FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Parsing of General Expressions and Describing Programming Language Syntax using Expressions
Student:	Aleksandr Levin
Supervisor:	Ing. Štěpán Plachý
Study program:	Informatics
Branch / specialization:	Computer Science 2021
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2025/2026

Instructions

Implement an LL(1) parser for general expressions: Given an alphabet of tokens and a table of operator priorities, construct an abstract syntax tree of an input string.
The priority table should accommodate all necessary properties influencing the syntax of an operator, including:
- Existence of left/right operands (prefix, infix, postfix, circumfix).
- Number of tokens (allowing general arity of an operator).
- Associativity of a level in the table (left, right, none).

This description also includes the characterization of elementary values as single token, non-associative operators without left and right operands.
Agree with the supervisor on the technologies used for implementation.

Additionally, explore the following hypothesis: The entire syntax of a programming language (preprocessed by a lexer) can be described as an expression defined by some table of operator priorities.
After an agreement with the supervisor, choose a common programming language, use an existing or custom lexer, and given its alphabet of tokens, design a priority table of operators such that the parser of general expressions can decide the syntactic correctness of an input source code.

Test all your implementations.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 7 March 2024 in Prague.

Bachelor's thesis

PARSING OF GENERAL EXPRESSIONS AND DESCRIBING PROGRAMMING LANGUAGE SYNTAX USING EXPRESSIONS

Aleksandr Levin

Faculty of Information Technology
Faculty of Computer Science
Supervisor: Ing. Štěpán Plachý
May 20, 2024

Contents

List of Figures

List of code listings

First of all, I want to give thanks to my parents and all those who stood by me during both my brightest days and darkest hours. I also want to express my deepest gratitude to the faculty and academic staff who provided us with an amazing study experience in this marvelous place full of knowledge and academic adventures. Special thanks to my supervisor, Ing. Štěpán Plachý. Despite all the difficulties encountered on our way to completing the thesis, we managed to reach the finish line, and I am very pleased because of it. Lastly, I am grateful to the university administration for their efforts in making our time at the university as comfortable as it could possibly be. Without of their support, I can hardly imagine this university being the great place it is today.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on 20 May 2024

Abstract

This thesis provides an LL(1) parser for general expressions. Given correct operator priority table, the parser is able to parse given input string and, if successful, provide AST representation of the input. The thesis also provides Pascal syntax grammar in the form of operators priority table to discuss and develop the idea of describing programming language syntax using expressions. The thesis is supported by a number of Pascal code examples, processed by the developed parser, used together with the provided syntax table.

Keywords Parser, LL(1) Parser, Pascal, AST, Operators, General Expressions, Grammars, Automata Theory

Abstrakt

Tato práce poskytuje parser LL(1) pro obecné výrazy. Za předpokladu správné tabulky priorit operátorů je analyzátor schopen analyzovat daný vstupní řetězec a v případě úspěchu poskytnout AST reprezentaci vstupu. Práce také poskytuje gramatiku syntaxe Pascalu ve formě tabulky priorit operátorů k diskuzi a rozvoji myšlenky popisu syntaxe programovacího jazyka pomocí výrazů. Práce je podpořena řadou příkladů kódu Pascal, zpracovaných vyvinutým parserem, použitých spolu s poskytnutou syntaxovou tabulkou.

Klíčová slova Parser, LL(1) Parser, Pascal, AST, Operátory, Obecné Výrazy, Gramatiky, Teorie Automatů

List of abbreviations

IR	Intermediate Representation
AST	Abstract Syntax Tree
LL(1)	Left-to-right, Leftmost derivation (with 1 lookahead token)
DFA	Deterministic Finite Automata
NFA	Non-deterministic Finite Automata
PDA	Pushdown Automata
BNF	Backus–Naur Form
PS	Parsed Storage
TP	Table Parser
TM	Turing Machine

Chapter 1

Introduction

Programming brings a lot of benefits to people: computer can automate different tasks, like calculations, manual labour in factories, even the thinking process. However, all the good things come with a cost. The cost in our case is the challenge of translating the way we, people, think into the way that a computer can understand, which means, translating our thoughts into binary. That is exactly where all the compiling and parsing appears. Compiler (just as the parser, which is an integral part of the former one) is a tool that helps us, people, to communicate with machines. It takes the code, written in the human-friendly way with a help of some programming language and translates this code into proper binary format, which is then used by machines.

This whole process of translation is a complex combination of different steps, like creation of IR (Intermediate Representation), checking for syntactic errors, application of some optimisations and finally, the code generation. To simplify things, people have divided the whole compilation process into different stages:

- **Front End**: Given some initial source code, usually preprocessed by so called lexer (also known as tokenizer, a tool that separates the whole code into different tokens), it translates the code into IR, which can have a form of an AST (Abstract Syntax Tree) or some stack-based structure. This IR is then used in the following steps as the simplified and standardized code representation. The IR is usually responsible for removal of all the syntactic sugar and its main purpose is representation of the logical structure of the code, rewritten into some standard form. Also at this stage, syntax of the language is checked, so that any source code that does not match the language syntax is handled properly.
- **Middle End**: During this stage, the result of Front End, the IR is modified. This stage is responsible for all the optimisations of the code and thus is optional - without it the code will still work, but it will have worse performance than the modified one.

- **Back End**: The final part of the compilation and the very purpose of it: final translation of (optionally) optimised IR into the machine-readable representation.

Even though all these stages are interesting to analyze, this thesis covers only one precise part of this great process: parsing. More precisely, LL(1) parsing of general expressions and a theoretical possibility of covering entire programming languages syntax using these expressions.

1.1 Thesis: Objectives and Contents

1.1.1 Objective of the Thesis

The primary objective of this thesis is to develop an LL(1) parser for general expressions. By designing a parser, capable of handling general expressions, it can be used in various ways:

- **Code Translation**: Translating code from programming languages to some IR or straight forward into machine code.
- **Education and Experimentation**: Creating and testing new programming languages, and serving as a demonstrative tool in schools and universities related to compilers and languages.

The secondary objective is exploration of a hypothesis that the entire syntax of a programming language can be defined as an expression, represented by a table of operator priorities. This exploration should be done by creating a priority table of operators that would be able, if used together with an implemented parser, to decide syntactic correctness of an input code.

1.1.2 Contents

- **Chapter 2 (Basic Theory)**: Covers theoretical aspects such as automata theory, grammars, language types, and operators.
- **Chapter 3 (Design)**: Discusses the challenges of the thesis topic, solutions, and the algorithmic structure of the parser.
- **Chapter 4 (Implementation)**: Focuses on the coding details of the parser, explaining the inner implementation and utilized structures.
- **Chapter 5 (Pascal language syntax in the form of general expressions)**: Develops the idea of describing programming language syntax using expressions with the Pascal language taken as an example.

- **Chapter 6 (Testing)**: Describes the testing methods, exceptional cases, and provides further insights into the code structure and algorithmic aspects.
- **Chapter 7 (Technical Details)**: Details the file structure, prerequisites for building and running the parser, and the steps for the build and run sequence.
- **Chapter 8 (User Manual)**: Provides information for users to utilize the parser, including method descriptions, class instances creation, and method invocation interfaces.
- **Chapter 9 (Conclusion)**: Summarizes the work done, concludes the completion of initial tasks, and suggests potential improvements and extensions.

Chapter 2
Basic Theory

This chapter focuses on the theoretical aspect of the thesis and aims to provide all the necessary theoretical background to understand the topic.

2.1 Basic Notations

2.1.1 Alphabet

An **alphabet** is a finite set, whose elements are called **symbols**.[1]

2.1.2 String

A **string** over an alphabet is a finite sequence of symbols from that alphabet. Other notions for strings are **word** and **sentence**. The empty string is denoted by ϵ.[1]

2.1.3 Formal language

A **formal language** over an alphabet is any subset of the set of all strings over the alphabet.[1]

2.2 The Chomsky Hierarchy

In computer science, particularly in formal languages and automata theory, languages are classified by the complexity of their grammatical structure using the **Chomsky hierarchy**. This classification includes several types of formal languages, with the most relevant to compiler design and parsing being regular languages and context-free languages.

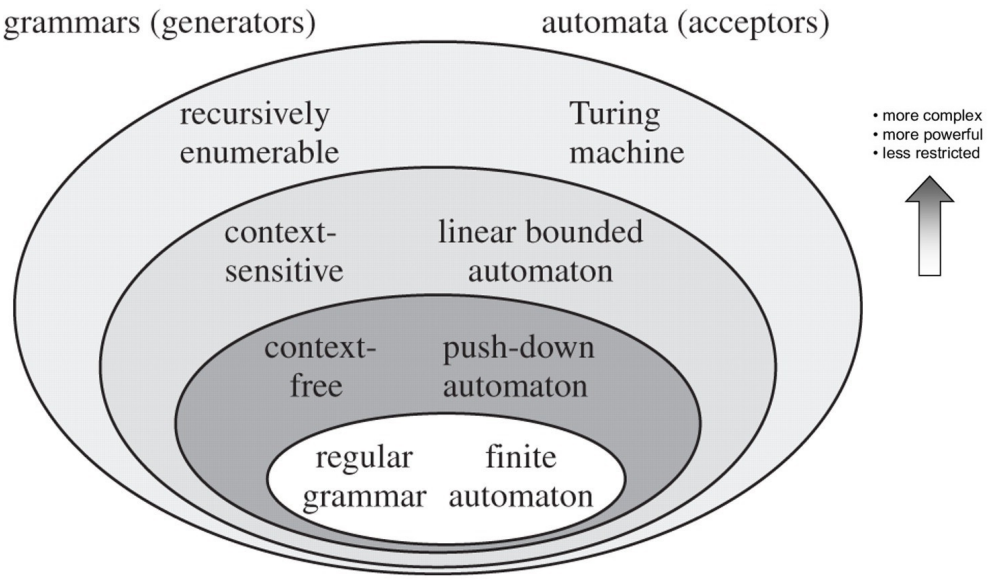

Figure 2.1 Chomsky Hierarchy of languages[2]

2.2.1 Regular Languages

Regular languages are the simplest formal languages, used for basic pattern matching and text scanning tasks. They cannot handle nested structures, such as matched parentheses or recursive function calls, common in programming languages.

2.2.2 Context-Free Languages

Context-free languages are more powerful and expressive than regular languages. They can handle recursive and nested structures, making them suitable for describing the syntax of programming languages.

2.2.3 Context-Sensitive Languages

Context-sensitive languages are more powerful than context-free languages. In these grammars, production rules can depend on the context surrounding nonterminals, allowing for more complex structures.

2.2.4 Recursively Enumerable Languages

Recursively enumerable languages are the most powerful, defined by any computable rule. They are not limited by memory or context, making them impractical for syntax analysis and parsing.

2.2.5 Formal Languages

Another definition for **Formal languages** is a set that unifies all previously mentioned languages and include other languages that do not belong to any of these groups.

2.3 Grammar

[1] At its core, a **grammar** is a tool for describing languages, formally defined as a quadruple $G = (N, E, P, S)$, where

- N is a finite non-empty set of **nonterminal symbols** (auxiliary variables, representing some syntactical parts).
- E is a finite set of **terminal symbols**, representing the alphabet of the language generated by the grammar. Elements of E and elements of N are two disjoint sets.
- P is a finite set of **production rules** (grammar rules).
- S is the **starting nonterminal symbol** of the grammar; each sentence generated by the grammar initially starts from it and then expands to its final form by the grammar rules.

2.3.1 Grammar Rules

Grammar rules define how a grammar describes its language. In general, grammar rules can have various available patterns. For each language type, there is a grammar that generates it. However, the most important type of rules for us right now is the one that defines context-free grammars. The rules of a context-free grammar are defined in the following way:

$$A \to a \tag{2.1}$$

where A is a single nonterminal symbol, and a is a string of terminals and/or nonterminals (which can be empty).[1]

This type of grammar rule (and grammar, consequently) is suitable for defining the syntax of programming languages. One of the main formats for writing these grammar rules is the **Backus–Naur Form** (BNF).

2.4 Automata Theory

Automata theory is a fundamental part of theoretical computer science. It deals with the design and analysis of algorithms and computational processes for abstract machines or automata. These abstract machines are used to recognize patterns, process text, and model computational problems.

In the context of automata theory, to recognize a string or a pattern means to determine, whether it is a part of a language or not.

2.4.1 Finite Automata

Finite automata are the simplest type of automaton. They are used to recognize regular languages. There are two main types of finite automata: deterministic finite automata (DFA) and non-deterministic finite automata (NFA).

2.4.1.1 Deterministic Finite Automata (DFA)

[1] A **DFA** is defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states.
- Σ is a finite set of input symbols (alphabet).
- δ is the transition function $\delta : Q \times \Sigma \to Q$.
- q_0 is the initial state.
- F is a set of accept states.

In a DFA, for each state and input symbol, there is exactly one transition to a new state.

2.4.1.2 Non-deterministic Finite Automata (NFA)

[1] An **NFA** is similar to a DFA but allows for multiple transitions for a given state and input symbol or even transitions without any input symbol (epsilon transitions). It is defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states.
- Σ is a finite set of input symbols (alphabet).
- δ is the transition function $\delta : Q \times \Sigma \to 2^Q$ (power set of Q).
- q_0 is the initial state.
- F is a set of accept states.

NFAs are more flexible than DFAs, but both recognize the same class of languages (regular languages).

2.4.2 Pushdown Automata

[1] Pushdown automata (PDA) are used to recognize context-free languages. A **PDA** is like a finite automaton but with an additional stack storage. It is defined by a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where:

- Q is a finite set of states.
- Σ is a finite set of input symbols (alphabet).
- Γ is a finite set of stack symbols.
- δ is the transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$.
- q_0 is the initial state.
- Z_0 is the initial stack symbol.
- F is a set of accept states.

PDAs can use the stack to handle recursive and nested structures, making them suitable for parsing context-free languages. Also, PDA is a computational model of LL(1) parsers.

2.4.3 Turing Machines

[1] Turing machines are the most powerful type of automaton, capable of recognizing recursively enumerable languages. A **Turing machine** is an abstract model of computation that can simulate any algorithm. It is defined by a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where:

- Q is a finite set of states.
- Σ is a finite set of input symbols (alphabet).
- Γ is a finite set of tape symbols, including the blank symbol.
- δ is the transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$.
- q_0 is the initial state.
- q_{accept} is the accept state.
- q_{reject} is the reject state.

A Turing machine has an infinite tape that it can read from and write to, and a head that can move left or right along the tape. Linear-bounded TM is a model for context-sensetive languages.

2.5 Parsing

Parsing is the process of analyzing a string of symbols (usually, a code of some programming language) according to the rules of a grammar. The goal is to determine if the string can be generated by the grammar rules and what sequence of rules generates it. This process is similar to checking some natural language sentence to see if it makes sense grammatically and structurally. In general terms, parsing is how a computer processes some sentence (code) to ensure it follows the syntax rules of the grammar (programming language).

2.6 Recursive Descent Parsing

Recursive Descent parsing is a top-down parsing technique, meaning it starts at the highest level of the grammar and works its way down to the input, which is then processed from left to right. Also, it is an algorithmic implementation of LL(1) parsing.

2.7 LL(1) Parsing and General Expressions

Parsing, a mathematically-based algorithm, has two main families: LL and LR. The primary difference between these types is the direction of derivation: LL parsers follow the leftmost derivation, while LR parsers follow the reverse of rightmost derivation. Each family has different members, such as LL(0), LL(1) for LL parsers, and LR(0), SLR(1), LALR(1), CLR(1) for LR parsers.

This thesis focuses on LL(1) parsers. The name ”LL(1)” indicates:

- The first ”L” stands for left-to-right input scanning.
- The second ”L” stands for leftmost derivation.
- The number ”1” denotes the lookahead tokens used. The lookahead, combined with the current parsing state, determines the next rule to apply.

For a parser to be LL(1), each combination of state and lookahead must uniquely determine the next action. If there is more than one possible transition, the parser is not LL(1).

General expressions, in this context, are combinations of literals, variables, operators, and function calls that produce a value when evaluated. These expressions are divided into levels, each expanding into an alternating combination of operators and other levels or into a higher priority level, which must be evaluated first.

2.8 Operators in Mathematical Terms

Operators are special symbols in programming that perform operations on one or more operands (the values that operators affect). From a mathematical perspective, operators have several properties that define how they behave:

- **Arity**: This refers to the number of operands an operator acts upon. Unary operators work on one operand, binary operators on two, etc.
- **Associativity**: This property defines how operators of the same precedence are grouped in the case of having no parentheses to specify the order of evaluation. For example, in arithmetic, the addition operator $(+)$ is left-associative, meaning $1+2+3+4+5$ is grouped as $(((1+2)+3)+4)+5$.
- **Precedence**: Some operators act before others unless parentheses change this order. For instance, multiplication in arithmetic has higher precedence than addition.
- **Left/Right**: Left and right denote whether operator has left and right operands respectively. Different combinations define different types of operators: infix, prefix, postfix and circumfix.

2.9 How to Design a Grammar for Expressions

2.9.1 Table Levels

Assuming levels with higher numbers have higher priority, the grammar will contain the following types of rules:

$L_i \to \langle$rules for each operator on i-th level$\rangle \mid L_{i+1}$
$L_{\max} \to \langle$rules for possible values$\rangle$ (and circumfix operators, explained later)

The idea is that in the AST, the root will be an operator with the lowest priority. We follow the recursive descent principle of constructing the AST: either we choose an operator on this level or move up to the next (higher priority) level until we reach the top, which will contain just values as leaves.

2.9.2 Operator Rules

The basic structure of the rule looks like this: (in this notation, "?" means optional and "(...)*" means repeated zero or more times)

$$L_i \to \text{LEFT? } \langle\text{operator token}\rangle(\text{MIDDLE } \langle\text{operator token}\rangle)^* \text{ RIGHT?}$$

LEFT, MIDDLE, and RIGHT will be some non-terminal symbols chosen later based on operator properties. The middle part appears only in the case of multi-token operators (e.g., ternary operator).

For each operator on the i-th level, we add the following rule:

$$L_i \rightarrow \text{LEFT? } \langle\text{op token}\rangle(\text{MIDDLE } \langle\text{op token}\rangle)^* \text{ RIGHT?}$$

where:

- **LEFT**
  - L_i if left associative (left operand mandatory)
  - L_{i+1} if not left associative and has a left operand
  - empty if does not have a left operand (left associativity not allowed)
- **MIDDLE**
  - $L_{\min}$ in all cases. This part is repeated for each of the operator tokens after the first one.
- **RIGHT**
  - L_i if right associative (right operand mandatory)
  - L_{i+1} if not right associative and has a right operand
  - empty if does not have a right operand (right associativity not allowed)

All operators on the same level must have the same associativity. Other properties, in general, do not have to be shared.

The highest priority level must be non-associative and therefore can contain only circumfix operators (which include values as well).

2.10 References for further study

To get better and more detailed understanding of such concepts as parsing, compiler design and Automata theory, you are advised to read the following related books:"Introduction to Automata Theory, Languages, and Computation"[3] and "Compilers: Principles, Techniques, and Tools"[4]

Chapter 3
Design

To begin with, let's recap the main task of this thesis: implementing a general expression parser.

3.1 Initial Input

The alphabet of tokens and a table of operator priorities.

3.1.1 Table Content

The table must contain the following elements:

- Existence of left/right operands (prefix, infix, postfix, circumfix).
- Number of tokens (allowing general arity of an operator).
- Associativity of a level in the table (left, right, none).

3.2 Input

A string that needs to be parsed according to the initially provided table.

3.3 Output

A boolean value indicating the parsing result. In case of success, the AST representation of the input string.

3.4 Main Design Idea

At the core of the solution is the concept of a structure called Parsed Storage (PS). This structure is used to store parsed elements. These elements are either tokens or other Parsed Storages, filled by the Recursive Descent parsing algorithm. They form an alternating sequence: if at the end of the storage is a token, only a PS can be pushed in and vice versa. PS creates an intermediate representation of the currently parsed part of the input string at each moment and each level. This helps navigate the parser and solve problems described further in this chapter. It might be beneficial to treat PS as a dynamic version of an AST, which builds itself gradually until it matches some expression.

3.5 Design Challenges

3.5.1 Left Recursion

Left recursion is a well-known problem for LL parsers. Essentially, left recursion means that there is a grammar rule for some nonterminal A that refers back to itself on the leftmost side of the right-hand side of the rule, either directly or indirectly. In other words, there is a rule that transforms some A into Aa, where a is the suffix of the rule and A is the left-recursive prefix. This situation is dangerous for LL(1) parsers as it may cause infinite recursion, causing the whole parsing process to never terminate.

3.5.1.1 Left Recursion Solution

The issue of left recursion is solved by the runtime transformation of left-recursive rules into right-recursive ones. It works technically the same as transforming a left-recursive rule by the following operation:

$$E \to E + T \mid T$$

becomes

$$E \to TE'$$

and

$$E' \to +TE' \mid \epsilon$$

where ϵ represents the empty element.

3.5.2 Prefix Collision

One of the main problems faced during the development of the solution was the prefix problem. If there are two or more rules with the same prefix, it creates ambiguity in the grammar.

3.5.2.1 Prefix Collision Solution

For LL(1) grammars, this problem is generally solved via the "Left factorization" technique. This method unifies the prefix part of all rules that share one, and then adds one nonterminal after the prefix, which then branches into all initial grammar rules (without the prefix, of course). An example of this factorization is as follows:

$$A \to CB1 \mid CB2$$

transforms into

$$A \to CA'$$

and

$$A' \to B1 \mid B2$$

However, to use this method, some pre-processing must take place, which is unnecessary as the Parsed Storage structure handles this case: during the parsing process, it stores the already parsed part inside. For example, after parsing C from the original first rule, it will be stored inside PS. Even if the whole rule fails, the parsed nonterminal C (the result of parsing it) will still be stored inside the PS, and when the second rule is parsed, the nonterminal C will be simply matched, and then the rest of the rule will be parsed.

3.5.3 Two Rules with One Being the Prefix of Another

Another common parsing problem appears when there are two or more rules, where one rule is the prefix of another. One example of this problem is the dangling else statement: a parsing problem in which an optional else clause in an if-then(-else) statement creates ambiguity in nested conditions.

3.5.3.1 Problem Solution

To solve this problem, some important information must be told beforehand. During the parsing process of a level, all of its operators are being parsed in some order. If rules are represented by some table in the thesis paper, for example like rules describing the syntax of Pascal5.1, they are being parsed from top to bottom. This applies to inner-level parsing as well. This is done intentionally to solve this prefix rule problem.

To make parsing unambiguous in case of encountering such prefix rule problem, we must ensure that, when we are putting rules in the table, that the longest rule will have some preference and will be parsed before its shorter

counterpart. This principle can be shown in action on the dangling else example, which is explained in the section 5. In case of problematic rules having an optional left operand, before parsing the body of a operator, optional left operand must be parsed, so this fact must be taken into account when arranging rules inside a table. To get some examples of how to handle these rules and how parsing algorithm works in general, visit the Testing section6.1 of the thesis.

Chapter 4

Implementation

This chapter will be centered around implementation details with the main focus on Parsed Storage and general structure of all the elements of the code.

4.1 Parsed Storage

Class of PS is defined as a wrapper class over

$std :: vector < std :: variant < ParsedStorage, Token >>$

It also has a special variable, storageIndex, which holds the index of the next element of a storage that we need to match. If storageIndex is equal to the size of the storage, it means that the storage is completely reparsed, so the next time some token or another PS will be parsed, it will be actually inserted and not only matched.

PS as the wrapper adds some rules to pushing back things into it's inner vector: it requires every next pushed element to be of the different type (there are two types, STORAGE and TOKEN in this $std :: variant$), so for example, if current element at the back of the storage has the type of a TOKEN, then only STORAGE type can be pushed, otherwise, any attempt to push another TOKEN will result in an runtime exception. This rule does not apply to the initially pushed element - it can have any type. By forcing this rule, PS makes its inner storage work as an imitation of an universal AST node, which can be partially defined before completing its definition.

There is also the system of packing and unpacking storages. Storage is packed if it contains only one element: another storage. Packed storage means that it's only inner storage is well-defined, so it works as the marker of completing parsing process of some storage - in the end of parsing process, in case of success, the storage is packed. This creates a rule: if there is a storage inside of another storage, it means that the inner substorage is well-defined, so it will not be modified in the future by any means.

4.2 Table Parser

Table Parser (TP) has the following inner deffinition:

```
class TableParser
{
public:
    TableParser( ParsingLayers layers ) : layers(
    ↪  std::move(layers) ) {};

    bool parse ( const std::string& input );

    bool compareStorage( const ParsedStorage& otherStorage );

private:
    ParsingLayers layers;
    ParsedStorage parsedAST;
    Tokenizer tokenizer;
};
```

Code listing 4.1 Inner state of a TP

Its layers must be defined in the process of the initialization of the table, so once initialized, it cannot change it's table (this behaviour can be changed easily if it is requested). Layers correspond to the priority table of parsing.

ParsedAST is an instance of PS. After the parsing process, in case of success, AST representation of the parsed string will be stored there. In case of failure the content of parsedAST in undefined.

Tokenizer is the lexical analyzer of the parser, it is fed new string each time parsing method is invoked.

compareStorage is the method, created mainly for the testing purposes.

4.3 Parsing Layers

Parsing Layers is basically a container for layers, it has a collection of layers and different methods to get and set its inner layers.

4.4 Parsing Layer

Parsing Layer is, in some way, also just a collection of different Atomic Operations, united under the same layer, so it as well has some setting and getting methods for its inner operations. Also, it provides a method to parse a layer:

```
bool ParsingLayer::parse(ParsingLayers &layers, ParsedStorage
↪  &storage,
Tokenizer &tokenizer, size_t layerIndex) const {
    for ( auto op = operators . begin(); op != operators .
    ↪  end(); ) {
        if ( op -> parse( layers, storage, tokenizer, layerIndex
        ↪  ) ) {
            if ( associativity != LEFT && storage . isPacked() )
            ↪  return true;
            op = operators.begin();
        } else
            ++op;
        storage . resetPosition();
    }

    auto nextLayer = layers . getLayer( layerIndex + 1 );
    return storage.isPacked() || ( nextLayer
                && nextLayer->parse(layers, storage, tokenizer,
                ↪  layerIndex + 1) );
}
```

Code listing 4.2 Parse method of a Parsing Layer

The logic of this method is relatively simple: it goes through all operations of a layer and tries to parse them. This operation returns true in two cases, successful parsing or the case when optional left operand returns false, but current PS was empty, so it means that some higher priority operations failed, but still, it might have written something in the storage, so we inherit it and try to parse the whole layer again with the new PS content. In case that no operation succeeded, it means that we still should try to parse one layer lower because of the definition of general expressions.

The if statement in the parsing loop is for successful parsing operations of right recursive or not recursive in general rules.

Also, *storage.isPacked*() here works as marker of a successful parsing, because of the way packing system works for Parsed Storages.

4.5 Atomic Parsing Operations

Atomic Parsing operations are characterized by the following member variables: two boolean flags for existence of left and right optional operands and a vector of tokens. All of the initialization is done in the constructor.

Main feature of Atomic Parsing operations is its ”parse” method as it features, the runtime transformation of left recursive rules into right recursive ones. Also, the logic for inheritance of storages from the failed optional left operand parsing is described there. The method itself and its helping methods are defined in the following way:

```
bool AtomicParsingOp::parse(ParsingLayers &layers, ParsedStorage
↪  &storage,
                            Tokenizer &tokenizer, size_t layerIndex)
                            ↪  const {
    auto upperLayer = layers.getUpperLayer(),
         nextLayer = layers.getLayer(layerIndex + 1);
    //Parse optional left operand
    if ( this -> hasLeftOperand() ) {
        bool storageWasEmpty = storage . empty();
        if ( !nextLayer ) throw std::runtime_error("Wrong expression
        ↪  format:
                                operator has left operand at the last
                                ↪  layer.");
        if ( !parseInnerLayer( layers, storage, tokenizer,
                               layerIndex + 1, nextLayer ) ) {
            if ( storageWasEmpty && !storage . empty() )
                return true;
            else
                return false;
        }
    }
    //Parse necessary first token
    if ( !parseInnerToken( storage, tokenizer, tokens.front() ) )
        return false;
    //Parse middle part
    for (size_t tokenPos = 1; tokenPos < tokens.size(); ++tokenPos) {
        if ( !parseInnerLayer( layers, storage, tokenizer, 0,
        ↪  upperLayer )
          || !parseInnerToken( storage, tokenizer, tokens . at(
          ↪  tokenPos ) ) )
            return false;
    }
    //Parse optional right operand
    if ( this -> hasRightOperand() ) {
        switch ( layers . getLayer( layerIndex ) -> getAssociativity()
        ↪  ) {
            case RIGHT:
                if ( !parseInnerLayer( layers, storage, tokenizer,
                ↪  layerIndex,
                                       layers . getLayer(
                                       ↪  layerIndex ) ) )
                    return false;
                break;
            default:
                if ( !parseInnerLayer( layers, storage, tokenizer,
                ↪  layerIndex + 1, nextLayer ) )
                    return false;
        }
    }
    //Mark completion of the storage
    storage.pack();
    return true;
}
```

Code listing 4.3 Parse method of Atomic Parsing operations

```
bool AtomicParsingOp::parseInnerLayer(ParsingLayers &layers,
            ParsedStorage &storage, Tokenizer &tokenizer,
            size_t layerIndex, const ParsingLayer *layer) {

    ParsedStorage exploringStorage;
    if ( storage . reparsed() ) {
        if ( layer -> parse( layers, exploringStorage, tokenizer,
        ↪  layerIndex ) ) {
            storage . pushBack( exploringStorage . getPackedStorage()
            ↪  );
        } else {
            if ( storage.empty() ) {
                std::swap( storage, exploringStorage );
                return false;
            } else {
                throw std::logic_error("Parsing error while having
                ↪  something in the
                            storage -> two unfinished storages ->
                            ↪  parsing failed.");
            }
        }
    } else {
        return storage . matchSubstorage();
    }

    return true;
}

bool AtomicParsingOp::parseInnerToken(ParsedStorage &storage,
                            Tokenizer &tokenizer, Token token) {
    if ( storage . reparsed() ) {
        if ( !tokenizer . tokenMatch( token ) )
            return false;
        storage . pushBack( token );
    } else {
        return storage . match( token );
    }

    return true;
}
```

Code listing 4.4 Helper methods of parse method

Chapter 5

Pascal Language Syntax in the Form of General Expressions

This chapter focuses on the second objective of this thesis: possibility of describing a programming language using general expressions. The best way of exploring this topic is via creating a grammar for some sample programming language. In our case, Pascal was chosen to be a language: it has simple enough syntactic structure and it also provides official grammar that describes it's syntax.

Priority table5.1, used in this thesis, was created based on the following book: "Pascal User Manual and Report"[5]. The original grammar with a complete description of language features is available via the link, provided in the Bibliography.

Also note that there exists a "dangling else" problem in the table. However, the parsing process is still unambiguous and the else statement is guaranteed to belong to the closest if statement. This is achieved by correct placement of two rules for the if statement and the fact, that when parsing a level, operators of a rule are parsed from top to bottom, hence if-then-else rule is parsed before if-then rule and so if there is an "else" token, if-then-else rule will be chosen, while if there are no "else" statement, if-then-else rule will fail and the next one on the queue will be if-then statement, which at that time will already be completed in a PS, so it will be simply matched.

5.1 Arising Expression's Issues

This chapter asks one important question: is it possible to describe a syntax of programming language utilizing only expressions? The simplest answer to this question is no. The main reason is the lack of control over rule definitions. We simply do not have enough control to restrict a language the way we want. The source of this issue is the way general expressions are defined: we can either stay on the same level in case of some rule being recursive, go down one level (either as a part of some rule or as the default case when a nonterminal goes to a nonterminal one level lower), or go to the lowest priority level when parsing something in between two tokens. Staying on the same level creates no issues, while going one layer down can be somewhat handled by clever rule rearranging. However, there still might be some unsolvable situations, like "to" and "downto" operators in the Pascal grammar5.1: we should not be able to compare them and at the same time there should never appear any comparison as their operands, so if these rules are the way they are in the grammar, they produce invalid statements, when used in comparison operators, while if they are moved up the table, there is a chance of getting comparison as operands, which also results in an invalid language element. But that's not the only problem.

The biggest problem arises when we go to the lowest priority case, as the thing between two tokens can be completely anything since it starts from the lowest priority. For example, it creates a problem if we have some unique part of the language which should not be repeated more than once. We should somehow define it by creating a sequence of rules of some levels that defines it. If we create these rules and insert them in the operators' priority table, it means that these rules will be accessible at least from the lowest priority level. Consequently, any rule with more than one token will be able to reproduce this "unique" part inside of it. This issue can be illustrated with an example from the Pascal language:

```
assert( parser.parse("program ImperfectionTest;\n"
                     "begin\n"
                     "  program HelloWorld;\n"
                     "  begin\n"
                     "      writeln('Hello, world!')\n"
                     "  end\n"
                     "end")
    );
```

Code listing 5.1 Expressions imperfection test

This code snippet should not be part of a language because of the repeated program definition. However, due to the *begin* – *end* rule at the level 45.1,

there can be anything in between ”begin” and ”end,” which creates this invalid statement.

For these reasons, there is no way to restrict anything in between two tokens by pure syntax. Thus, it is impossible to describe the entire syntax of a programming language as an expression.

However, there are some ways to deal with these issues.

5.2 Possible solutions

The issues described above exist because there is no way to apply any restrictions on the inner nonterminals of a rule. However, this limitation applies to purely syntactic solutions. From the perspective of semantics, there is a way to restrict unwanted strings from appearing in the language. If we take any rule from any context-free grammar, each nonterminal transforms into some combination of terminals and nonterminals. In general expressions, there is no way to restrict any nonterminal except the leftmost and rightmost ones if they exist (they can be somewhat restricted by the level as they cannot transform into something of lower precedence than the level they are currently on). In ordinary grammar, each nonterminal is restricted by the type of nonterminal that it transforms into. For example:

$$A \to BC$$

Here, nonterminal A goes to two other nonterminals, the first of which has the type B and the second one has the type C. These nonterminals can be anything, but they must satisfy their type. The same approach can be applied to general expressions and their operators' priority table. The best way to restrict unnecessary rule transitions is by applying some expressions type checking.

Given example from above5.1, this situation is possible because of $begin - end$ rule, specifically, the lowest priority element that appears in between two given tokens. This element can be any string from the language, which can create a bunch of unwanted elements in the language. However, if the element in between the tokens was restricted to be some special type, for example, ”statement” type, it would not allow any other type to appear there, so the double definition of a program in that case could not appear as program definition would have some other type, located higher in the table.

Same applies for ”to” and ”downto” examples. There could be some type restriction that would allow comparison operators to have only some ”value” types as their operands and that would exclude two problematic operators from the list of possible operands.

5.3 Summing Things Up

To sum things up, it is not possible to implement the precise syntax of a programming language using only general expressions due to the lack of control over grammar rule transitions. However, syntactic rules can define some super-set of a language, that would include some additional invalid statements, removal of which then could be done by imposing restrictions based on the type of expressions that each rule takes and produces. Also, some more restrictions could be applied via variable scoping system, but this would violate the principles of context-free parsing, as variable scopes can only be handled using context-sensitive grammars.

Operators	Associativity	Priority Level	Commentary
program __	NONE	0	
__ , __ __ ; __ __ : __	RIGHT	1	
label __ *const* __ *type* __ *var* __ *procedure* __ *function* __	NONE	2	
with __ *do* __ *for* __ *do* __ *repeat* __ *until* __ *while* __ *do* __ *if* __ *then* __ *else* __ *if* __ *then* __ *goto* __	RIGHT	3	
begin __ *end* *case* __ of end *case* __ *of* __ *end*	NONE	4	
__ := __	RIGHT	5	
__ <> __ __ = __ __ *in* __ __ < __ __ > __ __ <= __ __ >= __	LEFT	6	
__ *to* __ __ *downto* __	NONE	7	
__ + __ __ − __ __ *or* __	LEFT	8	
__ * __ __ *div* __ __ *mod* __ __ *and* __	LEFT	9	
packed __ *array* [__] *of* __ *record* __ *end* set of __ file of __	RIGHT	10	structure declaration
__ .. __	NONE	11	interval
− __ *not* __ + __ ^__	RIGHT	12	pointer dereferencing
__ ^	NONE	13	pointer declaration
__ . __	LEFT	14	attribute access
__ (__) __ [__]	LEFT	15	function call array access
integer value real value identifier (__) string value *nil* [ ] [__]	NONE	16	empty set set

Table 5.1 Pascal parsing layers and their operators (top-down parsing)

Chapter 6
Testing

This chapter focuses on the testing part of the task. There will be unit tests for the Parsed Storage (PS) structure and system tests for the Table Parser (TP) class with one predefined priority table. The current implementation includes only basic format checks for the table, such as ensuring the highest priority operations do not have any right or left optional operands (to prevent access out of bounds of the layers collection).

6.1 Tested Aspects of PS

- Initial state check (checks the status of PS in the initial state)
- Basic content matching (checks matching methods to work correctly with small input)
- Attempt to break the format by pushing two subsequent Parsed Storages (results in an exception)
- Content matching after the attempt to break the format (checks that after throwing an error content remains unchanged)
- Token mismatch inside a storage (checks the case when a storage token is mismatched)
- Substorage mismatch (same as above, but with a PS)
- Packing and unpacking (create a copy of a storage, pack the original PS and then check its content to match the copy)
- Safety check for memory address copying (checks that during assignment no address data is copied)
- Another attempt to break the format, now by inserting two subsequent tokens (results in an exception, state of PS is unchanged)

Operators	Associativity	Priority Level
_ ? _ : _	RIGHT	1
_ ? _ ! _	RIGHT	1
<> _ >=	RIGHT	2
<> _	RIGHT	2
/ _	LEFT	3
/ _ =	LEFT	3
? _ >	RIGHT	4
_ := _	RIGHT	4
_ + _	LEFT	5
_ - _	LEFT	5
_ * _	LEFT	6
_ / _	LEFT	6
_ ? _ ; _	RIGHT	7
? _ <	RIGHT	7
a	NONE	8
b	NONE	8
c	NONE	8
(_)	NONE	8

Table 6.1 Parsing layers and their operators (top-down parsing)

6.2 Tested Aspects of Table Parser

- Basic functionality check (parse `"a + b"`, check the return value and inner AST structure)
- Left recursive rules check (parse `"a + b + c"`, check the state of the AST to correspond to the left recursive definition)
- Right recursive rules check (parse `"a := b := c"`, check the state of AST to correspond to the right recursive definition)
- Layers combination check (parse `"a := a + b"`, check the basic layers interactions and correct order of evaluation defined by the priorities)
- Storage coming from below check (parse `"a ? b : a + b"`, details in the next section)
- Storage coming from above check (parse `"? a <"`, details in the next section)
- Storage transition on the same layer check (parse `"a ? b ! a + b"`, check that in case of failure of one rule on a layer, PS moves on to the next one correctly)

- Rules, one of which is the prefix of another: short one is in the table second (correct version check, parse `"<> a >="`, checks dangling else-like situation, rules in the table are placed correctly, so parsing is done correctly)
- Rules, one of which is the prefix of another: short one is in the table first (incorrect version check, parse `"/ a ="`, checks dangling else-like situation, rules in the table are placed incorrectly, breaking the defined rules for such cases, so parsing fails as it matches the prefix rule and returns true, without checking for the next rule, which is the correct one)
- Invalid sentence check (parse `"a ?  b :  :  a + b"`, get token mismatch)
- Invalid sentence check (parse `"a + a a"`, check for cases where some first part of the string is correctly created according to the table rules, while some part of the string is left unparsed)

6.3 Tested Aspects of the Pascal grammar

The grammar for the Pascal language can be found at the following index:5.1. The parser used in the tests is the original TP.

- Hello world test:

```
assert( parser.parse("program HelloWorld;\n"
                     "begin\n"
                     "  writeln('Hello, world!')\n"
                     "end")
    );
```

Code listing 6.1 HelloWorld test

- Record test:

```
assert( parser.parse("program RecordExample;\n"
                     "type\n"
                     "  Person = record\n"
                     "    name: string[50];\n"
                     "    age: integer\n"
                     "  end;\n"
                     "\n"
                     "var\n"
                     "  p: Person;\n"
                     "\n"
                     "begin\n"
                     "  p.name := 'John Doe';\n"
                     "  p.age := 30;\n"
                     "  writeln('Name: ', p.name);\n"
                     "  writeln('Age: ', p.age)\n"
                     "end")
    );
```

Code listing 6.2 Record test

- Dangling else test (with else):

```
assert( parser.parse("program ConditionalExample;\n"
                     "var\n"
                     "  a, b: integer;\n"
                     "begin\n"
                     "  a := 15;\n"
                     "  b := 20;\n"
                     "  if a > b then\n"
                     "    writeln('a is greater than b')\n"
                     "  else\n"
                     "    writeln('a is not greater than b')\n"
                     "end")
    );
```

Code listing 6.3 Dangling else test (with else)

- Dangling else test (without else):

```
assert( parser.parse("program ConditionalExample;\n"
                     "var\n"
                     "  a, b: integer;\n"
                     "begin\n"
                     "  a := 15;\n"
                     "  b := 20;\n"
                     "  if a > b then\n"
                     "    writeln('a is greater than b')\n"
                     "end")
    );
```

Code listing 6.4 Dangling else test (without else)

- Imperfection test (this should not be possible, but it is because of the way general expressions work → to fix this behaviour some expression type checking must be implemented):

```
assert( parser.parse("program ImperfectionTest;\n"
                     "begin\n"
                     "  program HelloWorld;\n"
                     "  begin\n"
                     "      writeln('Hello, world!')\n"
                     "  end\n"
                     "end")
    );
```

Code listing 6.5 Imperfection test

- Dangling else test (nested if case):

```
assert( parser.parse("program ConditionalExample;\n"
                     "begin\n"
                     "  if a > b then\n"
                     "    if a = b then\n"
                     "          a\n"
                     "        else\n"
                     "           b\n"
                     "end")
    );
```

Code listing 6.6 Dangling else test (nested if case)

- Token mismatch test:

```
assert( !parser.parse("program HelloWorld;\n"
                      "not begin\n"
                      "  writeln('Hello, world!')\n"
                      "end")
    );
```

Code listing 6.7 Token mismatch test

- Rules, one of which is the prefix of another: short one is in the table second (correct version check, parse `"<> a >="`, checks dangling else-like situation, rules in the table are placed correctly, so parsing is done correctly)
- Rules, one of which is the prefix of another: short one is in the table first (incorrect version check, parse `"/ a ="`, checks dangling else-like situation, rules in the table are placed incorrectly, breaking the defined rules for such cases, so parsing fails as it matches the prefix rule and returns true, without checking for the next rule, which is the correct one)
- Invalid sentence check (parse `"a ?  b :  :  a + b"`, get token mismatch)
- Invalid sentence check (parse `"a + a a"`, check for cases where some first part of the string is correctly created according to the table rules, while some part of the string is left unparsed)

6.4 Important Tests

This section focuses on two test cases and their significance in terms of the algorithm behind the parsing process. These cases are:

`"a ? b : a + b"` and `"? a <"`

According to the table structure, the first rule behaves as follows: Initially, the first rule of the table will try to parse its optional left operand. To do so, it will go down the table and, during that process, will try to parse the first rule of layer under index 6. It will eventually fail, but it will obtain a PS with some `storage(a)`, token question mark, and another `storage(b)` sequence in it. Then, it will propagate this storage upwards until it reaches the first rule, which will match the storage elements and complete the parsing process (by packing itself). This demonstrates the concept of storage coming from below.

The second test mirrors the first one. Here, due to the absence of optional left operands for all potentially needed rules, it will start by matching tokens. It begins with the lower priority rule at layer 3, fails, and then propagates the storage down by trying to parse rules of lower layers (refer to the code listing 4.2, especially the return statement). This illustrates the concept of PS coming from above.

These test cases are not special in terms of implementation handling. Although they inspired the creation of the PS structure, they are highlighted here to demonstrate the inner logic of the code, the idea of PS, and how it is used in the code.

6.5 How to Perform Testing: From Constructing to Testing

6.5.1 PS Testing

To create tests, we must first identify the possible testable aspects. In the case of PS, it is straightforward: you can either test its inner content using the equality operator or perform structural checks, which are more superficial. Essentially, you either check the structure of the upper layer of the storage using `match` and `matchSubstorage` methods, or you create an `expectedStorage` with the elements you expect to see in the testing storage and then compare them using the equality operator. Constructing a parsed storage is simple: when initialized by default, it is empty, so you use the `pushBack` method (which has overloads for both storages and tokens) to get the desired storage and then perform the tests.

6.5.2 Table Parser Testing

For testing an instance of TP, the basic steps are the same: initialize it by providing some layers, then try to parse some strings.

As with PS, for TP, you can check two things: the parsing result (the return value of the `parse` method) or the content of the AST after parsing. Checking the parsing result is straightforward as it is a return value, while

checking the AST content is more complex. To check the AST in the current implementation, the only way is through the `compareStorage` method. It works the same way as the comparison operator for PS: it compares the content of the storage inside a parser with some other storage provided as an argument. Important: do not forget to pack the storage in the argument, as the storage inside the parser will be packed if the `parse` method returns true.

To construct the TP instance, you can use the following code as an example. Additionally, there is a sample for AST comparison below.

```
TableParser parser(
            ParsingLayers()
            .addLayer(
                    ParsingLayer( LEFT )
                    .addOperator( AtomicParsingOp( { tok_plus }, true,
                    ↪  true ) )
                    .addOperator( AtomicParsingOp( { tok_minus },
                    ↪  true, true ) )
            )
            .addLayer(
                    ParsingLayer( NONE )
                    .addOperator( AtomicParsingOp( { tok_a }, false,
                    ↪  false ) )
                    .addOperator( AtomicParsingOp( { tok_b }, false,
                    ↪  false ) )
                    .addOperator( AtomicParsingOp( { tok_c }, false,
                    ↪  false ) )
                    .addOperator( AtomicParsingOp( { tok_op_par,
                    ↪  tok_cl_par }
                                                          , false,
                                                          ↪  false
                                                          ↪  ) )
            )
    );
```

Code listing 6.8 TP construction example

```
auto sumAB = ParsedStorage()
            .pushBack( ParsedStorage().pushBack( tok_a ) )
            .pushBack( tok_plus )
            .pushBack( ParsedStorage().pushBack( tok_b ) ).pack();

assert( parser.parse( "a + b + c" ) );
assert( parser.compareStorage(
            ParsedStorage()
                    .pushBack( sumAB.getPackedStorage() )
                    .pushBack( tok_plus )
                    .pushBack( ParsedStorage().pushBack( tok_c )
                    ↪  ).pack()
            )
    );
```

Code listing 6.9 AST comparison example

Chapter 7

Technical Details

7.1 Requirements to Build and Run

The code does not utilize any external libraries, nor it requires any modern C++ features, so the only requirement is a gcc compiler that supports C++ standard of 20.

7.2 Content of CMakeFile.txt

```
cmake_minimum_required(VERSION 3.27)
project(Bachelor_Thesis)

set(CMAKE_CXX_STANDARD 20)

add_executable(Bachelor_Thesis main.cpp
        ParsedStorage/ParsedStorage.cpp
        ParsedStorage/ParsedStorage.h
        TableParser/TableParser.cpp
        TableParser/TableParser.h
        ParsingLayers/ParsingLayer/ParsingLayer.cpp
        ParsingLayers/ParsingLayer/ParsingLayer.h
        AtomicParsingOp/AtomicParsingOp.cpp
        AtomicParsingOp/AtomicParsingOp.h
        Tokenizer/Tokenizer.cpp
        Tokenizer/Tokenizer.h
        ParsingLayers/ParsingLayers.cpp
        ParsingLayers/ParsingLayers.h)
```

Code listing 7.1 CMakeFile.txt content

Chapter 8

User Manual

This chapter is aimed to provide enough knowledge about TP class for a general person, not involved in the process of creation of this project, to be able to use this class for their own purposes.

8.1 Methods' Description

8.1.1 Constructor

To construct a TP instance one needs to provide the single required parameter - Parsing Layers instance that will define the priority table for the parser. Warning, this layers instance is not processed at the moment of creation, neither it is checked to obey all the necessary rules of the defined priority table format (more precisely, there is no check of positioning of prefix rules, so the whole table can be ambiguous, if the prefix case is not handled correctly: to learn more about this issue, go to 3.5.3).

8.1.2 Parse Method

Method `parse` is used to parse the input string parameter. All the rules in the process of parsing are taken from initial priority table, which is provided in the constructor.

8.1.3 Compare Storage Method

`CompareStorage` method compares the inner storage of a parser with some other storage, which is provided as an argument to the method call.

8.2 Instances Creation

TP instance is created via constructor call, no more methods are needed, after constructor call, the parser is complete. To see the example of how to initialize TP instances, visit Code listing 6.8.

8.3 Methods' Pre Requirements

All the methods of a parser have no special pre requirements, except for `compareStorage` method, which should be called after some `parse` method call. Violation of this rule will not result in any kind of error or an exception, it will result in a plain comparison of some storage with an empty inner storage of a perser as initially the storage is initialized by the default constructor and thus is empty.

Chapter 9

Conclusion

The primary objective of the thesis is complete, LL(1) parser for general expressions is implemented and all the challenges that appeared in the process are solved. Also, secondary objective, the idea of creating a programming language parser, based exclusively on general expressions, is developed as well. The result of research into this field showed that it is not possible to create an expression that would precisely describe a programming language syntax, as the purely syntactic solution does not impose enough restrictions on the grammar to generate the required language. Nevertheless, expression can generate super-set of the language, which then can be further restricted by applying some expressions type checking, along with some variable scopes system. The latter violates the principles of context-free languages and thus makes the language context-sensitive.

Appendix A

Context of attached media

A.1 Files Structure

```
Bachelor Thesis ..... Root directory
├── AtomicParsingOp
├── ParsedStorage
├── ParsingLayers
│   └── ParsingLayer
├── TableParser
└── Tokenizer
```

Bibliography

1. ŠESTÁKOVÁ, Eliška. *Automata and Grammars*. 1st. Prague: Czech Technical University in Prague, 2020.
2. DEVOPEDIA. *Chomsky Hierarchy*. 2023. Available also from: `https://devopedia.org/chomsky-hierarchy`. Accessed: 2024-05-20.
3. AHO, Alfred V.; MOTWANI, Rajeev; ULLMAN, Jeffrey D. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Addison-Wesley, 2006.
4. AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Addison-Wesley, 2010.
5. JENSEN, Kathleen; WIRTH, Niklaus. *Pascal User Manual and Report*. 2nd ed. New York: Springer-Verlag, 1978. ISBN 0387901442. Available also from: `https://archive.org/details/h42_Pascal_User_Manual_and_Report_Second_Edition/mode/1up?view=theater`. Special printing for Apple Computer.