>Exception</td><td>“-”</td></tr>
<tr><td>Repetition</td><td>“*”</td></tr>
<tr><td>Start group</td><td>“(”</td></tr>
<tr><td>Start option</td><td>“[” or “(/”</td></tr>
<tr><td>Start repeat</td><td>“{” or “(:”</td></tr>
<tr><td>Terminator</td><td>“;”</td></tr>
</table>

[**TODO**]

2.5. Methods of syntactic analysis

[**TODO** *[1]*]

2.5.1. Bottom-up parsing

[**TODO**]

2.5.2. Top-down parsing and parser combination

[**TODO** *opisanie parser combinatorów (w Haskellu?) [32] [20] [12]*]

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

3. Analysis of similar solutions

Regex101

paraphraseRegex101 [9] is an interactive console that lets the user debug regular expressions in real-time. Users can build their expressions and see how it affects a live data set all in one screen at the same time. The tool was created by Firas Dib, with contributions from many other developers. It is said to be the largest regex testing service in the world.

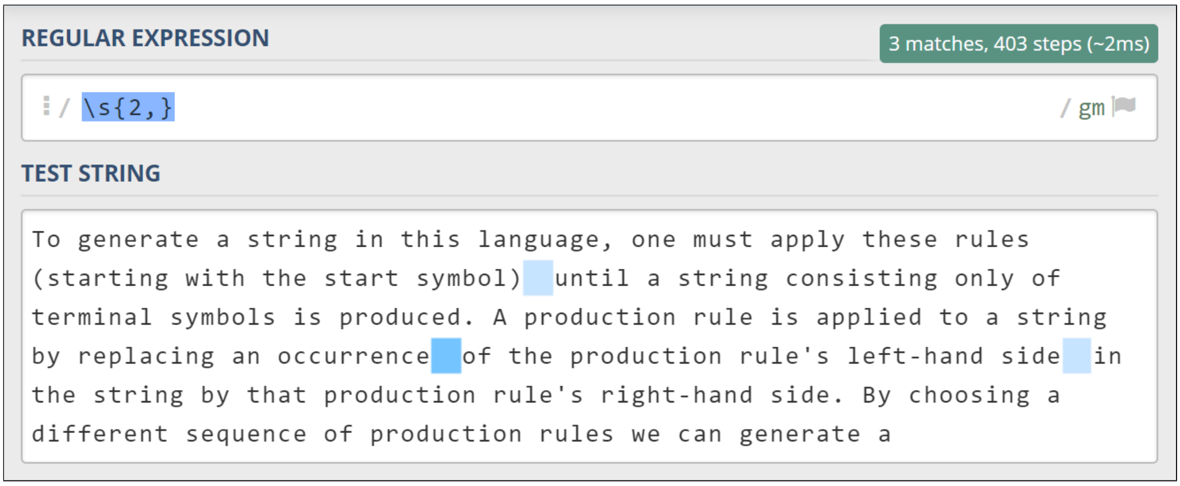

Figure 3.1: Screenshot of the Regex101's matching functionality. The user provided the "\s\s+" regular expression, which matched every occurrence of two or more consecutive space characters in the test string.

The tool is available to users in the form of a web application and can be accessed from `https://regex101.com/`. It lets users build expressions fast and debug them along the way, for example by pasting in a set of data and then, through trial and error, building an expression with desired behavior. Figure 3.1 shows a typical usage of Regex101 — matching a pasted test string to a regular expression. The tool makes it clear if data is matching the expression or not, it even notifies users when the expression is broken, and gives some explanation of why it is not working, as seen in figure 3.2.

These two feedback mechanisms are really helpful if the user is not accustomed with the regular expression language, or just does not know how to build the right expression yet. Being able to trace each step of the expression is a true lifesaver when users are not able to figure out why something is not working, or even if they are simply interested in learning more about regular expressions. Getting this instant feedback without Regex101 would have required users to write their expressions in a text editor and then run the code separately, without getting much feedback about why it is or isn't working. Regex101.com eliminates this mystery.

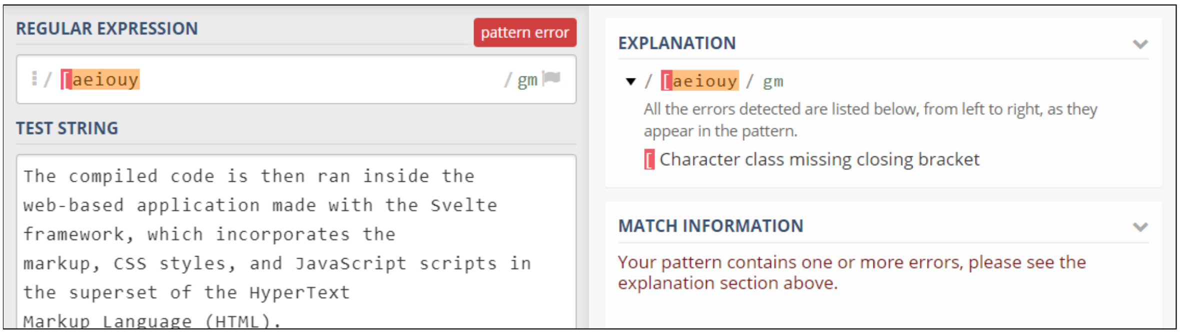

Figure 3.2: Screenshot of a basic error in the regular expression reported by Regex101.

Not only does Regex101.com make it easy to build expressions, find errors, and even learn the syntax, it makes looking up a token or character in regular expressions very easy. Always present, unless users minimize it, the *Quick Reference* tool lets them look up any token or character they need. Finally, Regex101 lets users switch which *flavor* or version of regular expressions they wish to use, as they might need to integrate a regular expression expression into any number of other programming languages such as Python, JavaScript, Golang, etc. Regex101 has the ability to change the version of the testing environment and will generate the code in that language for the user to use in other projects.

Parser-parser takes a lot of inspiration from Regex101 when it comes to availability — it's a web application, where all the work is done client-side. The user does not have to install any additional software except the web browser, the web application is accessed through a web page. In spite of its similar nature, Regex101 cannot be a replacement of Parser-parser — it focuses on various dialects of regular expressions rather than parsing EBNF and generating parse trees — it does, however, influence it with its accessibility and functionalities.

Pest

Pest [23] is a general purpose parser for the Rust programming language. It uses its own dialect of *parsing expression grammars* as input, similarly to Parser-parser. Pest addresses the problem of hand-written parsers in Rust, which in some circumstances can become hard to maintain by their developers. Writing a specialized, domain-specific parser for a language can become tedious, so developers usually gravitate towards using a grammar-generated parser. This allows the developers to focus on the definition of the language, rather than on the implementation of the parser. Grammars which define the language offer better correctness guarantees, and issues can be solved declaratively in the grammar itself. Rust's memory safety further limits the amount of damage bugs can do. High-level static analysis and careful low-level implementation build a solid foundation on which serious performance tuning is possible.

Developers of Pest, in spite of focusing mainly on the functionalities in the Rust programming language, also provide an online editor available from the browser on the Pest's homepage (`https://pest.rs/#editor`). The online editor allows potential future users of Pest to experience the syntactic characteristics of the Pest's dialect of PEGs and its error reporting capabilities. The editor will inform the user about any syntactic errors, as well as errors of semantic nature, such as undefined or left-recursive production rules (seen in figure 3.3) and highlight them in their exact locations.

After parsing the grammar, Pest provides a window, which acts as an input console, where users can type string that may or may not be parsed by the parser generated by Pest.

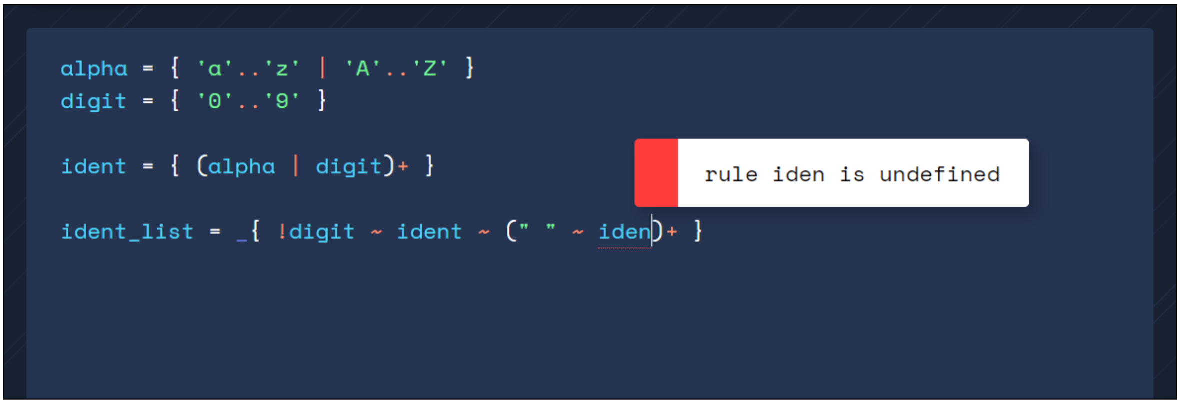

Figure 3.3: Screenshot of Pest's online editor example error report.

Additionally, users can choose the initial production rule from a dropdown menu, which is an interesting choice, as opposed to automatically detecting the initial rule based on the dependency graph of production rules. The output window presents the parse tree, or the errors encountered in the input string in case there are any. These features can be seen in figure 3.4.

Figure 3.4: Screenshot of the input and output windows in Pest's online editor.

While Pest focuses mainly on its integration with the Rust programming language, this is not the case for Parser-parser, which aims to provide all of its functionality inside the web application. The online editor of Pest serves largely as a "try me" feature for new users, rather than a reliable tool. The editor lacks the standard editor features, such as autocompletion, code folding, search and replace interface, as well as bracket and tag matching, all of which Parser-parser does provide. The parse tree in the output window is shown in a basic textual form, without any interactive capabilities, which the user may value. Finally, while Pest's grammar is based on PEGs, and is similar in nature to EBNF, it is, in fact, not EBNF. The whole point of using EBNF and other notations discussed in section 2.2 is that they're standardized, well-known, and accepted by the community; Pest's syntax is known only to users of Pest and requires them to learn a new, non-standard language just for the purpose of parsing grammars, where other, already established languages may have sufficed.

4. Design of the project

This chapter introduces a specification for the application described in chapter 1. The specification is presented in forms of a list of functional and non-functional requirements in section 4.1, and user stories in section 4.2. Section 4.3 describes use cases and their descriptions structured in the form of a use case diagram in the Unified Modeling Language, as well as their example scenarios, also presented as activity and sequence diagrams. The chapter describes the architecture of the system from the logical and physical perspective as component and deployment diagrams in section 4.4. The chapter does not cover any class or database diagrams, as the implementation of this project and its functionally-oriented nature, as opposed to being object-oriented, does not require them. Finally, the chapter concludes with the prototype and sketches of the user interface for the web application in section 4.5.

4.1. Requirements

4.1.1. Functional requirements

Functional requirements shown in table 4.1 define functionalities and features of the system. Each requirement is associated with a certain priority.

Table 4.1: The functional requirements of the project, their features, and priorities.

Id	Requirement	Features	Priority
FR1	Specifying the grammar	The user can specify the grammar of a given language in the EBNF notation by providing it in a textual form in a designated editor window.	high
FR2	Error reporting	The editor provides feedback about any syntactic or semantic[1] errors encountered during the parsing by highlighting the exact location of the error in the provided grammar. The user can then hover the mouse pointer over the highlighted area to read the error message.	high
FR3	Specifying the input string	The user can specify the input string in a designated editor window to check if it belongs to the language generated by the previously-defined grammar.	high
FR4	Visualizing the parse tree	The application visualizes the parse tree resulting from parsing the specified input string with the parser generated by the grammar defined by the user.	high

[1]Such as production rule duplication or left recursion.

FR5	Syntax highlighting	The editor highlights parts of the specified grammar with a different syntactic meaning in a different manner with the use of multi-colored fonts.	medium
FR6	Autocompletion of non-terminals	The editor predicts the identifier of a non-terminal a user is typing by providing a list of possible non-terminals, which then can be chosen by the user.	low
FR7	Production rule folding	The editor provides the ability to hide and reveal a production rule of the grammar inside the editor window.	low
FR8	Search and replace interface	The user can search for any occurrences of a phrase in the editor window and possibly replace them with a different phrase. The search and replace functionality should also support regular expressions.	low

4.1.2. Non-functional requirements

Table 4.2 describes requirements of the non-functional nature of the system, which focus on aspects of usability, availability, and compatibility of the system.

Table 4.2: The non-functional requirements of the project and their priorities.

Requirement	Priority
The web application should be available 24 hours a day, 7 days a week.	medium
Page loading time should be less than 1 second with internet download speed of 80 Mbps. Parsing and checking times should both be less than 50 milliseconds.	high
The application must work and display correctly in • Chrome version 86 or later, • Safari version 14 or later, • Edge version 86 or later, • Firefox version 82 or later, • Opera version 71 or later.	high
	medium
The source code of the product should be open source and freely available for possible modification and redistribution.	high
The project should include the documentation necessary for extension and maintenance of the system.	high
The system should provide high degree of integrability with future components which extend the functionalities of the system.	high

4.2. User stories

Stories in table 4.3 are short descriptions of a feature told from the perspective of the person who desires a new functionality in the system.

Table 4.3: The user stories.

Id	User story
US1	As the user, I want to be able to paste the contents of my clipboard into the editor window in the application.
US2	As the user, I want to be able to type in the editor window with my keyboard.
US3	As the user, I want to be able to appreciate the multi-colored appearance of the text that represents the syntax that I provided.
US4	As the user, I want to be able to select a portion of the text in the editor window and copy it to the clipboard using a keyboard shortcut.
US5	As the user, I want to be able to hold the *Alt* key on my keyboard to create multiple cursors in the editor window.
US6	As the user, I want to have the ability to autocomplete the non-terminal I am typing that has already been declared elsewhere in the code.
US7	As the user, I want to be able to hide any existing production rules that might appear too long, to increase the degree of clarity and readability of the grammar I'm working on.
US8	As the user, I want to be able to show any previously hidden production rules of the grammar.
US9	As the user, I want to have the ability to press a certain key combination on my keyboard that would allow me to type a specific phrase in the popup window, which would then find all the occurrences of that phrase in the editor window.
US10	As the user, I want to be able to provide a regular expression for the *find* functionality that would allow me to find all occurrences of phrases that pattern match that specific regular expression.
US11	As the user, I want to be able to replace some of the occurrences of phrases found with the *find* functionality with another phrase provided in a popup window.
US12	As the user, I want to be able to specify the initial production rule in the process of checking the input string against the grammar I provided.
US13	As the user, I want to be able to see errors in the syntax of the provided grammar in the form of underlined text in the location of where the errors actually occur.
US14	As the user, I want to have the ability to hover the mouse pointer over the underlined text to read the error message at that location. Alternatively, I want to be able to hover over the error indicator, which appears next to the line number.
US15	As the user, I want to be able to see the parse tree of the recognized input string that I provided.
US16	As the user, I want to have the ability to collapse any nodes in the visualized parse tree that might appear too long.

4.3. Use case specification

4.3.1. Use cases

Figure 4.1 shows the use case diagram of the system. Each use case also presented in table 4.4 along with a short description.

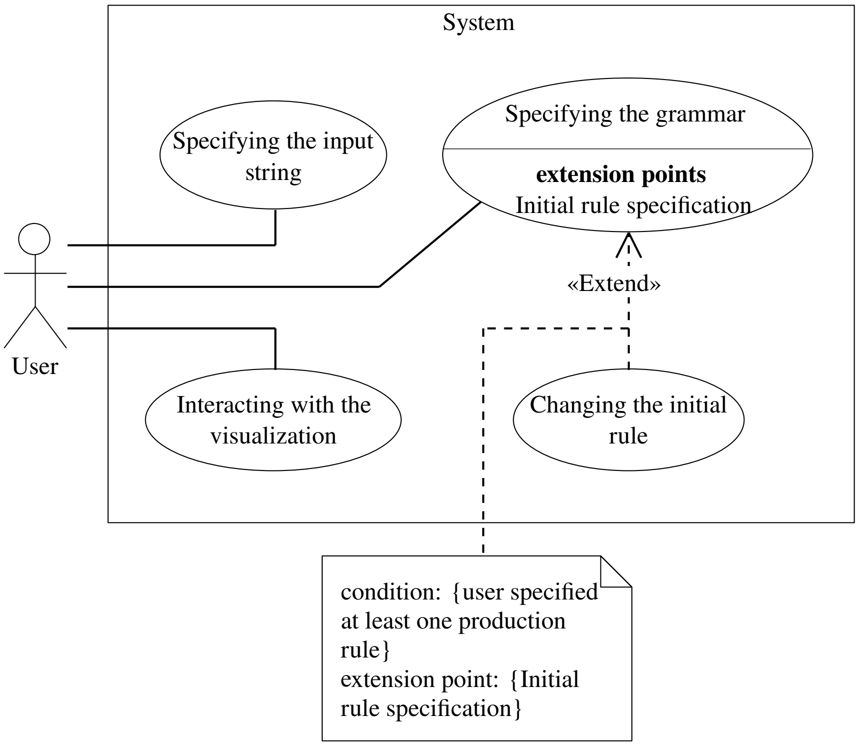

Figure 4.1: The use case diagram.

Table 4.4: Descriptions of the use cases.

Id	Name	Description
UC1	Specifying the grammar	Allows the user to specify the grammar of a given language in the EBNF notation by providing it in a textual form in a designated editor window.
UC2	Specifying the input string	Allows the user to specify the input string in a designated editor window to check if it belongs to the language generated by the previously-defined grammar.
UC3	Interacting with the visualization	Allows the user to observe the visualized parse tree of the provided input string and interact with it by expanding and collapsing the tree nodes.
UC4	Changing the initial rule	Allows the user to specify the initial production rule used in the process of checking the provided input string against the defined grammar.

4.3.2. Requirements traceability graph

Figure 4.2 presents the relationship between functional requirements, user stories and use cases in the form of a requirements traceability graph. It shows that every user story is connected with at least one functional requirement and vice versa, and that every use case is associated with at least one user story and vice versa.

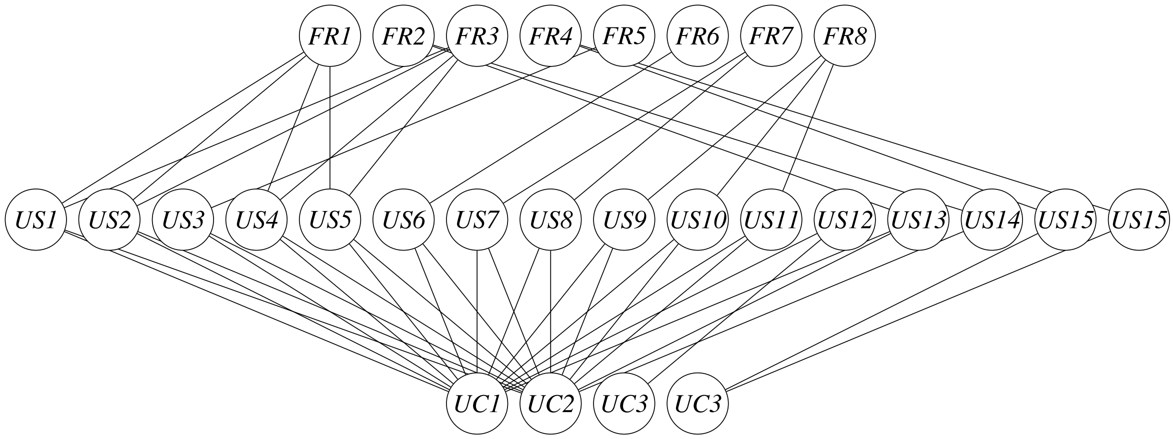

Figure 4.2: The requirements traceability graph.

4.3.3. Use case scenarios

Tables 4.5, 4.6, 4.7, and 4.8 describe the scenarios of each use case in the system. Every scenario is defined by its pre-conditions, its post-conditions, and a list of steps made by the system or the user required to complete it.

Table 4.5: Use case scenario of *UC1* Specifying the grammar.

Identifier	*UC1*
Name	Specifying the grammar
Summary	Allows the user to specify the grammar of a given language in the EBNF notation by providing it in a textual form in a designated editor window.
Pre-conditions	None.
Post-conditions	The grammar has been correctly defined by the user with no syntactic errors.
Main scenario	1. The system shows a grammar editor window to the user. 2. The user provides a syntactically and semantically correct definition of a grammar. 3. The system shows an icon indicating no errors detected in the grammar. End of scenario.
Alternative scenario	2a.1. The user provides an invalid definition of a grammar. 2a.2. The system highlights the text in the grammar editor window at the error location. Return to step 2.

Table 4.6: Use case scenario of *UC2* Specifying the input string.

Identifier	*UC2*
Name	Specifying the input string
Summary	Allows the user to specify the input string in a designated editor window to check if it belongs to the language generated by the previously-defined grammar.
Pre-conditions	None.
Post-conditions	The input string has been correctly entered by the user.
Main scenario	1. The system shows a input string editor window to the user. 2. The user provides a desired input string. 3. A valid grammar has been provided by the user in the grammar editor window. 4. The system shows the result of the checker in the result window. End of scenario.
Alternative scenario	3a.1. The user did not provide a valid grammar in the grammar editor window. 3a.2. The system does not show a result of the checker. End of scenario.

Table 4.7: Use case scenario of *UC3* Interacting with the visualization.

Identifier	*UC3*
Name	Interacting with the visualization
Summary	Allows the user to observe the visualized parse tree of the provided input string and interact with it by expanding and collapsing the tree nodes.
Pre-conditions	The user has provided a valid definition of a grammar, as well as an input string, that belongs to the language generated by that grammar.
Post-conditions	None.
Main scenario	1. [**TODO**] 2. [**TODO**] 3. [**TODO**] End of scenario.

Table 4.8: Use case scenario of *UC2* Specifying the input string.

Identifier	*UC4*
Name	Changing the initial rule
Summary	Allows the user to specify the initial production rule used in the process of checking the provided input string against the defined grammar.
Pre-conditions	The user has provided a valid definition of a grammar.

Post-conditions	The initial production rule has been successfully changed to the desired one.
Main scenario	1. The system shows a button the current initial production rule written on top. 2. The user clicks on the button. 3. The system shows a dropdown menu with a list of all production rules defined in the provided grammar. 4. The user clicks on an item of the list corresponding to the desired initial production rule. 5. The system changes the identifier of the initial production rule on the button. End of scenario.

4.3.4. Activity diagrams

Figures 4.3, 4.4, 4.5, and 4.6 are the graphical representations of use case scenarios defined in subsection 4.3.3, represented in the form of UML activity diagrams.

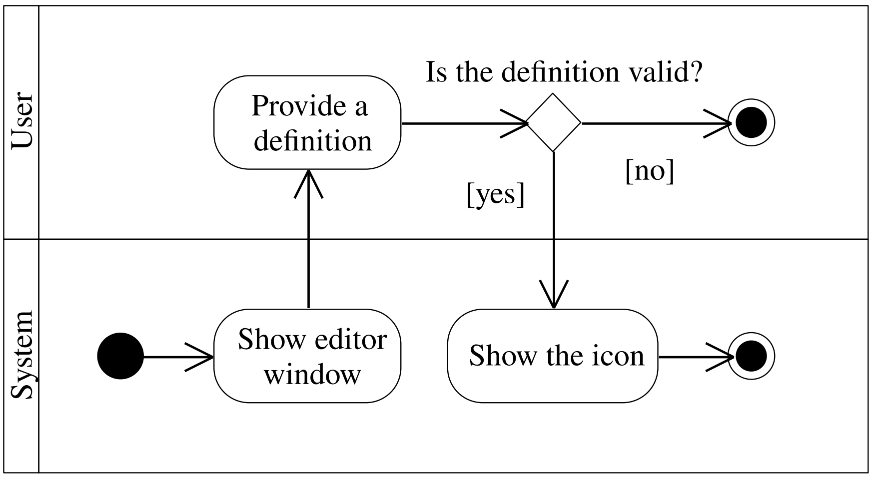

Figure 4.3: The activity diagram of *UC1* Specifying the grammar.

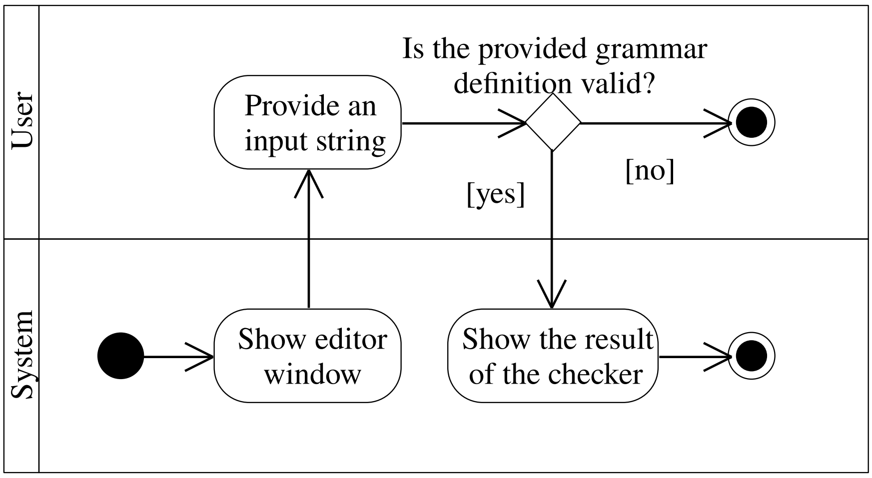

Figure 4.4: The activity diagram of *UC2* Specifying the input string.

[TODO]

Figure 4.5: The activity diagram of *UC3* Interacting with the visualization.

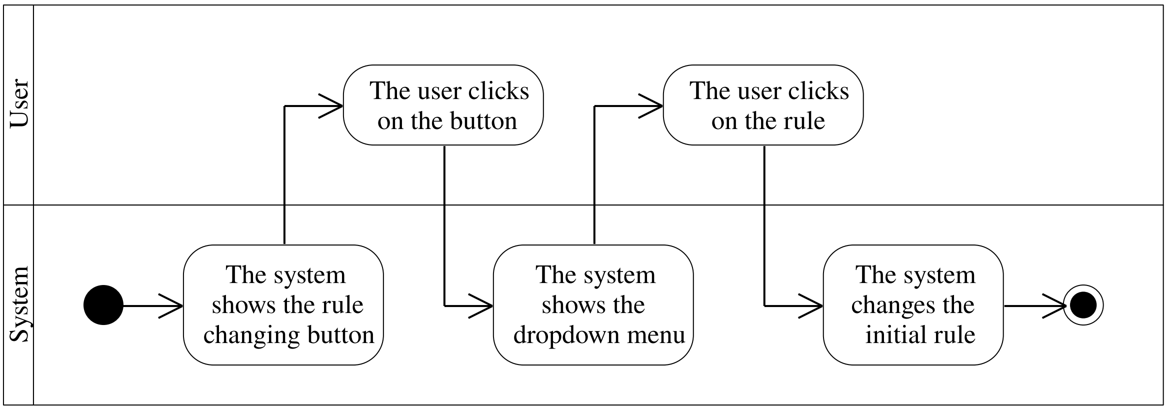

Figure 4.6: The activity diagram of *UC4* Changing the initial rule.

4.3.5. Sequence diagram

Figure 4.7 shows a sequence diagram, that is in essence an interaction diagram that details how operations in the system are carried out and visualizes interactions between objects and components. It captures interactions from every use case, all of which were defined in subsection 4.3.1.

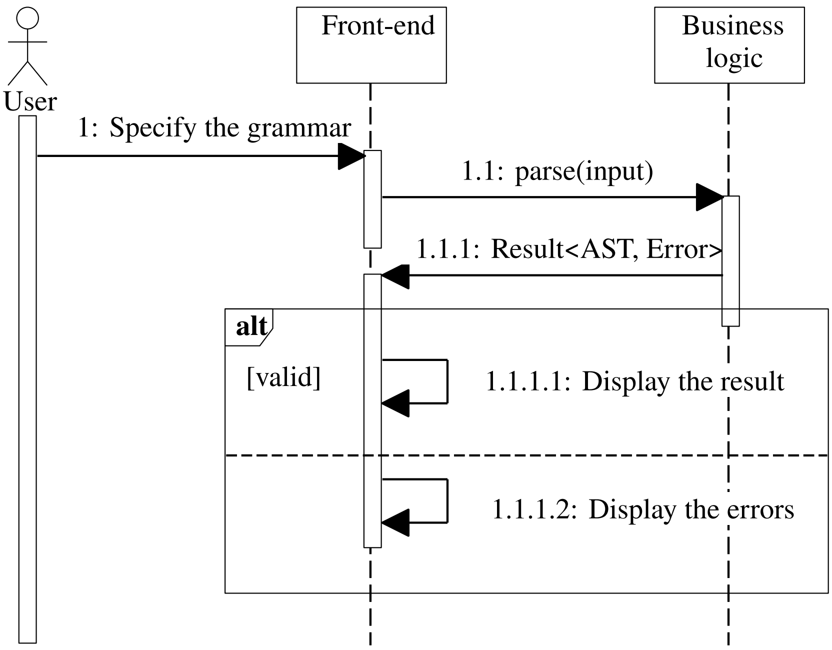

Figure 4.7: **[TODO]**

4.4. System architecture

4.4.1. Logical architecture

Logical architecture of a system can be represented by UML component diagrams, which focus on a system's components that are often used to model the static implementation view of a system. A component diagram breaks down the system into various high levels of functionality. Each component is responsible for one clear aim within the entire system and only interacts with other essential elements on a need-to-know basis. In a system with a functional-oriented approach it is more suitable for modelling interactions between components. The logical architecture of Parser-parser is modelled with such a diagram and can be seen in figure 4.8.

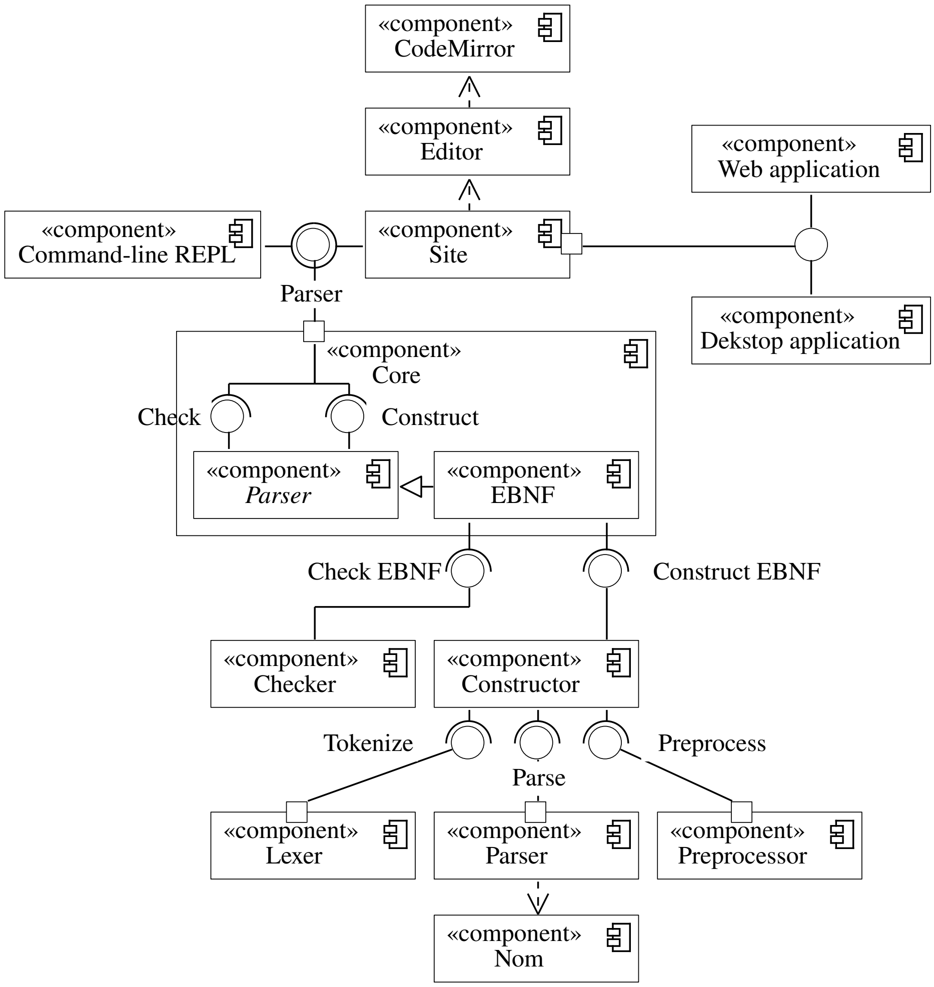

Figure 4.8: The logical architecture of the system represented with a UML component diagram.

4.4.2. Physical architecture

A deployment diagram in the Unified Modeling Language models the physical deployment of artifacts on nodes and can represent a physical architecture of a system. Diagram shown on figure 4.9 visualizes the architecture for Parser-parser.

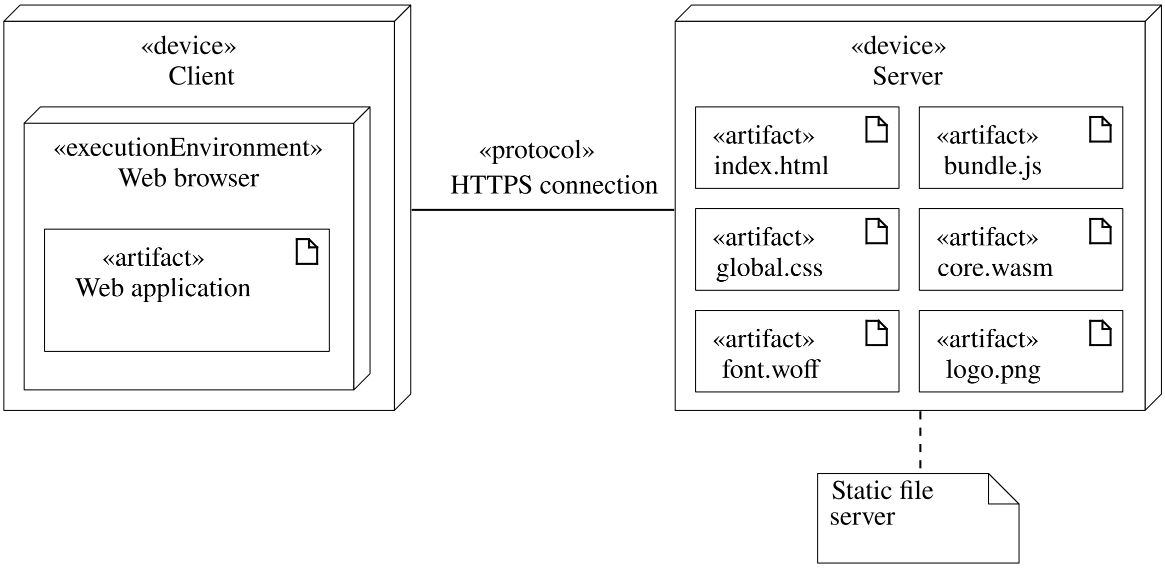

Figure 4.9: The physical architecture of the system represented with a UML deployment diagram.

4.5. Interface prototype

Because of the visual nature of the web application, a prototype of the user interface design should be established to allow the developer to plan out the implementation of the front-end aspect of the application. Parser-parser, being a rather simple application, will consist of a single view (as seen in figure 4.10), which is made out of several components.

[TODO]

Figure 4.10: The user interface sketch.

5. Implementation of the project

5.1. Software environment

5.1.1. Used technologies

Visual Studio Code

Visual Studio Code [22] is a free, open-source text editor made by Microsoft for Windows, Linux and macOS. It is designed to write code and features syntax highlighting, code completion, snippets, code refactoring, and code debugging. The editor can be used with various programming languages, and supports extensions, which can be installed through a central repository called VS Code Marketplace available in the editor itself. The extensions may provide feature additions to the editor, as well as the support for various programming languages in the form of code linters, static code analysers, and debuggers. The editor is integrated with various version control systems, including Git and Subversion

According to the 2019 Developers Survey of Stack Overflow, Visual Studio Code ranked #1 among the top popular developer tools, with 50.7 % of the 87317 respondents using it. [29]

The extensions for the editor are created by the members of Visual Studio Code community. Two main extensions used by the author to develop the project were:

rust-analyzer [11] An implementation of the Language Server Protocol for the Rust programming language, which provides features such as code completion, messages for syntax and semantic errors, code actions, diagnostics, “go to definition” and other editor actions.

Svelte for VS Code [31] An implementation of the Language Server Protocol for the Svelte framework. The extension provides diagnostic messages for warnings and errors, support for Svelte pre-processors that provide source maps, as well as the support for Svelte-specific formatting (via prettier-plugin-svelte). Besides the Svelte language, the extension supports features such as hover info, messages for syntax and lint errors, and autocompletions for HTML, CSS/SCSS/LESS, as well as TypeScript and JavaScript.

The extensions have not proven to be crucial for the development of the project, but were an excellent addition to the workflow.

Besides the editor extensions, the terminal integrated with Visual Studio Code editor has been a valuable feature throughout the development process. The command line is a substantial factor in the development of modern applications, so a built-in terminal window allows the user to swiftly switch between the code editor and the command line.

The support for the Git version control system has also been advantageous when it comes to code editing. Every added, modified, or removed line of code is highlighted with an appropriate color in the code editor. This greatly improves the readability of the code, and allows the users to revert the code to its previous state right from the editor without any external tools.

Git

Git [14] is a free and open source distributed version control system. It has been a major part of the development process for the project, and has been used mainly as a tool for keeping track of the changes made to the source code and for integrating features in a smooth, non-disruptive manner.

Git supports branching and merging, which means that several project features may be implemented simultaneously and independently on separate *branches* and then *merged* into the main project. Every major code change has been implemented on a designated branch and was merged into the main branch only after a thorough testing process — this has made parallel development very easy, by isolating new development from finished work. This style of a workflow is known as GitFlow, made popular by Vincent Driessen [10], it has shown itself to be very effective for projects of any scale. Efficient switching between different versions of project files enables developers to work effectively on the project. Git includes specific tools for visualizing and navigating a non-linear development history. The author used [3] as a reference for using the tool.

Git is now the most widely used source-code management tool, with 87.2 % of the 74298 respondents of the 2018 Developers Survey of Stack Overflow reporting that they use Git as their primary source-control system. [28].

The main client of Git used in the project was the command-line tool on the Ubuntu operating system running on Windows Subsystem for Linux. Figure 5.1 shows an example of GitFlow's *feature branches* and changes in the project repository in the Git version control system.

```
* f6282e3 (HEAD -> master, origin/master, origin/HEAD) Fix some clippy warnings
*   1fdf06a Merge pull request #6 from karolbelina/feature/checker
|\
| * 7c0f2e4 (origin/feature/checker, feature/checker) Add a basic checker
|/
* 9910c79 Unify the spanning with the Spanned struct
| * d7874b0 Updates
| * 1cf0984 (origin/prototype/inline_spans, prototype/inline_spans) Inline the span attributes for AST nodes
|/
* 948a23f Fix the production and alternative parser
*   bbf8086 Merge pull request #4 from karolbelina/feature/unicode_code_points
|\
| * 33d6747 (origin/feature/unicode_code_points, feature/unicode_code_points) Remove the unused imports
| * 7cca55e Add line and column properties to the structures
| * 291994c Refactor the lexer to use Unicode graphemes
|/
* 429395e Add multiline tests
* 93fb739 Fix some edge cases for the lexer
* 3ea0bb0 Move the EBNF parser-parser into a separate crate
* 4ea2ff4 Redefine the wasm-bindgen exports
* 3d83b22 Add the JetBrains Mono font
* 3fe43d7 Add error tooltips on hover
```

Figure 5.1: Screenshot of the command-line interface of the Git version control system.

GitHub

GitHub [15] is a for-profit company owned by Microsoft that offers a cloud-based Git repository hosting service. As a company, GitHub makes money by selling hosted private code repositories, as well as other business-focused plans that make it easier for organizations to manage team members and security. The author used the free GitHub plan as the service for hosting the project's Git repository. Having the source code on an external server protected the project against data loss and allowed the developer to work on the project from any device at any convenient time.

In addition to using GitHub as a hosting service, one can also exploit its project management features. Developers can create project boards related to the project's code repository, which are simple kanban board that can help organize and prioritize the work. With projects, the developers have the flexibility to manage boards for an entire project, or just for specific features. Figure 5.2 shows an example project board from Parser-parser.

Project boards contain *issues* and *pull requests*, which can be moved from one kanban column to another, indicating that some work is currently "to do", work in progress, or complete. These work "cards" contain information about the author, assignees, the status, as well as simple textual notes. The *issues* are a way of reporting ideas, bugs, enhancements, or tasks natively on GitHub. After completing the work on an issue, a developer might create a *pull request* to allow other developers on the project to review and discuss the changes made to the code, and then deploy the changes by "pulling" the code to the central code repository.

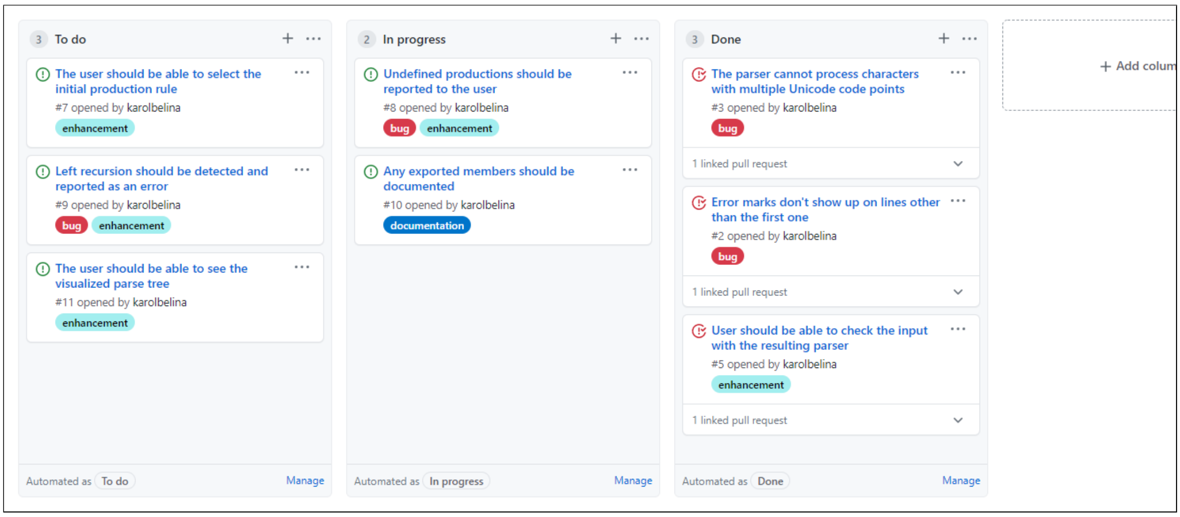

Figure 5.2: Screenshot of one of the project's kanban boards on GitHub.

GitHub supports Continuous integration and Continuous Delivery functionalities in form of *Actions* and *Pages*. GitHub Actions are a way to automate and execute any software development workflow after any change to the code in the repository. The user may set up many various actions for testing the changes on many development environments and operating systems at the same time, as well as building and deploying the code as a package or an arbitrary artifact. An action consists of jobs, which are defined by a list of steps required to execute them.

The GitHub Actions are used by the author to automate the testing and build process on every change made to the code repository. The built application is then deployed to a static site hosting service called GitHub Pages, which integrates itself seamlessly with Actions and GitHub repositories. GitHub Pages allows the user to host a website directly from a GitHub repository by combining static HTML, CSS, JavaScript, and other files straight from a repository into a website and publishing it on a `github.io` domain or a custom one.

Rust

Rust [26] is the main programming language used in Parser-parser — it powers the business logic part of the project. The language has been the most loved language for four years in a row in the Stack Overflow's survey [29]. The core idea of the language is memory safety — the language enforces certain rules checked at compile time, which guarantee that the program is safe from bugs like dereferencing null or dangling pointers, as well as making it difficult for the programmer to leak memory. Rust does this through a system of ownership and borrowing. The language, besides the safety, focuses on speed — its design lets the developer create programs that have the performance and control of a low-level language, but with the powerful abstractions of a high-level language.

Rust's design borrows heavily from the one of Haskell — both languages feature a rich type system, both are immutable-by-default, avoid mutation of shared references et cetera. Many developers tend to write Rust code in a functional style and adhere to the principles of functional programming, even though the language is multi-paradigm.

Without the need of a garbage collector, Rust projects are well-suited to be used as libraries by other programming languages. The language over the last few years has manifested itself in several distinct domains, including command line tools, networking, and embedded systems. Rust is supported on multiple operating systems and targets multiple platforms, has notable documentation, a user-friendly compiler with convenient error messages, and excellent tooling and ecosystem. For referencing the language, the author used [19], which covers many features and concepts of Rust.

WebAssembly

WebAssembly [34] (abbreviated *Wasm*) is a safe, portable, low-level code format designed for efficient execution and compact representation. Its main goal is to enable high performance applications on the Web, working alongside JavaScript, but not to be a replacement of it. It is designed to be portable, compact, and execute at or near native speeds. Although it has currently gathered attention in the JavaScript and Web communities in general, Wasm makes no assumptions about its host environment. WebAssembly is supported as a target for many programming languages, including C♯ via Blazor, C++ via EmScripten, and the main language used in Parser-parser — Rust. The author compiles Rust code to WebAssembly to be then used in a web environment for several reasons:

- Code size is incredibly important since the `.wasm` file must be downloaded over the network. Rust lacks a runtime, enabling small Wasm sizes because there is no extra code included, like a garbage collector,

- Rust and WebAssembly integrates with existing JavaScript tooling. It supports ECMAScript modules and the developer can continue using the tooling their already use, like npm and Webpack,

- JavaScript Web applications struggle to attain and retain reliable performance. The code is required to be ran frequently, so Wasm can solve this kind of problem with better memory and CPU efficiency at a lower level compared to the JavaScript interpreter.

- The Rust language itself, with a strong package manager, high performance, memory safety, and zero-cost abstractions.

Cargo

Cargo is the Rust's package manager. It downloads the Rust package's dependencies and compiles them, ensuring that the developer will always get a repeatable build. To accomplish this goal, Cargo introduces two metadata files with various bits of package information, fetches and builds the dependencies, invokes the Rust compiler with correct parameters to build the package, and introduces conventions to make working with Rust packages easier.

Rust provides first-class support for unit and integration testing, and Cargo allows the developer to execute all tests with a single command. Additionally, Cargo allows the developer to install extensions, which enhance the workflow and the development process. One of extensions useful for the author was Clippy — a collection of lints to catch common mistakes and improve the Rust code.

`crates.io` is the Rust community's central package registry that serves as a location to discover and download packages. Cargo is configured to use it by default to find requested packages. The project uses several dependencies, the most important of which include:

nom [6, 7] A parser combinators library for Rust. Its goal is to provide tools to build safe parsers without compromising the speed or memory consumption. To that end, it uses extensively Rust's strong typing and memory safety to produce fast and correct parsers, and provides functions, macros and traits to abstract most of the error prone details.

While programming language parsers are usually written manually for more flexibility and performance, nom can be (and has been successfully) used as a prototyping parser for a language. The resulting code is small, and looks like the grammar the developer would have written with other parser approaches. The resulting parsers are small and easy to write, as well as easy to test separately.

unicode-segmentation [33] A library with a set of iterators which split strings on *grapheme clusters*, *words* or *sentence boundaries*, according to the Unicode Standard Annex #29 [8] rules.

wasm-bindgen [25] A Rust library and CLI tool that facilitate high-level interactions between Wasm modules and JavaScript. More specifically, this library allows JavaScript and Wasm to communicate with strings, JS objects, classes, etc, as opposed to purely integers and floats. Notable features of this project include:

- Importing JS functionality into Rust such as DOM manipulation, console logging, or performance monitoring.
- Working with rich types like strings, numbers, classes, closures, and objects.
- Automatically generating TypeScript bindings for Rust code being consumed by JS.

Wasm-bindgen only generates bindings and glue for the JavaScript imports that are actually being used and Rust functionality that is being exported.

quickcheck A property-based testing framework inspired by the QuickCheck framework for Haskell. The crate comes with the ability to randomly generate and shrink integers, floats, tuples, booleans, lists, strings, options and results. All QuickCheck needs is a property function — it will then randomly generate inputs to that function and call the property for each set of inputs. If the property fails (whether by a runtime error like index out-of-bounds or by not satisfying the property), the inputs are "shrunk" to find a smaller counter-example.

criterion This library provides a powerful but simple way to measure software performance. It provides both a framework for executing and analyzing benchmarks and a set of driver functions that makes it easy to build and run benchmarks, and to analyse their results.

All of the above dependencies are available under the MIT license.

Wasm-pack

[TODO]

Svelte

Svelte [30] is a free and open-source front end JavaScript framework. Svelte has its own compiler for converting app code into client-side JavaScript at build time. The developer writes the components using HTML, CSS and JavaScript and during the build process Svelte compiles them into small standalone JavaScript modules. While frameworks like React and Vue do the bulk of their work in the user's browser while the app is running, Svelte shifts that work into a compile step that happens only when the developer builds their app, producing highly-optimized vanilla JavaScript. By statically analysing the component template, the compiler can make sure that the browser does as little work as possible. The outcome of this approach is not only smaller application bundles and better performance, but also a developer experience that is more approachable for people that have limited experience of the modern tooling ecosystem. Svelte is particularly appropriate to tackle the following situations:

- Web applications intended for low power devices: Applications built with Svelte have smaller bundle sizes, which is ideal for devices with slow network connections and limited processing power.
- Highly interactive pages or complex visualizations: If the user is building data-visualizations that need to display a large number of DOM elements, the performance gains that come from a framework with no runtime overhead will ensure that user interactions are responsive.
- Onboarding people with basic web development knowledge: Svelte has a shallow learning curve. Web developers with basic HTML, CSS, and JavaScript knowledge can easily grasp Svelte specifics in a short time and start building web applications.

Being a compiler, Svelte can extend HTML, CSS, and JavaScript, generating optimal JavaScript code without any runtime overhead. To achieve this, Svelte extends vanilla web technologies and only intervenes in very specific situations and only in the context of Svelte components.

Rollup

Rollup [24] is a module bundler for JavaScript which compiles small pieces of code into a complex library or application. It uses the standardized ES module format for code, which lets the developer freely and seamlessly combine individual functions and external libraries. Rollup can optimize ES modules for faster native loading in modern browsers, or output a legacy module format.

By dividing the project into smaller separate pieces, the development process is often times more straightforward, since that usually removes unexpected interactions and dramatically reduces the complexity of the problems the developer needs to solve, and simply writing smaller projects in the first place isn't necessarily the answer. Unfortunately, JavaScript has not historically included this capability as a core feature in the language. This finally changed with the ES6 revision of JavaScript, which includes a syntax for importing and exporting functions and data so they can be shared between separate scripts. The specification is now fixed, but it is only implemented in modern browsers. Rollup allows the user to write code using the new module system, and will then compile it back down to existing supported formats such as CommonJS modules, AMD modules, and IIFE-style scripts. This means that the developer gets to write future-proof code.

In addition to enabling the use of ES modules, Rollup also statically analyzes the imported code, and will exclude anything that isn't actually used. This allows the user to build on top of existing tools and modules without adding extra dependencies or bloating the size of the project. Because Rollup includes the bare minimum, it results in lighter, faster, and less complicated libraries and applications. Since this approach can utilise explicit `import` and `export` statements, it is more effective than simply running an automated *minifier* to detect unused variables in the compiled output code.

npm

Node Package Manager is a package manager for the JavaScript programming language. It consists of a command line client, also called npm, and an online database of public and paid-for private packages, called the npm registry. The registry is accessed via the client, and the available packages can be browsed and searched via the npm website.

Npm provides several built-in scripts and allows users to define their own. An npm script is a convenient way to bundle common shell commands for the project. They are typically commands, or a string of commands, which would normally be entered at the command line in order to do something with the application. Scripts are stored in a project's configuration file, which means they're shared amongst everyone using the codebase. They help automate repetitive tasks, and mean having to learn fewer tools. Scripts also ensure that everyone is using the same command with the same flags. Common use cases for npm scripts include building the project, starting a development server, compiling CSS, linting, or minifying.

The project is dependent on several npm packages:

CodeMirror [5] A versatile text editor implemented in JavaScript for the browser. It is specialized for editing code, and comes with a number of language modes and addons that implement more advanced editing functionality. A rich programming API and a CSS theming system are available for customizing CodeMirror to fit the needs, as well as extending it with new functionality. It is the editor used in the dev tools for Firefox, Chrome, and Safari, in Light Table, Adobe Brackets, Bitbucket, and many other projects.

CodeMirror supports a wide variety of configurations — the basic version of the editor without any addons provides the support for over 100 languages, autocompletion, code folding, configurable keybindings, search and replace interface, bracket and tag matching, support for split view, linter integration, various themes, and many more.

svelte-tree A tree-like view component for Svelte. The component has the ability to display a collapsable tree structure based on a provided tree of JavaScript objects with custom `name` and `children` properties. The component provides a slot space to display custom nodes, which will give the tree node DOM/components the access to the nodes being rendered.

The configuration file also lists dependencies for the development process, which can be divided into several categories:

- Rollup and its plugins
  - `rollup`,
  - `@rollup/plugin-alias`,
  - `@rollup/plugin-commonjs`,
  - `@rollup/plugin-node-resolve`,
  - `@rollup/plugin-typescript`,
  - `@wasm-tool/rollup-plugin-rust`,
  - `rollup-plugin-copy`,
  - `rollup-plugin-css-only`,
  - `rollup-plugin-livereload`,
  - `rollup-plugin-svelte`,
  - `rollup-plugin-terser`
- Svelte and its plugins
  - `svelte`,
  - `svelte-check`,
  - `svelte-loader`,
  - `svelte-preprocess`
- Miscellaneous dependencies
  - `gh-pages`,
  - `prettier`,
  - `prettier-plugin-svelte`,
  - `rimraf`,
  - `sirv-cli`

5.1.2. Technology infrastructure

[**TODO** *użyte technologie i zwizualizowany stack*]

5.1.3. Project structure

[**TODO**]

5.2. Business logic

5.2.1. Domain modelling

Domain Modeling is a way to describe and model entities and the relationships between them, which collectively describe the problem domain space. Types can be used to represent the domain in a fine-grained way. In many cases, types can even be used to encode business rules so that the developer cannot create incorrect code. Static type checking can be used as an instant unit test — making sure that the code is correct at compile time. Types are the laws that dictate what is allowed to happen in the domain, and could be used to prevent anyone else from putting the system in a state invalid to the domain. Making illegal states unrepresentable is all about statically proving that all runtime values correspond to valid objects in the business domain, and that makes the code much easier to reason about — that gives the developer confidence that the business rules are being respected.

If the logic is represented by types, it is automatically self-documenting, and any changes to the business rules will immediately create breaking changes, which is a generally a welcome feature. This way the developer can encode business requirements and create a compiler-enforced documentation in the development process.

Using algebraic data types is a powerful technique for designing with types and making illegal states unrepresentable. Constructs such as sum types and product types provide us with an expressive method of modelling the business rules. This method also allows the developer to utilize property-based testing — letting the computer generate test cases.

For modelling the domain, the author will use the Haskell programming language, with its expressive data types and highly reusable abstractions, as well as a concise syntax. However, modelling based on algebraic data types is also practical for other languages with complex enough type systems — Rust, used as the main language in Parser-parser, is one example of such a language.

Token type definition

The tokenization process, described in section 2.4, converts a stream of characters into a stream of tokens. A set of valid tokens can be represented as a sum type of all individual token types, shown in listing 5.1. Several type constructors carry additional information about the token:

`Non-terminal` is specified by the textual form of the meta-identifier represented by the `String` type,

`Terminal` is specified by the contents of the terminal in the form of a `String`,

`Special` carries with it exact contents of the special sequence specified in the grammar, to be processed further,

`Integer` is specified by an actual numeric value encoded as Haskell's `Integer` type.

```haskell
data Token
  = Nonterminal String
  | Terminal String
  | Special String
  | Integer Integer
  | Concatenation
  | Definition
  | DefinitionSeparator
  | EndGroup
  | EndOption
  | EndRepeat
  | Exception
  | Repetition
  | StartGroup
  | StartOption
  | StartRepeat
  | Terminator
```

Listing 5.1: Definition of the `Token` type in Haskell.

Grammar type definition

```haskell
type Grammar = [Production]
type Production = (String, Expression)
data Expression
  = Alternative Expression Expression [Expression]
  | Sequence Expression Expression [Expression]
  | Optional Expression
  | Repeated Expression
  | Factor Integer Expression
  | Exception Expression Expression
  | Nonterminal String
  | Terminal String
  | Special String
  | Empty
```

Listing 5.2: Definition of the types related to the AST in Haskell.

[**TODO** *opis*]

5.2.2. Lexical analyser

The lexical analyser, also known as *the lexer* or *the tokenizer*, performs the tokenization described in section 2.4. A simplified version of the EBNF tokenizer could be modelled as a Deterministic Finite-state Automaton (DFA) (see figure 5.3). This version, however, would not support nested comments defined in the specification — any nested structure cannot be tokenized with the use of regular languages, and those are, in fact, equivalent to DFAs.

The implementation does not follow the pure DFA approach. Instead, the lexer stores the current state of the tokenization process in various forms. For the purpose of tokenizing comments, the lexer remembers the *nest level*, which essentially counts the number of recursively-nested comments, and ends the comment only when the nest level reaches zero.

The lexer's main control flow is a simple infinite loop followed by a pattern match, whose cases are the initial characters of each token type, and each case may be a loop to consume the rest of the token and return its type. Every token is preceded by the "whitespace-comment-whitespace" search, which skips any whitespace characters and (possibly nested) comments. Any whitespace inside integers or meta-identifiers is handled in the corresponding token loop.

The header of the `lex` function is

```
fn lex(string: &str) -> Result<Vec<Spanned<Token>>, Spanned<Error>>
```

Please note the return type of the function, which is either a `Vec` of `Spanned` `Token`s in case the tokenizer succeeded, or an `Error`, which is defined as

```
enum Error {
  InvalidSymbol(String),
  UnterminatedSpecial,
  UnterminatedComment,
  UnterminatedTerminal,
  EmptyTerminal,
}
```

which encodes every possible error the tokenizer can report. The `InvalidSymbol` case contains the additional information about the actual invalid symbol.

The whole tokenization process is preceded by the procedure of splitting the input string into individual Unicode graphemes according to the Unicode Standard Annex #29 [8] rules with the use of the unicode-segmentation crate. As each grapheme may consist of several characters, it is encoded a string, which means the lexer cannot use native functions for checking if a character is whitespace, alphabetic, alphanumeric, or a digit — these have to be defined separately with graphemes in mind.

The code related to the lexer is contained in the `lexer` module and is split into several files:

`mod.rs` is the main module file with the core business logic of the lexer and contains the `scan` and `lex` functions, as well as the utility functions related to graphemes,

`token.rs` contains the definition of the `Token` type,

`error.rs` contains the definition of the `lexer::Error` type,

`tests.rs` holds unit tests related to lexical analysis, which will be later discussed in section 6.1.

5.2.3. Syntactic analyser

The syntactic analysis phase, also known as the parsing phase (described in section 2.5) is conducted by the parser module. The goal of this thesis is not implementing a parser combinator library from scratch, so instead of writing every parser by hand, Parser-parser uses Nom (see subsection 5.1.1) as the parser combinator library of choice, which ships with a large number of utility parsers and combinators already defined. These are in turn combined into more sophisticated parsers that are capable of parsing certain EBNF-like structures.

For instance, the parser seen in listing 5.3 parses the sequence of `terms` separated by commas with the use of the `separated_list_1` native to Nom. It also utilizes the `map` combinator to transform the result into an appropriate type — it returns the `Sequence` of terms in case it parsed two or more terms, or just the single term in case it was the only one in the sequence. Note that this parser uses the `term` and `concatenation_symbol` parsers defined in a similar fashion.

```rust
fn sequence(i: Tokens) -> IResult<Tokens, Spanned<Expression>, Spanned<Error>> {
  map(
    separated_list1(concatenation_symbol, term),
    |nodes| match nodes.len() {
      1 => nodes[0].clone(),
      _ => Expression::Sequence {
        first: Box::new(nodes[0].clone()),
        second: Box::new(nodes[1].clone()),
        rest: nodes[2..].to_vec(),
      }
      .spanning(Span::combine(&nodes[0].span, &nodes[nodes.len() - 1].span)),
    },
  )(i)
}
```

Listing 5.3: The `sequence` parser.

All the more complicated parsers eventually use the so-called *literals* — the most simple parsers capable of parsing single tokens. Every token from the `Token` type (defined in subsection 5.2.1) has an equivalent literal parser.

All parsers in Nom are basically functions, which most generally take a string as the input, and return an `IResult` — either the parsed *thing* along with the rest of the unconsumed input, or an error of some sort indicating that the parser failed. In the case of Parser-parser, however, the input is not a string of characters, but a string of tokens. Fortunately, Nom, thanks to its high extensibility, can parse any type as long as that type implements certain traits. This resulted in the definition of the `Tokens` data structure, which encapsulates the `&[Spanned<Token>]` type — a slice of a sequence of (spanned) tokens. `Tokens` implements such traits as

- `InputLength`
- `InputIter`
- `InputTake`
- `UnspecializedInput`
- `Compare<Tokens>`
- `Slice<Range<usize>>`
- `Slice<RangeTo<usize>>`
- `Slice<RangeFrom<usize>>`
- `Slice<RangeFull>`
- `FindToken<Spanned<Token>>`

- `FindSubstring<&[Spanned<Token>]>`

The `Tokens` type can then be used by Nom in a very efficient manner — these traits are one of the reasons why Nom is so performant.

The parsers generally return a single AST node as their output. These nodes are then combined into other nodes, which are then combined into `Productions`, and finally into a `Grammar`. In fact, the main `parse` function, the header of which is

```
fn parse(tokens: &[Spanned<Token>]) -> Result<Spanned<Grammar>,
↪   Spanned<Error>>
```

returns a `Grammar`, which contains the information about the whole syntax tree parsed from the provided sequence of tokens.

The code related to syntactic analysis can be found in the `parser` module. It is divided among several files:

`mod.rs` defines all major parsers along with the main `parse` function exported from the module,

`ast.rs` contains the definition of the AST,

`tokens.rs` holds the definition of the `Tokens` type and its trait implementations,

`error.rs` contains the definition of the `parser::Error` type

`utils.rs` defines the utility functions and *literal* parsers,

`tests.rs` contains unit tests related to syntactic analysis, which are going to be discussed in section 6.1.

5.2.4. Preprocessing and detecting left recursion

[**TODO** *wykrywanie niepoprawnych non-terminali*]

In the formal language theory, a production rule is left-recursive if the leftmost symbol of it is itself (in the case of direct left recursion) or can be made itself by some sequence of substitutions (in the case of indirect left recursion). A PEG is called *well-formed* if it contains no left-recursive rules, i.e., rules that allow a non-terminal to expand to an expression in which the same non-terminal occurs as the leftmost symbol.

Consider the following example:

```
integer = ? [0-9.]+ ?
value   = integer | '(', expr, ')';
product = expr, {('*' | '/'), expr}
sum     = expr, {('+' | '-'), expr}
expr    = product | sum | value
```

In this grammar, matching an `expr` requires testing if a `product` matches while matching a `product` requires testing if an `expr` matches. Because the term appears in the leftmost position, these rules make up a circular definition that cannot be resolved.

Left recursion often poses problems for parsers because it leads them into infinite recursion. As left-recursive rules can always be rewritten, a grammar is often preprocessed to eliminate them. Parser-parser, however, does not attempt to do any rewrites. Instead, it only detects the presence of direct or indirect left recursion in the provided grammar. The process of detecting left recursion can be seen in algorithm 5.1.

[**TODO** *struktura plików*]

5.2.5. Grammar processing

After creating the AST of a grammar, it can be analysed along with an input string to check if that input string belongs to the language generated by the grammar. For this purpose, the program recursively checks nodes of the AST to see if they match the currently scanned part of the input string. First, the user must provide the initial production rule, from which the process will begin. The recursive function must either return *success* or *failure* to indicate whether the parsing has succeeded. The program processes the input differently depending on the type of the AST node:

Alternative The program processes each case of the *Alternative* sequentially and returns the first one to return *success*. If no case returned *success*, it returns *failure*,

Sequence The program processes each expression of the *Sequence* sequentially and returns *success* if and only if every processed case returned *success*, otherwise it returns *failure*,

Optional The program processes the expression inside of the *Optional* and returns *success* regardless of the result,

Repeated The program repeatedly processes the expression inside of the *Repeated* and returns *success* regardless of any results,

Factor The program processes the expression inside of the *Factor* N times, where N is the number of repetitions of the *Factor*. It returns *success* if the processing succeeds all N times, otherwise it returns *failure*,

Exception The program processes the *subject* of the *Exception* and in case of *success*, it stores the processed input string to be then processed by the *restriction* of the *Exception*. If the restriction succeeds and the resulting processed input string is the same as the input string in case of the subject, it returns *failure*. In every other case it returns *success*,

Non-terminal The program recursively processes the production rule specified by the identifier inside of the *Non-terminal*,

Terminal The program checks if the processed input string starts with the input specified by the *Terminal* and returns *success* if it does,

Special Currently, the program always returns *failure* for any special sequence,

Empty The program always returns *success* without processing the input string.

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**]

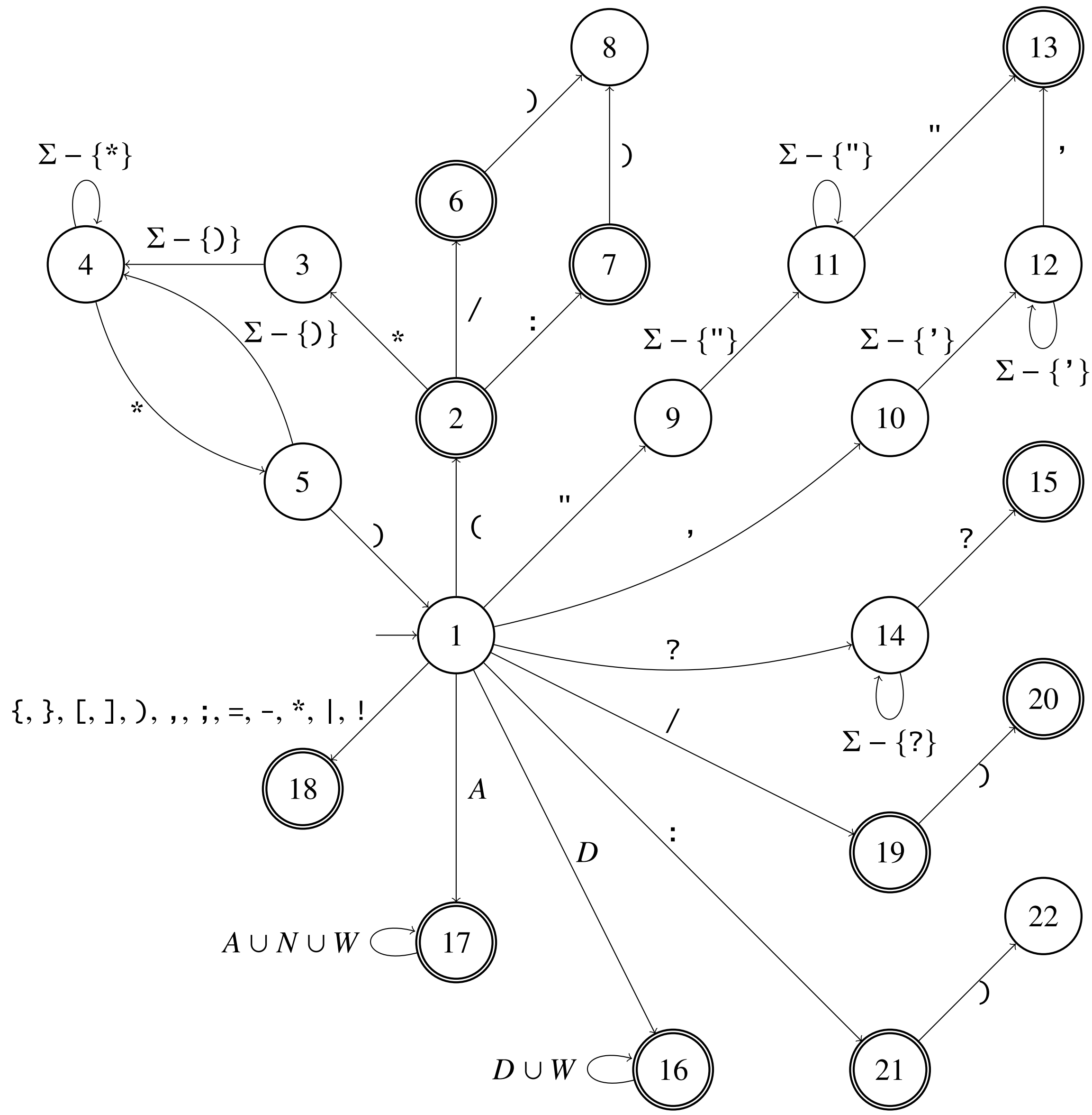

Figure 5.3: The DFA representation of the lexer. Note that this DFA does not support nested comments. Σ is the set of all characters, A is the set of all alphabetic characters, N is the set of all numeric characters, D is the set of all ten decimal digits, and W is the set off all whitespace characters. Set notation on the transitions is omitted.

Algorithm 5.1: Detecting left recursion in production rules R of a grammar.

```
input: Dictionary of production rules R
function CheckExpr(e, t) begin
    inputs: The current expression e; stack of identifiers t
    switch e do
        case Alternative(S) do
            foreach s ∈ S do
                CheckExpr(s, t)
        case Sequence((s_1, s_2, ..., s_n)) do
            /* Skip the last expression */
            foreach s ∈ (s_1, s_2, ..., s_{n-1}) do
                /* check if s can be an empty expression (e.g. Optional) */
                if s ≠ ε then
                    CheckExpr(s, t)
            CheckExpr(s_n, t)                // Check the last expression
        case Optional(s) do
            CheckExpr(s, t)
        case Repeated(s) do
            CheckExpr(s, t)
        case Factor(n, s) do
            if n > 0 then
                CheckExpr(s, t)
        case Exception(s, r) do
            CheckExpr(s, t) CheckExpr(r, t)
        case Nonterminal(i) do
            if i = t[0] then
                t.push(i)
                exit()       // Report the left recursion error with trace t
            if i ∉ t then
                t.push(i)
                CheckExpr(R[i], t)
                t.pop()
foreach (i, e) ∈ R do
    CheckExpr(e, R)
```

6. Project quality study

6.1. Business logic testing

6.1.1. Unit testing

Unit testing is a type of software testing where individual units or components of a software are tested. The purpose is to validate that each unit of the software code performs as expected. Unit tests help to fix bugs early in the development cycle, and help to understand the code base and make changes to the code quickly. Good unit tests also serve as project documentation. Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (i.e. Regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified and fixed. Unit testing, however, can't be expected to catch every error in a program. It is not possible to evaluate all execution paths even in the most trivial programs. Unit testing by its very nature focuses on a unit of code. Hence it can't catch integration errors or broad system level errors.

A developer writes a section of code in the application just to test the function. Isolating the tested code helps in revealing unnecessary dependencies between the code being tested and other units in the project — testing should be focused on only one piece of code at a time. Unit Test cases should be independent. In case of any enhancements or change in requirements, unit test cases should not be affected. If units are made less interdependent to make unit testing possible, the unintended impact of changes to any code is less.

Unit tests in Parser-parser focus on testing the individual components of the EBNF parser, that is: the lexer, the parser, and the preprocessor. If the developer is sure that all of these components work correctly in isolation, the same thing can be said about the whole system, because there are no hidden dependencies between these components.

Tests related to the lexer test the functionality of the lexer, that is, test the ability of the lexer to turn the textual representation of EBNF into individual tokens. An typical unit test related to the lexer can be seen in listing 6.1, where it makes an assertion that the result of the `lex` function on input “`␣abc␣\n␣=␣'def'␣`” is a success (the `Ok` case), and it's a vector of three certain tokens.

```
#[test]
fn test_multiline() {
  assert_eq!(
    lex(" abc \n = 'def' "),
    Ok(vec![
      Token::Nonterminal("abc".to_owned()).spanning(Span::from(((1, 0), (4, 0)))),
      Token::Definition.spanning(Span::from(((1, 1), (2, 1)))),
      Token::Terminal("def".to_owned()).spanning(Span::from(((3, 1), (8, 1))))
    ])
  );
}
```

Listing 6.1: A unit test related to the lexer testing the proper tokenization of the input string.

All test cases related to the lexer are listed in table 6.1 in a simplified format, where the result can either be a success case or a failure case, but does not provide information about the lexed tokens for simplicity. The actual test cases for the lexer are located in the file `ebnf/src/lexer/tests.rs`.

Table 6.1: Tests cases related to the lexer along with the expected results.

<table>
<tr><th>Test case</th><th>Result</th><th>Test case</th><th>Result</th><th>Test case</th><th>Result</th></tr>
<tr><td>,</td><td>success</td><td>""</td><td>failure</td><td>abc12␣3␣</td><td>success</td></tr>
<tr><td>|␣/!</td><td>success</td><td>''</td><td>failure</td><td>␣x␣</td><td>success</td></tr>
<tr><td>abc␣=␣b;</td><td>success</td><td>␣?␣test␣?</td><td>success</td><td>␣+␣</td><td>failure</td></tr>
<tr><td>␣(/␣[␣/)␣]</td><td>success</td><td>?a\nbc?␣␣</td><td>success</td><td>␣␣,␣\n,</td><td>success</td></tr>
<tr><td>␣(/)␣</td><td>failure</td><td>␣?bbb␣␣</td><td>failure</td><td>␣(*␣test␣*)␣</td><td>success</td></tr>
<tr><td>␣/</td><td>success</td><td>??</td><td>success</td><td>␣(*␣test␣*␣</td><td>failure</td></tr>
<tr><td>(::)␣{␣}␣</td><td>success</td><td>␣123␣</td><td>success</td><td>␣(*␣(</td><td>failure</td></tr>
<tr><td>␣(:)␣</td><td>failure</td><td>␣1␣2␣␣3␣␣</td><td>success</td><td>,␣(*,␣*)␣,␣</td><td>success</td></tr>
<tr><td>␣"ab␣c␣"␣</td><td>success</td><td>␣01234␣5"</td><td>success</td><td>␣,(*,␣(*␣,*)␣,*)␣,␣,</td><td>success</td></tr>
<tr><td>␣␣'"aba'␣</td><td>success</td><td>␣0␣</td><td>success</td><td>␣(*␣(*␣*)␣</td><td>failure</td></tr>
<tr><td>␣'␣a␣"</td><td>failure</td><td>␣abc␣</td><td>success</td><td>␣(*)␣</td><td>failure</td></tr>
<tr><td>"bbb'␣␣␣</td><td>failure</td><td>a␣␣bc␣</td><td>success</td><td>␣abc␣\n␣=␣'def'␣</td><td>success</td></tr>
</table>

Tests related to the parser test the functionality of the parser, that is, test the ability to transform a list of tokens into an AST. To make the unit tests of the parser independent from the implementation of the lexer, the input to the `parse` function is a vector of tokens, rather than a result of the `lex` function. An example test case related to the parser is seen in listing 6.2, where the `ok_case` macro represents a test case that, after testing the `factor` parser, should result in a success, parse 3 tokens, and return a `Factor` AST node. In the same file, that is `ebnf/src/parser/tests.rs` one can also find the `error_case` macro, which represents a test case that should fail with a specific error.

```
ok_case!(
  factor,
  &vec![
    Token::Integer(2).spanning(Span::from(((0, 0), (1, 0)))),
    Token::Repetition.spanning(Span::from(((2, 0), (3, 0)))),
    Token::Terminal("terminal".to_owned()).spanning(Span::from(((4, 0), (14, 0))))
  ],
  3,
  Expression::Factor {
    count: 2.spanning(Span::from(((0, 0), (1, 0)))),
    primary: Box::new(
      Expression::Terminal("terminal".to_owned()).spanning(Span::from(((4, 0), (14, 0))))
    )
  }
  .spanning(Span::from(((0, 0), (14, 0))))
);
```

Listing 6.2: A unit test related to the parser, where the ability to turn a string of tokens into an AST is tested.

Tests related to the preprocessor test the functionality of the preprocessor, that is, test the ability to detect undefined rules and left recursion. Test case in listing 6.3 tests the grammar

```
a = b;
b = a;
```

and the detection of indirect left recursion $b \longrightarrow a \longrightarrow b$.

```
#[test]
fn test_indirect_left_recursion() {
  assert_eq!(
    preprocess(
      Grammar {
        productions: vec![
          Production {
            lhs: "a".to_owned().spanning(Span::from(((0, 0), (1, 0)))),
            rhs: Expression::Nonterminal("b".to_owned())
              .spanning(Span::from(((4, 0), (5, 0))))
          }
          .spanning(Span::from(((0, 0), (6, 0)))),
          Production {
            lhs: "b".to_owned().spanning(Span::from(((0, 1), (1, 1)))),
            rhs: Expression::Nonterminal("a".to_owned())
              .spanning(Span::from(((4, 1), (5, 1))))
          }
          .spanning(Span::from(((0, 1), (6, 1))))
        ]
      }
      .spanning(Span::from(((0, 0), (6, 1))))
    ),
    Err(
      Error::LeftRecursion(vec!["b".to_owned(), "a".to_owned(), "b".to_owned()])
        .spanning(Span::from(((4, 0), (5, 0))))
    )
  );
}
```

Listing 6.3: A preprocessor unit test testing the left recursion detection in an AST.

All tests related to business logic are written in Rust, as the business logic is also written in Rust. All unit tests in the project can be ran with the use of Cargo with the `cargo test` command in the terminal.

6.1.2. Property-based testing

Test engineers write mostly example-based tests where only one input scenario is tested. Property-based testing is a useful addition to a test suite because it runs one statement hundreds of times with different inputs. Property-based testing frameworks use almost every conceivable input that could break the code, such as empty lists, negative numbers, and really long lists or strings. Property based testing has become quite famous in the functional world. Mainly introduced by QuickCheck framework in Haskell, it suggests another way to test software.

Property-based tests are designed to test the aspects of a property that should always be true. They allow for a range of inputs to be programmed and tested within a single test, rather than having to write a different test for every value that the programmer wants to test. These tests are useful when a range of inputs needs to be tested on a given aspect of a software property, or if the developer is concerned about finding missed edge cases.

```rust
use quickcheck_macros::quickcheck;

#[quickcheck]
fn test_arbitrary_input(input: String) {
  let _ = lex(&input);
}
```

Listing 6.4: A QuickCheck test testing arbitrary inputs on the lexer.

Parser-parser uses the `quickcheck` crate for Rust, which is a property-based testing framework inspired by Haskell's QuickCheck and has been described in section 5.1.1. `quickcheck` tests the "correctness" of `lex` and `parse`, i.e. it checks if these functions do not emit a *panic* for an arbitrary input, whether it's a random string of characters for the case of `lex`, or a random string of tokens for `parse`. Example `quickcheck` tests can be seen in listings 6.4 and 6.5, where the `quickcheck` macro is used to streamline the usage of the library. `quickcheck` catches any panics that may occur in `lex` and `parse` functions and reports them to the user along with the input provided to these functions. In the case of `parse`, to generate an arbitrary string of `Token`s, it was necessary to implement the `Arbitrary` trait provided by `quickcheck` for `Token`. `Spanned<T>`, `Span`, and `Location` types (most of this has been omitted in listing 6.5).

```rust
// ...
impl<T: Arbitrary> Arbitrary for Spanned<T> {
  fn arbitrary<G: Gen>(g: &mut G) -> Spanned<T> {
    Spanned {
      node: T::arbitrary(g),
      span: Span::arbitrary(g),
    }
  }
}

#[quickcheck]
fn test_arbitrary_input(tokens: Vec<Spanned<Token>>) {
    use super::parse;
    let _ = parse(tokens.as_slice());
}
```

Listing 6.5: A QuickCheck test related to the parser, testing strings of arbitrary tokens.

6.2. Integration testing

Integration testing is defined as a type of testing where software modules are integrated logically and tested as a group. A typical software project consists of multiple software modules, coded by different programmers. The purpose of this level of testing is to expose defects in the interaction between these software modules when they are integrated.

Parser-parser utilizes integration testing thanks to *wasm-pack*. This tool allows the developer you build rust-generated WebAssembly packages, as well as test them in a headless web browser via the wasm-pack test command. A tool used to define the integration tests is the wasm-bindgen-test crate — an experimental test harness for Rust programs compiled to wasm using wasm-bindgen and the wasm32-unknown-unknown target. The `wasm-pack test` command wraps the wasm-bindgen-test-runner CLI allowing the developer to run wasm tests in different browsers without needing to install the different webdrivers. An example integration test ran with wasm-bindgen-test can be seen in listing 6.6.

```
use ebnf::parse;
use wasm_bindgen_test::*;

#[wasm_bindgen_test]
fn test_ebnf() {
  assert!(parse(" abc = 'def'; ").is_ok());
  assert!(parse(" (* test *) ").is_err());
  assert!(parse(" (* test *").is_err());
  assert!(parse("a = b;").is_err());
  assert!(parse("a = 'a';;").is_err());
  assert!(parse("a = ;").is_ok());
  assert!(parse("a = a;").is_err());
}
```

Listing 6.6: An integration test ran in a headless browser, which tests various grammars in a textual form.

6.3. Benchmarking

Benchmarks check the performance of the code. Parser-parser's benchmark tool of choice is Criterion.rs — a statistics-driven micro-benchmarking tool. It is a Rust port of Haskell's Criterion library. Criterion.rs benchmarks collect and store statistical information from run to run and can automatically detect performance regressions as well as measuring optimizations. Listing 6.7 shows a Criterion.rs benchmark that parses a sample input grammar multiple times.

```
use criterion::{black_box, criterion_group, criterion_main, Criterion};
use ebnf::parse;

const GRAMMAR: &str = "
expression = term, { ('+' | '-'), term };
term       = factor, { ('*' | '/'), factor };
factor     = constant | variable | '(', expression, ')';
variable   = 'x' | 'y' | 'z';
constant   = digit, { digit };
digit      = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
";

fn criterion_benchmark(c: &mut Criterion) {
  c.bench_function("parse", |b| b.iter(|| parse(black_box(GRAMMAR))));
}

criterion_group!(benches, criterion_benchmark);
criterion_main!(benches);
```

Listing 6.7: A benchmark testing the speed of parsing a sample grammar.

Criterion.rs can generate a number of useful charts and graphs which the developer can check to get a better understanding of the behavior of the benchmark. These charts will be generated with *gnuplot* by default, and the examples below were generated using the gnuplot backend.

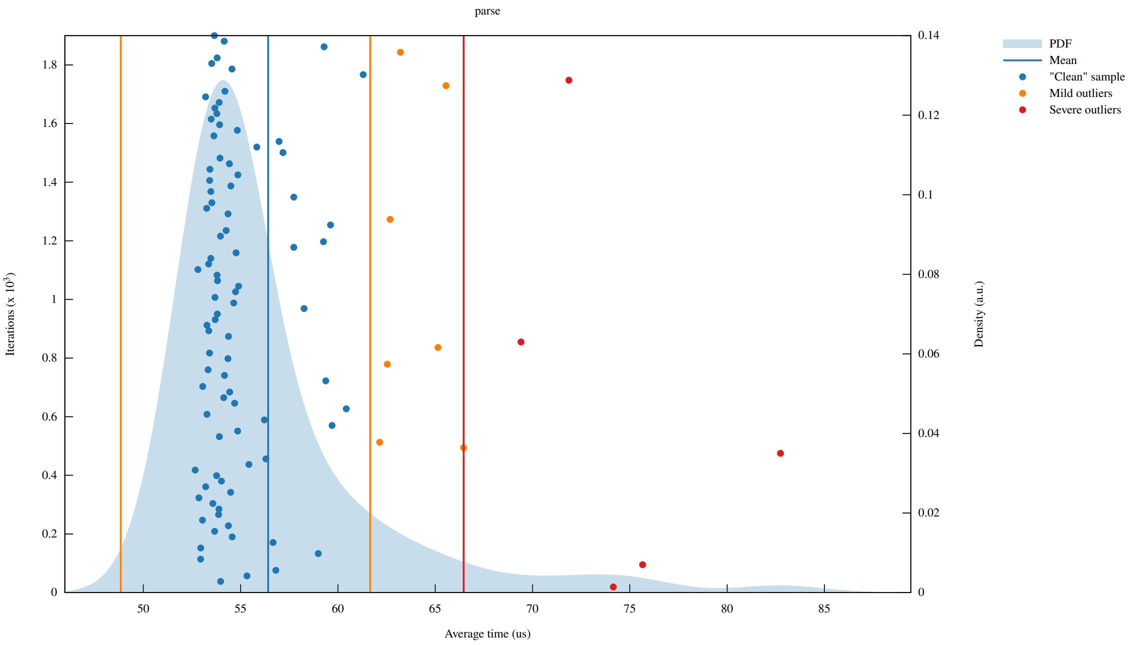

Figure 6.1: Probability Distribution Function chart generated by Criterion.rs.

The *PDF chart* in figure 6.1 shows the *probability distribution function* for the samples. It also shows the ranges used to classify samples as outliers. In this example we can see that the performance trend does not change noticeably across the whole benchmark and stays around 55 µs

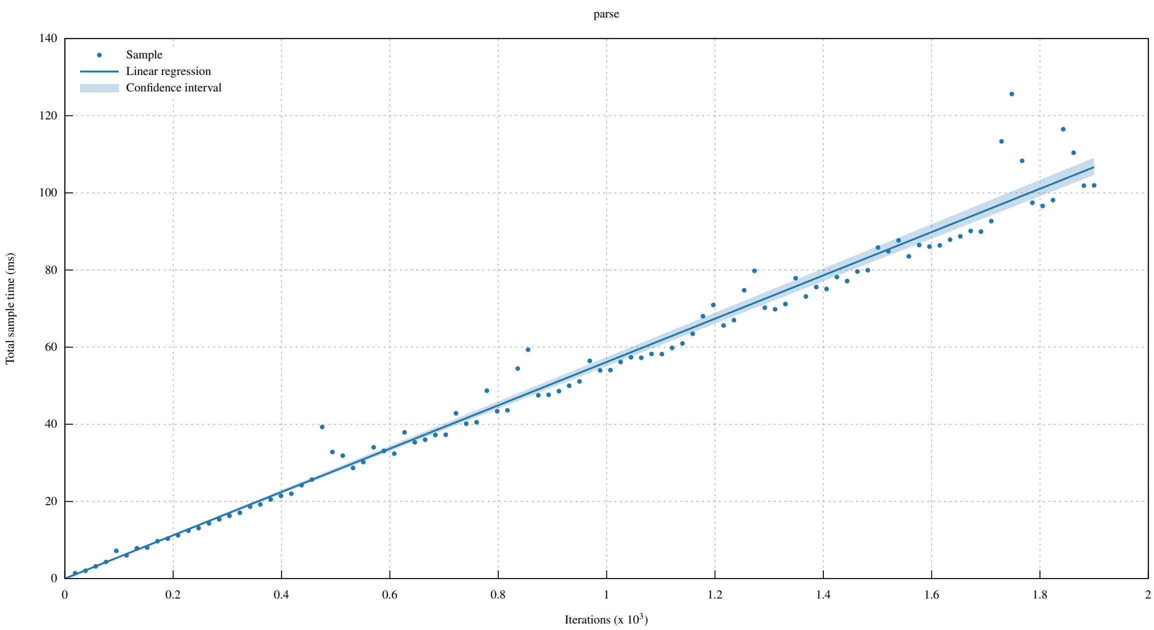

Figure 6.2: Regression plot generated by Criterion.rs.

The *regression plot*, shown in figure 6.2, shows each data point plotted on an X-Y plane showing the number of iterations versus the time taken. It also shows the line representing Criterion.rs' best guess at the time per iteration. A good benchmark will show the data points all closely following the line. If the data points are scattered widely, this indicates that there is a lot of noise in the data and that the benchmark may not be reliable. If the data points follow a consistent trend but don't match the line (eg. if they follow a curved pattern or show several discrete line segments) this indicates that the benchmark is doing different amounts of work depending on the number of iterations, which prevents Criterion.rs from generating accurate statistics and means that the benchmark may need to be reworked.

The graphics that Criterion.rs generates are perfect for contributors of the project as there is no dearth of info. Criterion generates graphics that break down mean, median, standard deviation, MAD, etc., which are invaluable when trying to pinpoint areas of improvement.

6.4. Auditing

A software audit is an internal or external review of a software program to check its quality, progress or adherence to plans, standards and regulations. Parser-parser utilizes Google Lighthouse — an open-source, automated tool for improving the quality of web pages. Developers can run it against any web page, public or requiring authentication. It has audits for performance, accessibility, progressive web apps, SEO and more. When the tool finishes analyzing a web page, it returns a report with the calculated scores for each metric, a list of problems with the page, and general, or sometimes specific, recommendations regarding solving those problems. Figure 6.3 shows a short summary of Lighthouse ran against the deployed web application. For most metrics, Lighthouse calculates a score by comparing the page with the FCP data present in HTTP Archive. The tool uses a color-coding system to display how well a page performs according to a particular metric.

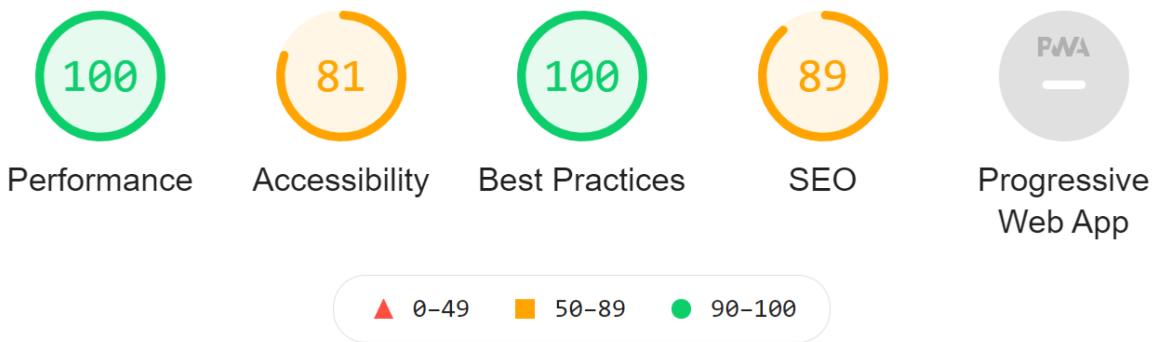

Figure 6.3: Google Lighthouse's scores of the deployed web application.

Parser-parser scores 100 points in the performance category. This category has metrics that together reflect how fast the page is perceptually:

First Contentful Paint Shows how long it takes for a browser to render DOM content,

Speed Index Shows how quickly the contents of a page load visually. To do this, Lighthouse records a video of your page loading and then computes a visual progression between frames,

Largest Contentful Paint Reports the render time of the largest image or text block within the viewport (i.e., the visible part of the page),

Time To Interactive Measures how long it takes a page to become *fully* interactive, i.e. useful content (measured by FCP) is displayed, JavaScript event handlers are bound to visible elements' events, and the page responds to user interaction within 50 milliseconds,

Total Blocking Time The sum of all time records between FCP and TTI when a page is blocked from user interaction for more than 50 milliseconds,

Cumulative Layout Shift Metric for showing how aggressively elements shift each other during the loading.

All values for performance metrics generated by Lighthouse can be seen in table 6.2. Besides the metrics, the website passed additional 29 audits, such as *Minify CSS*, *Minify JavaScript*, *Enable text compression*, *Initial server response time was short*, *Remove duplicate modules in JavaScript bundles*, *Avoids an excessive DOM size* and more.

Table 6.2: Performance metrics generated by Lighthouse.

Metric	Value
First Contentful Paint	0.2 s
Speed Index	0.4 s
Largest Contentful Paint	0.4 s
Time to Interactive	0.4 s
Total Blocking Time	0 ms
Cumulative Layout Shift	0

In terms of best practices, the web application scores 100 points, thanks to HTTPS usage provided by GitHub Pages, safe links to cross-origin destinations, avoiding requesting geolocation and notification permissions on page load, and avoiding front-end JavaScript libraries with known security vulnerabilities.

Besides performance audits, Lighthouse also provides audits regarding accessibility, SEO (Search Engine Optimization), and PWA (Progressive Web App). These audits were not the main focus of the project, however they provide useful information about possible future improvements. Lighthouse informs that background and foreground colors do not have a sufficient contrast ratio and that low-contrast text is difficult or impossible for many users to read, which is a valid concern that should be taken into consideration. The accessibility audit also suggests providing labels for associated form elements, which most likely relates to `textarea` grammar and input string fields. In terms of Progressive Web App audits, Lighthouse suggests using a service worker, which enables the web app to be reliable in unpredictable network conditions and use many Progressive Web App features, such as *offline*, *add to homescreen*, and *push notifications*.

6.5. Complexity analysis

Software complexity is a way to describe a specific set of characteristics of the code. These characteristics all focus on how the code interacts with other pieces of code.

A popular metric for measuring code complexity is *cyclomatic complexity*. Cyclomatic complexity uses a mathematical model to assess methods, producing accurate measurements of the effort required to test them, but inaccurate measurements of the effort required to understand them.

Cognitive complexity, on the other hand, breaks from the practice of using mathematical models to assess software maintainability. It starts from the precedents set by cyclomatic complexity, but uses human judgement to assess how structures should be counted, and to decide what should be added to the model as a whole. As a result, it yields method complexity scores which strike programmers as fairer relative assessments of maintainability than have been available with previous models.

Calculating cognitive complexity in Rust can be achieved with the *rust-clippy* tool. Clippy is a collection of lints to catch common mistakes and improve Rust code. There are over 400 lints included in the tool, and are divided into categories, each with a default lint level. One way to use Clippy is by installing Clippy through `rustup` as a Cargo subcommand. Clippy can then be invoked with the `cargo clippy` command.

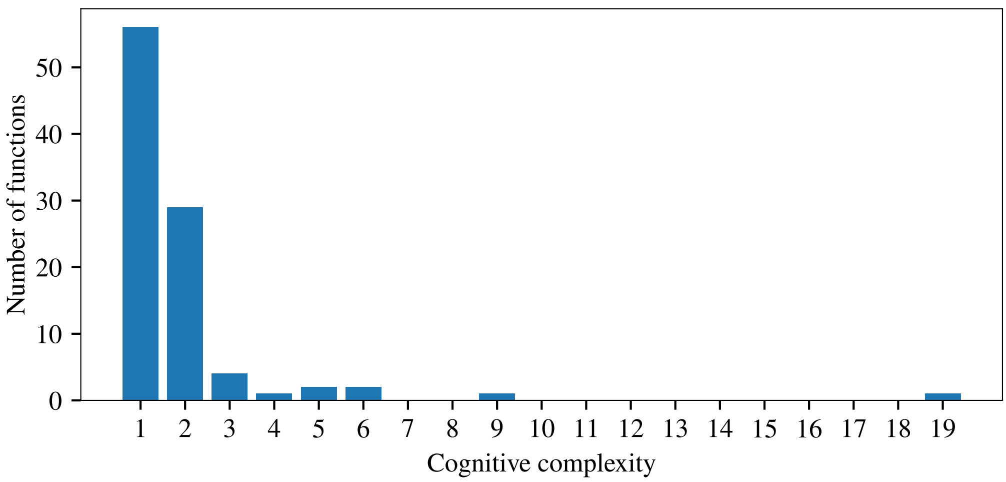

Figure 6.4: Number of functions with various degrees of cognitive complexity.

Figure 6.4 shows the bar plot of a number of functions of a particular cognitive complexity value. Most of functions in the project have a low cognitive complexity of around 1 or 2. Acceptable values for cognitive complexity of functions generally are in the range between 0 and 10. Based on this, there is one function with value above that range — the `lex` function. However, the complexity of the `lex` function is deliberate — the lexer was written with complex behavior in a single procedure in mind, for performance reasons. The written code is simple and low-risk when it comes to maintenance.

7. Deployment

7.1. Application building

Deployment involves packaging up the web application and putting it in a production environment that can run the app. This section is related to the “packaging” process of the application.

[TODO]

7.2. Production environment

Parser-parser uses a **[TODO]**

7.3. Continuous deployment

```yaml
name: CI/CD

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout sources
      uses: actions/checkout@v2
    - name: Set up Node.js
      uses: actions/setup-node@v1
      with:
        node-version: '12'
    - name: Install nightly Rust toolchain
      uses: actions-rs/toolchain@v1
      with:
        profile: minimal
        toolchain: nightly
        override: true
    - name: Install wasm-pack
      uses: jetli/wasm-pack-action@v0.3.0
      with:
        version: 'latest'
    - run: cd web
    - run: npm ci
    - run: npm run build
    - name: Deploy to gh-pages
      run: |
        git config --global user.name $user_name
        git config --global user.email $user_email
        git remote set-url origin https://${github_token}@github.com/${repository}
        npm run deploy
      env:
        user_name: 'github-actions[bot]'
        user_email: 'github-actions[bot]@users.noreply.github.com'
        github_token: ${{ secrets.ACTIONS_DEPLOY_ACCESS_TOKEN }}
        repository: ${{ github.repository }}
```

Listing 7.1: **[TODO]**

[TODO]

8. Artifacts

8.1. Source code

[TODO]

8.2. Web application

[TODO]

8.3. Command-line application

[TODO]

8.4. Documentation

[TODO]

9. User manual

9.1. System requirements

[TODO]

9.2. Installation guide

[TODO]

9.3. Usage guide

[TODO]

10. Summary

[TODO]

Bibliography

[1] Aho, A., Ullman, J. D., Lam, M. S., and Ravi, S. *Kompilatory: reguły, metody, narzędzia*. Wydawnictwo Naukowe PWN, Warsaw, 2019.

[2] Aho, A. V. Algorithms for finding patterns in strings. In *Algorithms and Complexity*. Elsevier, 1990, pp. 255–300.

[3] Chacon, S., and Straub, B. *Pro Git*. Apress, Berkeley, CA New York, NY, 2014.

[4] Chomsky, N. Three models for the description of language. *IEEE Transactions on Information Theory 2*, 3 (Sept. 1956), 113–124.

[5] CodeMirror team. CodeMirror homepage. `https://codemirror.net/`. Retrieved 09.11.2020.

[6] Couprie, G. Nom GitHub page. `https://github.com/Geal/nom`. Retrieved 08.11.2020.

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

[8] Davis, M., and Chapman, C. Unicode standard annex #29. `http://www.unicode.org/reports/tr29/`. Retrieved 08.11.2020.

[9] Dib, F. Regex101 homepage. `https://regex101.com/`. Retrieved 24.10.2020.

[10] Driessen, V. A successful Git branching model. `https://nvie.com/posts/a-successful-git-branching-model/`, 2010. Retrieved 07.11.2020.

[11] Ferrous Systems. rust-analyzer homepage. `https://rust-analyzer.github.io/`. Retrieved 07.11.2020.

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

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

[14] Git community. Git homepage. `https://git-scm.com/`. Retrieved 07.11.2020.

[15] GitHub Inc. GitHub homepage. `https://github.com/`. Retrieved 08.11.2020.

[16] Hopcroft, J., Motwani, R., and Ullman, J. D. *Wprowadzenie do teorii automatów, języków i obliczeń*, second ed. Wydawnictwo Naukowe PWN, Warsaw, 2012.

[17] ISO/IEC. *ISO/IEC 14977:1996(E) — Information technology, syntactic metalanguage, Extended BNF*. Geneva, 1996.

[18] 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.

[19] KLABNIK, S., AND NICHOLS, C. *The Rust programming language*. No Starch Press, Inc, San Francisco, 2018.

[20] LEIJEN, D., AND MEIJER, E. Parsec: Direct style monadic parser combinators for the real world.

[21] MEDUNA, A. *Formal Languages and Computation: Models and Their Applications*. CRC Press, Taylor & Francis Group, Boca Raton, 2014.

[22] MICROSOFT. Visual Studio Code homepage. `https://code.visualstudio.com/`. Retrieved 07.11.2020.

[23] PEST TEAM. Pest homepage. `https://pest.rs/`. Retrieved 14.11.2020.

[24] ROLLUP TEAM. Rollup homepage. `https://rollupjs.org/guide/en/`. Retrieved 09.11.2020.

[25] RUST AND WEBASSEMBLY TEAM. Wasm-bindgen GitHub page. `https://github.com/rustwasm/wasm-bindgen`. Retrieved 08.11.2020.

[26] RUST TEAM. Rust homepage. `https://www.rust-lang.org/`. Retrieved 08.11.2020.

[27] SIPSER, M. *Wprowadzenie do teorii obliczeń*. Wydawnictwa Naukowo-Techniczne, Warsaw, 2020.

[28] STACK OVERFLOW. Developer Survey Results 2018. `https://insights.stackoverflow.com/survey/2018`. Retrieved 07.11.2020.

[29] STACK OVERFLOW. Developer Survey Results 2019. `https://insights.stackoverflow.com/survey/2019`. Retrieved 07.11.2020.

[30] SVELTE TEAM. Svelte homepage. `https://svelte.dev`. Retrieved 08.11.2020.

[31] SVELTE TEAM. Svelte Language Tools GitHub page. `https://github.com/sveltejs/language-tools`. Retrieved 07.11.2020.

[32] SWIERSTRA, S. D. Combinator parsing: A short tutorial. In *Language Engineering and Rigorous Software Development*. Springer Berlin Heidelberg, 2009, pp. 252–300.

[33] UNICODE-RS TEAM. Unicode-segmentation GitHub page. `https://github.com/unicode-rs/unicode-segmentation`. Retrieved 08.11.2020.

[34] WEBASSEMBLY TEAM. WebAssembly homepage. `https://webassembly.org/`. Retrieved 08.11.2020.

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 [17].