UNIVERSITY OF PLOVDIV "PAISII HILENDARSKI"
FACULTY OF MATHEMATICS AND INFORMATICS
DEPARTMENT OF COMPUTER INFORMATICS

Nikolay Georgiev Handzhiyski

An iterative parsing algorithm with application in the profiling of parsers

ABSTRACT

of a dissertation
for awarding the educational and scientific degree "Doctor"

Field of higher education:
4. Natural sciences, mathematics and informatics
Professional direction:
4.6. Informatics and computer science
Doctoral program: "Informatics"

scientific supervisor
Prof. Elena Somova, PhD

Plovdiv
2024

The dissertation work was discussed and directed for defense before a scientific jury, at a meeting of the "Computer Informatics" department at the Faculty of Mathematics and Informatics of the University of Plovdiv "Paisii Hilendarski", on 02/28/2024.

The dissertation „An iterative parsing algorithm with application in the profiling of parsers" contains 163 pages. The list of used literature includes 191 sources. The list of the author's publications on the topic consists of 6 titles.

The materials for the defense are available in the Dean's Office of the Faculty of Mathematics and Informatics, New Building of PU "Paisii Hilendarski", office 330 every working day from 8:30 am to 5:00 pm.

Author: Nikolay Georgiev Handzhiyski

Title: An iterative parsing algorithm with application in the profiling of parsers

Table of Contents

1 Overview....6
1.1 Formal means....7
1.2 Translators....8
1.3 Conclusions....8
2 Parsing machine....9
2.1 Basic concept of the machine....9
2.2 Characters and lexemes in the machine....9
2.3 Formal definition of a parsing machine....10
2.4 Tokens....10
2.5 Advanced level 1 grammars....11
2.6 Advanced level 2 grammars....11
2.6.1 Well-defined advanced grammars....12
2.6.2 Sets of symbols in advanced grammars....12
2.6.3 Generation of strings....12
2.6.4 Sets of tokens....13
2.6.5 Metagrammar for advanced grammars....14
2.7 Phrase state machine....14
2.8 Modules of a parsing machine....15
2.9 Syntax trees....16
2.10 Conclusions....17
3 Tunnel parsing algorithm....18
3.1 Basic concept of the algorithm....18
3.2 Stacks....18
3.3 Automata....19
3.3.1 Finding of the shortest paths....19
3.3.2 Reachable advanced symbols....19
3.3.3 Conflicts....20
3.3.4 Trees with empty nodes....21
3.4 Tunnels....21
3.5 Routers....22
3.6 Control objects....23
3.7 Parsing....26
3.8 Properties of the Tunnel parsing....26
3.9 Conclusions....26
4 Profiling of automatically generated parsers....26
4.1 Tokens....27
4.2 Template grammar language....27
4.3 Parser generator profiler....27
4.4 Visualizer....28
4.5 Experiments....29
4.6 Conclusions....30
Conclusion....30
References....32

List of the used abbreviations

Abbreviation	Full form	Page/Source
AABNF	Advanced Augmented Backus–Naur Form	14/Dissertation work
ABNF	Augmented Backus–Naur Form	7/[5]
TP	Tunnel Parsing	18/Dissertation work

Introduction

The mathematical basis of the translators, and therefore of the programming languages, is the theory of the formal languages and the abstract machines that process them. Since every programming language is a formal language, it is convenient and common to describe the programming languages or parts of them (vocabulary, syntax, semantics, etc.) by generative grammars, and parts of the translation process itself (lexical, syntactic, and semantic analyses) to be described with different classes of abstract recognizers (finite automata, push-down automata, Turing machines, etc.).

This dissertation is devoted to two important elements of any translator, namely lexical and syntactic analyses. Throughout the dissertation, unless otherwise stated, "data" will mean data processed by a translator.

Checking whether particular data belongs to a given language is a subject of lexical and syntactic analyses, and it is called recognition. The parsing (syntactic analysis) of the specific data is their recognition and the derivation of concrete structural information about them.

The development of algorithms related to the recognition and parsing of data is done by many authors, as the similarities and differences between the algorithms are not always immediately visible. The known algorithms for data recognition and parsing have various advantages and disadvantages.

There are computer programs that automatically generate the source code of a parser based on a given grammar, which code is written in a particular programming language. This code can be used to parse arbitrary data and output the corresponding structure. The derived data structures can be used for both further compilation and analysis.

The dissertation work involves the design of a machine for parsing data, called a parsing machine, in which different parsing algorithms can be used. The advantages of this parsing machine are that it unites together different parsing approaches/algorithms and adds some capabilities missing in the concrete parsers known in the literature.

An important part of the dissertation is the design of a new parsing algorithm called Tunnel Parsing (TP), which is built into the parser module of the parsing machine and takes advantage of the added capabilities of the parsing machine.

Part of this dissertation is also aimed at the study of a class of ambiguous grammars, a proper subclass of the ambiguous context-free grammars, generating data, for which the above-mentioned Tunnel parsing is performed in linear time.

Another essential element of the work is the development of a tool to measure the resources used during the recognition/parsing of data (generated by different context-free grammars) by automatically generated parsers with different parser generators and compilers, called parser generator profiler, profiler for short.

Goal and tasks of the dissertation

The main goal of the dissertation research is to investigate, propose, design, develop, experimentally apply, and approbate means (machines, algorithms, models, languages, and tools) that are suitable for linear data translation based on some ambiguous context-free grammars.

To achieve the set goal of the dissertation research, the following six main tasks were planned:

Task 1. Research of theories, formal means, approaches, methods, algorithms, models, architectures, machines, and systems that are related to the translation of data;

Task 2. Designing a common parsing machine architecture that combines different approaches/algorithms for lexical and syntactic analyses with some new added capabilities;

Task 3. Defining a new kind of context-free grammar, equally powerful as the context-free grammars, conforming to the proposed common parsing machine architecture;

Task 4. Designing a new parsing algorithm embedded into the parsing machine, which, using the added capabilities in it, can parse data based on the new kind of grammars when there is no left recursion in them;

Task 5. Designing and implementing a prototype of a tool for measuring and comparing the resources used by different parsers, including automatic creation of grammars written in a specially created metaprogramming language, as well as creating parsers for these grammars with various parser generators and compilers;

Task 6. Conducting experiments using the created tool.

Structure of the dissertation

The dissertation consists of lists of tables and figures, an introduction, four chapters, a conclusion, a list of the author's publications on the topic, a list of the noted citations, appendices, a list of references, and a statement of originality.

The main text of the dissertation consists of 163 pages and is accompanied by 2 (two) appendices (2 pages).

Chapter 1 Overview contains theories of the formal languages and their processing abstract machines related to translation. The elements of a translator that perform lexical and syntactic analyses are discussed, as well as the generative grammars they use. The chapter provides an overview of finite automata, push-down automata, Turing machines, Markov algorithms, etc. The overview includes well-known algorithms for recognition and parsing working on the basis of context-free grammars.

Chapter 2 Parsing machine contains formal definitions of advanced grammars, a phrase machine that is created on the basis of these grammars, and a detailed description of a parsing machine. The chapter contains the definitions of languages that are defined by advanced grammars with advanced symbols. The parsing machine shown contains different types of modules, such as supplier, scanner, lexer, parser, optimizer, builder, and filter. The different modules and their functionalities are described in detail. The chapter concludes with the definitions of different types of syntax trees and the construction commands on the basis of which the trees can be created.

Chapter 3 Tunnel parsing algorithm contains a detailed description of the Tunnel parsing algorithm. The various objects that are created before the parsing begins, based on a given advanced grammar, are described in detail. The list of these objects includes: execution stack, depth stack, repetition stack, archive stacks, automata, reachable trees, conflicts, tunnels, routers, and control objects. The detailed description of the control objects contains the steps that the parser performs and is effectively the pseudocode of the Tunnel parsing algorithm. The chapter contains an example

of the defined objects that are used by the algorithm and concludes with an example execution of the algorithm for a selected string of input data.

Chapter 4 Profiling of automatically generated parsers discusses a tool specially created for the purpose of the dissertation, called a profiler, with the help of which a large number of context-free grammars and inputs can be generated and experiments can be performed with parsers that are generated by various parser generators based on these grammars. The chapter contains a description of a template grammar language specially created for the purpose of the dissertation for imperative metaprogramming of grammars. The different grammars that define the scripts that are valid according to the template language are shown. The chapter concludes with the interpretation of the results of four different experiments that were conducted using the profiler.

In the conclusion, the main results are summarized and systematized, indicating the scientific, scientific-applied, and applied contributions of the dissertation work. The prospects for the development of the dissertation topic are formulated.

The list of used literature includes 191 (one hundred and ninety-one) titles, two of which are in Bulgarian, two are in Russian, and the rest of them are in English.

Approbation

The main results of the research were reported at an international conference and an international scientific forum – 14th International Conference Education and Research in Information Society (ERIS), Plovdiv and 8th Summer School, CEFP 2019, Budapest, Hungary.

The results of the dissertation research are presented in 6 (six) publications - 4 (four) in specialized journals, 1 (one) - in the proceedings of an international conference, and 1 (one) - in the proceedings of an international forum. The six publications are indexed in world-renowned databases: 4 (four) in Web of Science and 5 (five) in Scopus. Five of the publications are in editions with SJR.

Noted is a citation of 1 (one) of the publications on the topic in one scientific study, which is indexed in the world databases.

Acknowledgments

I express my gratitude to my supervisor, Prof. Dr. Elena Somova, for giving me the opportunity to conduct the dissertation research in the field and for the large amount of constructive criticism that led to a higher level of the dissertation. I thank Prof. Dr. Hristo Kiskinov, Prof. Dr. Emil Hadzhikolev, and Ch. Asst. Prof. Dr. Eng. Georgi Pashev for the valuable recommendations about the improvement of the dissertation, as well as the entire department of "Computer Informatics" for the support during the doctoral studies. I also thank my wife, Polina Handzhiyska, for her support, patience, and help in finding some inconsistencies in the working version of the dissertation. I thank my mother, Yordanka Katsarova, for the suggested linguistic improvements to unclear statements in the working version of the dissertation. Last but not least, I thank Dimitar Nanev for his moral support.

I dedicate this work to my daughter, Maya Handzhiyska.

1 Overview

The theory of translation includes the work with the formal languages and the abstract machines that process them [1, p. 208t], with which translators work. It is generally accepted that a translator performs lexical and syntactic analyses.

1.1 Formal means

The formal grammars and their corresponding abstract machines make it possible to write languages in various forms and to check for the membership of arbitrary strings in these languages. Combining formal means enables working with programs written in programming languages.

Switching Circuits

A way to work with switching circuits composed of relays is considered by Shannon (Claude Elwood Shannon), who draws a parallel between propositional logic (sentential logic) and the circuits that implement it.

Neural Networks

The time a network operates is assumed to be discrete (divided into consecutive and equal time intervals) and starts at one. At each time interval, all neurons either fire or do not fire an impulse. Depending on how a network can write and read information in the environment, different, more complex types of machines are possible.

Regular expressions

In [2, p.104c] X^+ is used as an abbreviation of **XX***.

An efficient way to transform a finite automaton into a regular expression is to use a generalized nondeterministic finite automaton — an automaton with transitions consisting of regular expressions [3, p. 70t]. Every nondeterministic automaton is generalized, but not vice versa.

In practice, user-written regular expressions do not always define exactly the strings that the user expects [4].

Generative grammars, Chomsky's hierarchy

The Augmented Backus-Naur Form (ABNF) is described in [5] and [6]. The more expressive capability a metalanguage has, the easier it is for the developer to develop the grammar that defines the object language.

The languages defined by four types of grammars (unrestricted, context-sensitive, context-free, and regular) form Chomsky's hierarchy.

It is said (already used in some cases above) that all activities (generation of strings in the language, which is defined by the grammar, data recognition, etc.) are performed based on the grammar.

Context-free grammars

The renaming rules can be removed from a grammar without changing the language, but this will change the syntax trees that are generated based on the grammar. In other words, the transformation is always possible, but not always acceptable.

In a self-embedding grammar for an active and a central nonterminal **A**, there exists $A \Rightarrow^+ xAy$ for $x, y \in V^+$ [7, p. 148b].

Finite automata

In [8, p. 210t], it is noted that if the function **f** outputs more than one result, then the automaton becomes nondeterministic. All tapes **T(A)** recognized by a finite automaton **A** are regular language.

The size of a finite automaton can directly affect the time it takes to recognize strings. For this reason, it is desirable that the finite automata used for string recognition have as few states and transitions as possible.

Push-down automata

The transition function is $\delta:S\times(\Sigma\cup\{\varepsilon\})\times\Gamma\rightarrow^{\{\}}S\times\Gamma^*$, where $\rightarrow^{\{\}}$ is defined to mean that the function can return more than one result as a sequence and that all results are elements of a given set.

A push-down automaton is deterministic if the automaton always has at most one possible transition to apply. All other push-down automata are nondeterministic.

If a push-down automaton can recognize each string without using more than a constant number of symbols in the stack, then this push-down automaton can be transformed into a (deterministic) finite automaton [9, p. 259b].

Definition of an algorithm

A normal Markov algorithm has a shorter description than the equivalent Turing machine. One reason is that the result is obtained by always replacing the leftmost of the embedded words when applying a given formula. Thus, an explicit description of the head motion in the equivalent Turing machine is not required.

A function that always halts [1, p.208t] for all possible arguments after a finite number of executed instructions is called an algorithm [2, p.27t].

Closure of the operations and decidability of the problems

According to [10, p. 159b], for arbitrary context-free languages **A** and **B**, it is undecidable whether: $L(A)=L(B)$; $L(A)\subset L(B)$; and $L(A)\cap L(B)=\emptyset$.

1.2 Translators

Translation is a fundamental process when working with programs written in programming languages. The finite and the push-down automata find application in the translation.

Another possibility is that the language does not have a lexer grammar (but only a parser grammar) and therefore does not have a lexer. The parsing of this kind of language is called single-phase parsing in [11, p. 3rb] and scannerless parsing in [12].

The translators are required for the programming languages. Since every programming language is a formal language, the theory described above applies directly to the creation of translators.

It is possible for the lexer to be context-aware [13] — the lexer sends multiple token types, taking into account the current state of the parser.

A popular way for a parser to work with the tokens is to use them for recognition by a push-down automaton that is generated based on context-free grammar (the type of grammar used in this section).

The development of grammars and programming languages is directly related to the compilation process.

1.3 Conclusions

Many recognition and parsing algorithms exist. The parsing algorithms often have problems with the grammars having rules that generate empty words. The grammars in ABNF contain various objects that can generate an empty word. Some of these objects are the rules. Based on the research in this chapter, the following conclusions are drawn:

- The popular parsing algorithms do not support repetitions of grammar elements that can exist in one ABNF grammar and perform many operations during analysis that can be precomputed from the grammar;

- A key task of the parsing algorithms is to quickly use rules that generate empty words. In [14], it is shown that this can lead to a lot of work for a graph that does not contain a single character;
- The lexers in the literature group sequences of characters into a token with a name (a unique identifier according to the rule used for the grouping). According to the ABNF standard [5], **`%x30-32`** is a character range that can be represented as the set of characters **`{'0','1','2'}`**. If the parser works with a grammar in ABNF, then the following range problem exists — how to parse based on the range **`%x30-32`** and which part of the token to compare to;
- This creates the following case problem — how to distinguish the token names from the rule names in the parser grammar;
- There is a lack of detailed empirical measurements about the performance of different automatically generated parsers that provide information on the expected amount of resources that are necessary to parse data according to specific grammar elements;
- A parsing machine is necessary to combine the parsing with and without lexical analysis in such a way that the benefits of both ways are combined into one, and the user can choose whether to have lexical analysis.

2 Parsing machine

This chapter outlines the design of a machine for parsing data, called a parsing machine, in which various parsing algorithms can be used.

Informally, a parsing machine is an abstract machine built from modules with connections between them that works by taking unstructured data as an input and sending structured data as an output. Each parsing machine always scans (reads) the input data and always performs syntactic analysis.

2.1 Basic concept of the machine

The formalization of the concept of a parsing machine is necessary, so that the interactions between the modules in the machine and the specific special cases are shown. In order for the machine to use the tokens, a new kind of grammar is needed to enable the parsing based on all tokens (except the limit token) in the machine.

The parsing machine is designed to distribute the analysis of the input data between the different modules for the purpose of parallelism and to allow different analysis strategies to be applied by the different modules. In the machine, the following modules are distinguished: supplier, scanner, lexer, parser, optimizer, builder, and filter.

Similar to the working with tokens by the scanner, the lexer, and the parser, down is discussed the working with commands for building syntax trees by the parser, the optimizer, and the builder.

2.2 Characters and lexemes in the machine

It is reasonable to expect a machine that parses textual data to be able to accept (at least) UTF-8 [15] encoded input and to handle the encoding errors automatically. A character is defined as a Unicode position — a nonnegative integer, with a largest possible value that depends on the Unicode standard used. The set of characters is denoted by **`U`**, and it is emphasized that there is a character with Unicode position zero.

Many authors use Unicode positions in the implementation of translators, but at a theoretical level, they work with abstract letters. In the theory of the proposed parsing machine, the characters are specified as Unicode positions and are required for all modules in the machine (some of which

are new or with new functionalities), not just for the lexer and the parser. The concretization of the characters brings the theory closer to practice and enables the formal treatment of some practical problems.

2.3 Formal definition of a parsing machine

A parsing machine is the n-tuple (M,T,I,S,L,P,O,B,F), where M is a finite nonempty set of modules, C is a finite nonempty set of connections between the modules, $I \in M$ is a nonempty set of suppliers, $S \in M$ is a scanner, $L \in M$ is a set of lexers, $P \in M$ is a parser, $O \in M$ is a set of optimizers, $B \in M$ is a nonempty set of builders, and $F \in M$ is a set of filters. A module is an output module when it sends data outside the machine.

2.4 Tokens

In order to solve the range and case problems, several types of tokens are distinguished, which are intended to be used as application data on the transition from the scanner to the lexer, on the transition from the lexer to the parser, or on the transition from the scanner to the parser (if there is no lexer).

A module is classified as a transmitter when it outputs tokens to another module that is classified as a receiver. For a transmitter that uses grammar $G=(\Phi,\Theta,-,-)$ for nonterminals Φ and characters $\Theta \subseteq U$, the following is defined:

- An attribute is written as $l=v$, where l is a nonempty string of characters (a label) and v is a value with a domain, which depends on the label;
- A character token is the n-tuple $(\texttt{t-character},n,\alpha)$, where $n \in \Theta$ is a name and α is a finite set of attributes. If $|\alpha|=0$, then the token is written as $(\texttt{t-character},n)$;
- A sequence token is the n-tuple $(\texttt{t-sequence},n,e,\alpha)$, where $n \in \Phi$ is a name, e is a string over Θ (a lexeme), $|e|>0$, and α is a finite set of attributes. If $|\alpha|=0$, then the token is written as $(\texttt{t-sequence},n,e)$. The unbounded length of e makes these token elements in an infinite set;
- A limit token is the n-tuple $(\texttt{t-limit},\beta,e,\alpha)$, where $\beta \subseteq \Phi$ are nonterminals, $|\beta|>0$, e is a lexeme, $|e|>0$, and α is a finite set of attributes. If $|\alpha|=0$, then the token is written as $(\texttt{t-limit},\beta,e)$;
- An eof token is the n-tuple $(\texttt{t-eof},\alpha)$, where α is a finite set of attributes. If $|\alpha|=0$, then the token is written as $(\texttt{t-eof})$;
- The infinite set of tokens is denoted by H;
- The type of $h \in H$ is the first element in the n-tuple of h, which is obtained as a result of the execution of the function $\Pi: H \rightarrow H'$ for $H'=\{\texttt{t-character},\texttt{t-sequence},\texttt{t-limit},\texttt{t-eof}\}$.

In the above definitions of tokens, only a sequence token (of the `t-sequence` type) has a lexeme that can be used for parsing, and the character token (of the `t-character` type) has a name instead of a lexeme. This difference from the literature sources enables different ways of working with these two types of tokens. The limit token (of the `t-limit` type) provides the theoretical basis for the machine implementations to correctly handle situations (instead of freezing) where the working with tokens machine modules do not have enough memory to continue their work.

2.5 Advanced level 1 grammars

In order to solve the case problem and to enable the use of the lexeme in the **t-sequence** tokens for analysis, a new type of symbols are added to the grammars, which are called phrase symbols.

Defined are the set of comparators $\mathbf{M=\{\approx,\equiv\}}$, where $\approx$ means insensitive comparator, $\equiv$ means sensitive, and the case function $\mathbf{\Delta:U\times U\times M\to^{\{\}}U}$. For example, $\mathbf{\Delta('a','a',\approx)=\{'a','A'\}}$.

2.6 Advanced level 2 grammars

Advanced level 2 grammars are an upgrade of the level 1 grammars. The goal of this upgrade is to bring the advanced grammars closer to those in ABNF by adding groups, concatenations, alternations, symbol repetitions, and a symbol defining the empty word.

An advanced level 2 grammar is n-tuple $\mathbf{A=(C,N,\Sigma,\Omega,R,S)}$ for a set of categories $\mathbf{C}$, a set of nonterminals $\mathbf{N}$, a set of characters $\mathbf{T\subseteq U}$, a set of advanced symbols $\mathbf{\Omega}$, a set of rules $\mathbf{R}$, and a set of start nonterminals $\mathbf{S\subseteq N}$.

It is defined that:

- Every element $\omega\in\Omega$ is an indexed n-tuple — an n-tuple that holds the symbol index $\mathbf{k}\in\mathbb{N}_0$, which is written as $\alpha^k\in\Omega$ for n-tuple α. The index is not written when it is not used;
- $\nexists x,y\,|\,x=y\wedge\alpha^x\in\Omega\wedge\beta^y\in\Omega\wedge\alpha\neq\beta$ — all symbol indices in Ω are distinct;
- The type of $\omega\in\Omega$ is the first element in the n-tuple of ω (with an index of zero) and is obtained as a result of the execution of function $\Pi:\Omega\to\Omega'$ for $\Omega'\in$ **{s-reference, s-character, s-phrase, s-eof, s-concatenation, s-alternation, s-group, s-repeat, s-epsilon}**;
- The access function to the elements in the n-tuples is defined as $\mathbf{GET}:\Omega\times\mathbb{N}_0\to\Omega'\cup C\cup N\cup\Omega$, which, upon execution of $\mathbf{GET}(\omega,n)$ for $\omega\in\Omega$ and $n\in[0..|n|)$, returns the element at index $\mathbf{n}$ in the n-tuple of ω and $\Pi(\omega)=\mathbf{GET}(\omega,0)$.

The definitions of advanced level 1 grammars are carried over to level 2, with the n-tuples becoming indexed:

- A phrase is a string of characters over Σ;
- A reference symbol is an indexed n-tuple $(\mathbf{s\text{-}reference},n)\in\Omega$ for $n\in N$;
- A character symbol is an indexed n-tuple $(\mathbf{s\text{-}character},f,t,m)\in\Omega$ for $f,t\in\Sigma$, $f\leq t$, $m\in M$;
- A phrase symbol is an indexed n-tuple $(\mathbf{s\text{-}phrase},c,p,m)\in\Omega$ for $c\in C$, phrase $\mathbf{p}$, and $m\in M$;
- An eof symbol is an indexed n-tuple $(\mathbf{s\text{-}eof})\in\Omega$.

The following level 2 symbols are added:

- A concatenation is an indexed (n+1)-tuple $(\mathbf{s\text{-}concatenation},\omega_0,\ldots,\omega_{n-1})\in\Omega$ for $\omega_i\in\Omega$, $n>0$ and $\Pi(\omega_i)\notin\{\mathbf{s\text{-}concatenation},\mathbf{s\text{-}alterantion}\}$;
- An alternation is an indexed (n+1)-tuple $(\mathbf{s\text{-}alternantion},\omega_0,\ldots,\omega_{n-1})\in\Omega$ for $\omega_i\in\Omega$, $n>0$ and $\Pi(\omega_i)=\mathbf{s\text{-}concatenation}$;
- A group symbol is an indexed n-tuple $(\mathbf{s\text{-}group},\omega)\in\Omega$ for $\omega\in\Omega\wedge\Pi(\omega)=\mathbf{s\text{-}alternantion}$;
- A repetition symbol is an indexed n-tuple $(\mathbf{s\text{-}repeat},n,m,\omega)\in\Omega$ for $n\in\mathbb{N}_0$, $m\in(\mathbb{N}_0\cup\infty)$, $n\leq m$, $n\neq1\vee m\neq1$, $\omega\in\Omega$ and $\Pi(\omega)\in\{\mathbf{s\text{-}reference},\mathbf{s\text{-}group},\mathbf{s\text{-}character},\mathbf{s\text{-}phrase},\mathbf{s\text{-}eof}\}$. A repetition with $n>1\vee(m>1\wedge m\neq\infty)$ is called a countable repetition;

- An epsilon symbol is an indexed n-tuple **(s-epsilon)∈Ω**.

The binary irreflexive relation ↦ is defined, as for **x,y∈Ω**, it is true that **x↦y** if and only if **GET(x,n)=y** for some **n∈[0..|x|)**. The sign ↦ is read as "has a pointer to". When **x** has a pointer to **y**, it will be said that **y** is located in **x** and that **x** contains or uses **y**. The transitive closure [1, p. 26c] of ↦ is defined as **↦***, which is read as "has a transitive pointer to".

The rules in an advanced level 2 grammar can be in a different form. An advanced level 2 context-free grammar is defined as an advanced level 2 grammar having rules in **R** with the form **A→α** for **A∈N** and **Π(α)=s-alternation**. From here on, unless otherwise stated, an advanced grammar means an advanced level 2 context-free grammar.

2.6.1 Well-defined advanced grammars

An advanced grammar **A=(-,-,-,Ω,R,-)** is well-defined when each symbol in **Ω** is used by one and only one object in **A** and each symbol is used (transitively) in a single rule.

The following is defined:

- ⟦ ⟧ denotes a set with repeated elements;
- $\mathbf{POINTER_{\Omega}(\omega)=[\![x|(x\mapsto\omega|x\in\Omega)]\!]}$ are all symbols that use **ω**;
- $\mathbf{POINTER_{R}(\omega)=[\![r|(r\mapsto\omega|r\in R)]\!]}$ are all rules in the grammar that use **ω**;

An advanced grammar **A** is well-defined if and only if the following conditions hold:

1. $\mathbf{|POINTER_{\Omega}(\omega)|=1}$ for each **ω∈Ω|Π(ω)≠s-alternation**;
2. $\mathbf{|POINTER_{\Omega}(\alpha)|+|POINTER_{R}(\alpha)|=1}$ for each **α∈Ω|Π(ω)=s-alternation**;
3. For each **ω∈Ω**, it is true that **r↦*ω** for some **r∈R**.

From here on, all grammars are assumed to be well-defined. When a grammar is well-defined, it enables one to uniquely determine the number of repetitions of a given symbol by defining and using the function $\mathbf{REPEAT:\Omega\to\mathbb{N}_0\times(\mathbb{N}_0\cup\infty)}$.

2.6.2 Sets of symbols in advanced grammars

From here on, the following denotations are used:

- All reference symbols in **Ω** are $\mathbf{N_{\Omega}=\{\omega|\omega\in\Omega\wedge\Pi(\omega)=s\text{-}reference\}}$;
- All phrase symbols in **Ω** are $\mathbf{P_{\Omega}=\{\omega|\omega\in\Omega\wedge\Pi(\omega)=s\text{-}phrase\}}$;
- All phrase symbols with an empty phrase are called universal phrase symbols (or **s-universal**) and are $\mathbf{PA_{\Omega}=\{\omega|\omega\in P_{\Omega}\wedge|\omega.p|=0\}}$;
- All nonempty phrase symbols with a sensitive comparator are called sensitive phrase symbols (or **s-sensitive**) and are $\mathbf{PS_{\Omega}=\{\omega|\omega\in P_{\Omega}\wedge|\omega.p|>0\wedge|\omega.m|=\equiv\}}$;
- All nonempty phrase symbols with an insensitive comparator are called insensitive phrase symbols (or **s-insensitive**) and are $\mathbf{PI_{\Omega}=\{\omega|\omega\in P_{\Omega}\wedge|\omega.p|>0\wedge|\omega.m|=\approx\}}$;
- All eof symbols in **Ω** are $\mathbf{F_{\Omega}=\{\omega|\omega\in\Omega\wedge\Pi(\omega)=s\text{-}eof\}}$;
- All group symbols in **Ω** are $\mathbf{G_{\Omega}=\{\omega|\omega\in\Omega\wedge\Pi(\omega)=s\text{-}group\}}$;
- All repetition symbols in **Ω** are $\mathbf{Y_{\Omega}=\{\omega|\omega\in\Omega\wedge\Pi(\omega)=s\text{-}repeat\}}$.

2.6.3 Generation of strings

The **Δ** is extended to also work with strings, as $\mathbf{\Delta:U^{*}\times M\to^{\{\}}U^{*}}$. For example, **Δ("ab",≈)={"ab","aB","Ab","AB"}**.

Each advanced symbol **ω∈Ω** defines a set of strings over **N∪H∪Ω**. That **ω** defines a string **β∈(N∪H∪Ω)*** is written as **ω⇉β**. Below are the strings that are defined by the various advanced symbols:

- For advanced grammars at all levels:
  - If $\omega=(\texttt{s-reference},\mathbf{n})\in\Omega$, then $\omega\rightrightarrows\mathbf{n}$;
  - If $\omega=(\texttt{s-character},f,t,m)\in\Omega$, then $\omega\rightrightarrows(\texttt{t-character},x)$ for each character $x\in\Delta(f,t,m)$;
  - If $\omega=(\texttt{s-phrase},c,-,-)\in PA_\Omega$, then $\omega\rightrightarrows(\texttt{t-sequence},c,\beta)$ for each string $\beta\in T^+$;
  - If $\omega=(\texttt{s-phrase},c,p,m)\in(PS_\Omega\cup PI_\Omega)$, then $\omega\rightrightarrows(\texttt{t-sequence},c,\beta)$ for each string $\beta\in\Delta(p,m)$;
  - If $\omega=(\texttt{s-eof})\in\Omega$, then $\omega\rightrightarrows(\texttt{t-eof})$.
- For advanced level 2 grammars:
  - If $\omega=(\texttt{s-concatenation},\omega_0,\ldots,\omega_{n-1})\in\Omega$, then $\omega\rightrightarrows\omega_0\ldots\omega_{n-1}$;
  - If $\omega=(\texttt{s-alternation},\omega_0,\ldots,\omega_{n-1})\in\Omega$, then $\omega\rightrightarrows\omega_i$ for each $i\in[0..n)$;
  - If $\omega=(\texttt{s-group},\alpha)\in\Omega$, then $\omega\rightrightarrows\alpha$;
  - If $\omega=(\texttt{s-repeat},n,m,x)\in\Omega$, then $\omega\rightrightarrows\beta$ for each string β over $\{x\}$, where $n\le|\beta|\le m$;
  - If $\omega=(\texttt{s-epsilon})\in\Omega$, then $\omega\rightrightarrows\varepsilon$.

An advanced symbol $\omega\in\Omega$, which defines a string γ, can be <u>decomposed</u> to the string $\alpha\omega\beta$, as the result of the decomposition of the symbol is $\alpha\gamma\beta$, where $\alpha,\beta\in(N\cup\Omega\cup H)^*$. The sequence $(\mu_0,\ldots,\mu_n)$ for $n\ge0$ is a φ decomposition of ψ, if $\varphi,\psi\in(N\cup H\cup\Omega)^*$, $\varphi=\mu_0$, $\mu_n=\psi$ and μ_{i+1} is the result of the decomposition of some advanced symbol in μ_i for $i\in[0..n)$. The decomposition is <u>terminal</u> when $\mu_n\in(N\cup H)^*$. It is said that from φ can be derived ψ, when there exists a φ decomposition of ψ.

The <u>normal application</u> of rule $D\to\gamma\in R$ to a string $\alpha D\beta$ completes with result $\alpha\gamma\beta$, where $\alpha,\beta\in(N\cup\Omega\cup H)^*$. From here on, all rule applications are normal. The sequence $(\mu_0,\ldots,\mu_n)$ for $n\ge0$ is a <u>normal</u> φ derivation of ψ, if $\varphi,\psi\in(N\cup H\cup\Omega)^*$, $\varphi=\mu_0$, $\mu_n=\psi$ and μ_{i+1} for $i\in[1..n)$ is the result of: a) the normal application of some rule in R to μ_i; or b) the decomposition of advanced symbol $\omega\in\Omega$ in μ_i. From here on, all derivations are normal. The number of steps in a normal φ derivation of ψ is n. The normal derivation is <u>terminal</u> when $\mu_n\in H^*$. It is said that from φ can be derived ψ, when there exists a normal φ derivation of ψ. With $\varphi\Rightarrow\psi$, a one-step derivation is written, with $\varphi\Rightarrow^*\psi$ — a derivation with zero or more steps, and with $\varphi\Rightarrow^+\psi$ — a derivation with one or more steps.

The infinite set of character and sequence tokens is $H_{CS}=\{h\mid h\in H\wedge\Pi(h)\in\{\texttt{t-character},\texttt{t-sequence}\}\}$. The language generated by a given start rule $J\in S$ in grammar A is $L_A(J)=J_{CS}\cup J_E$, where $J_{CS}=\{w(\texttt{t-eof})\mid J\Rightarrow^+w\wedge w\in H_{CS}^*\}$ and $J_E=\{w\mid J\Rightarrow^+w\wedge w\in H_{CS}^*(\texttt{t-eof})\}$. When the grammar is implied, instead of $L_A(J)$, just $L(J)$ will be written.

2.6.4 Sets of tokens

The advanced symbols that directly define a token are in the set $H_\Omega=\{\omega\mid\omega\in\Omega\wedge\Pi(\omega)\in\{\texttt{s-character},\texttt{s-phrase},\texttt{s-eof}\}\}$. The function $\Psi:H_\Omega\to^{\{\}}H$ is defined such that executing $\Psi(\omega)$ with $\omega\in H_\Omega$ returns as a result the set of tokens defined by ω, according to the definitions above. It is said that two advanced symbols $x,y\in\Omega$ <u>overlap</u> when $\Psi(x)\cap\Psi(y)\neq\varnothing$ (i.e., when the sets the symbols define have at least one element in common).

2.6.5 Metagrammar for advanced grammars

In order to define advanced grammars, several rules are added to the ABNF standard because: a) the standard does not allow Unicode characters; and b) the standard cannot express phrase symbols, nor can it express with a single element the symbol $(\texttt{s-character},x,x,\approx)$, when $|\Delta(x,x,\approx)|>1$. The result obtained from the ABNF standard with the above additions is called Advanced ABNF (AABNF).

2.7 Phrase state machine

During parsing based on an advanced grammar, it is possible for the parser to check whether a given token $h\in H$ simultaneously belongs to a large number of sets, each of which results from $\Psi(\omega)$ for $\omega\in Z$ and $Z\subseteq H_\Omega$.

From the sets that are defined by phrase symbols, it follows directly that all phrase symbols $P\subseteq Z$ can be divided into sets that have no elements in common P_c according to their category — $P_c=\{\omega|\omega\in P\wedge\omega.c=c\}$.

To make token-to-phrase comparisons fast, a phrase machine is created that classifies the different lexemes in the tokens and the phrases in the grammar's phrase symbols, so that instead of comparisons between strings, comparisons are made between classes of strings.

Each unique phrase in a sensitive phrase symbol is assigned a unique sensitive index. Each phrase in an insensitive phrase symbol is assigned an insensitive index.

After all the phrases in the phrase symbols are classified, in order to check the membership of a given sequence token in the sets of tokens defined by the different phrase symbols in a given set P_c, a series of actions must be performed.

A phrase state machine (phrase machine for short) is the n-tuple (Σ,Q,δ,F,q_0) for alphabet $\Sigma\subseteq U$, a nonempty set of states Q, a transition function $\delta:Q\times\Sigma\rightarrow Q\times M$, a set of final states F, and a start state $q_0\in Q$.

The transition function δ is represented as a set of transitions in the form $\alpha\rightarrow\beta$ for n-tuple of arguments $\alpha=(s,\sigma)$, n-tuple of results $\beta=(d,m)$, input $s\in Q$, transition character $\sigma\in\Sigma$, output $d\in Q$, and comparator $m\in M$. Each final state is in the form $q\rightarrow(n,m)$ for $q\in Q$, $n\in\mathbb{N}_1$, and $m\in M$. No two final states have the same q — $\nexists x,y|x,y\in F\wedge x\neq y\wedge x.q=y.q$. For $x\in\delta$, it is said that x is a sensitive transition when $x.m=\equiv$ and an insensitive transition when $x.m=\approx$.

In order for the machine to work in the required way, it must be well defined, which requires additional constraints on the formal definition. The first way to analyze a string w on a well-defined phrase machine is called sensitive analysis. In this type of analysis, the phrase machine works like a finite automaton. The second way to analyze a string w by a well-defined phrase machine is called insensitive analysis.

Function **INSENSITIVE:Σ*→ℕ₀×M** with a parameter **w**

```
 1. begin
 2.   c←q₀, i←0                         ▹ preparation
 3.   while i<|w| do                    ▹ steps
 4.     l←LOWER(wᵢ), u←UPPER(wᵢ)        ▹ variants of wᵢ
 5.     if l=u then                     ▹ test for one variant
 6.       t|t∈δ∧t.s=c∧t.σ=l             ▹ search
 7.       if ∄t then return (0,≡)       ▹ failure if ∄t
 8.       c←t.d                         ▹ next state
 9.     else
10.       L|L∈δ∧L.s=c∧L.σ=l             ▹ search
11.       if ∄L then return (0,≡)       ▹ failure if ∄L
12.       U|U∈δ∧U.s=c∧U.σ=u             ▹ search
13.       if ∄U then return (0,≡)       ▹ failure if ∄U
14.       if L.m=≈∧U.m=≈ then c←L.d
15.       else if L.m=≈ then c←L.d      ▹ next state
16.       else if U.m=≈ then c←U.d      ▹ next state
17.       else c←L.d                    ▹ preference of L
18.     i←i+1                           ▹ next character
19.   f|f∈F∧f.q=c                       ▹ search
20.   if ∃f return (f.n,f.m)            ▹ success if ∃f
21.   return (0,=)                      ▹ failure
22. end
```

Figure 1: Pseudocode for insensitive analysis by a phrase machine

The formal description of the insensitive analysis is shown in pseudocode in Figure 1.

An advanced grammar can be compiled to a phrase machine using the phrase symbols in the grammar.

The result of classifying each string is the n-tuple **(n,m)** for $n \in \mathbb{N}_0$ and $m \in M$.

2.8 Modules of a parsing machine

After defining the different objects that are used in the machine, this section will introduce the modules that work with these objects.

The first module in a machine is called a supplier. This module outputs sequences of bits to the next module. There is at least one supplier in a machine.

A scanner is called the module that accepts a sequence of bits from the supplier and decodes them into characters that it outputs to the next module in the machine. The decoding can be according to different types of standards, but the decoding is assumed to be according to the Unicode standard used by the machine.

For each decoded character **ψ** from the input data, the scanner outputs a token **(t-character,ψ,α)**. This module serves as a "border" in the machine that separates the bit operations and the token operations.

A lexer is the module that accepts tokens from the previous module and outputs the same or different tokens to the next module, and the tokens that are output depend on the specific lexer. There can be zero (meaning a lexerless machine) or more lexers, all ordered one after the other and all after the scanner.

No strict restriction is placed on the type of grammar in the lexer specification, but for convenience, a normal grammar **G=(N,–,–,–)** will be used. The rule in the lexer grammar, based on which the lexer accepts the longest possible sequence of tokens, is denoted by **q**. The traditional way the lexer works changes (if there is no rule in the grammar that accepts the current data, it is an error) by defining a new one. At any moment of the lexer's work:

- If rule **q** is uniquely established (after at least one accepted token), then:
  1. The lexer outputs **t–sequence(n,e,α)** token, where **n∈N** is the name of **q** and **e** is a string of the names of the accepted **t–character** tokens based on rule **q**. If the machine works with certain attributes, then the lexer adds them to **α**;
  2. The lexer removes the used **t–character** tokens;
  3. The analysis starts over with the remaining tokens that have been accepted by the scanner.
- If the lexer finds that rule **q** will not be established, then:
  1. The lexer outputs to the parser the first of the tokens it received from the scanner;
  2. The analysis starts over with the remaining tokens.
- At the moment the lexer reaches its limit (if any) before rule **q** is uniquely established, then:
  1. The lexer outputs a **t–limit(Q)** token to the parser, where **Q** is the set of rule names that the lexer did not establish that they do not accept the string of tokens accepted from the scanner;
  2. The analysis stops.
- If the lexer needs to analyze only one remaining token of type **t–eof**, then:
  1. The lexer sends an eof token to the next module;
  2. The analysis stops.

A lexer that works in the described way is called a continuous lexer.

A parser is called the module that accepts input tokens, analyzes them based on a given specification, and outputs syntax structure construction commands (short for just commands).

An optimizer is called the module that accepts commands and outputs the same or other commands to the next module, where the commands output depend on the specific optimizer.

A builder is called the module that accepts commands from the previous module and performs activities that are related to the syntax structure of the tree. The builder module is a "border" between working with commands and working with syntax structures.

A filter is called the module that accepts syntax structures and outputs the same or other syntax structures, and the structures that are output depend on the particular filter.

2.9 Syntax trees

A popular understanding is that the abstract syntax trees may not contain all possible nodes labeled with nonterminals and may not contain certain nodes when they are implied by the context. These informal definitions, while expressing the difference between trees, leave much unclarity as to how much information is "enough" to distinguish different parts of a tree and by what criteria a particular tree becomes abstract.

To derive a syntax tree, the builder uses facts. The set of all facts is defined by **Φ**. Depending on the relationships between the objects in the parser grammar, groups of facts derived from the advanced grammar **A=(–,–,–,–,R,–)** are distinguished, which is used for parsing by the parser in the machine:

- Which alternation is in which rule or group;
- Which concatenation is in which alternation;

- Which token is analyzed by the parser as part of the set that is defined by a particular symbol, before the token is output to the builder;
- Which nonterminal is referenced by which reference symbol.

Depending on the repetition of a given symbol in **A**, several types of facts are distinguished. These facts are mutually exclusive for each individual symbol and are as follows:

- skippable — exists for the symbol $\omega^k \in \Omega$ that can be recognized in the input data zero or one number of times. Formally, there is one fact **(f-skippable,k)** for each of the symbols $\omega^k | \omega^k \in \Omega \wedge \mathbf{REPEAT}(\omega^k) = (0,1)$;
- single — exists for the symbol $\omega^k \in \Omega$, which must be in the input data exactly once. Formally, there is one fact **(f-single,k)**$\in \Phi$ for each of the symbols $\omega^k | \omega^k \in \Omega \wedge \mathbf{REPEAT}(\omega^k) = (1,1)$;
- array — exists for the symbol $\omega^k \in \Omega$, which must be in the input data, when the minimum and maximum number of repetitions of the symbol are equal, are greater than one, and are a finite number. Formally, there is one fact **(f-array,k,n)**$\in \Phi$ for each of the symbols $\omega^k | \omega^k \in \Omega \wedge \mathbf{REPEAT}(\omega^k) = (n,n) \wedge n > 1 \wedge n \neq \infty$;
- list — exists for the symbol $\omega^k \in \Omega$, which must be in the input, when the minimum number of occurrences of the symbol is less than the maximum number and the maximum number is greater than one. Formally, there is one fact **(f-list,k,n,m)**$\in \Phi$ for each of the symbols $\omega^k | \omega^k \in \Omega \wedge \mathbf{REPEAT}(\omega^k) = (n,m) \wedge n < m \wedge m > 1$;

The facts that the builder possesses are defined by $K \subseteq \Phi$. It is assumed that the parser has all the knowledge Φ and that the builder uses all the knowledge it possesses.

The syntax trees are classified according to the structure of the advanced grammars in AABNF. One of the most important properties of the advanced grammars is that they have a minimum and maximum number of repetitions for the individual elements.

According to the information contained in the tree, the syntax trees are as follows:

- Concrete — a syntax tree that is built based on all available facts, when $K = F$;
- Abstract — a tree that is built without using at least one required fact, when $K \subset F$. This means that the maximum number of abstract trees is $2^{|F|-1}$ (to be sure that the tree is not concrete, one is subtracted from the number of facts).

Commands for preparation (type **d-prepare**), unpreparation (type **d-unprepare**), enter for a rule (type **d-rule-enter**), success for a rule (type **d-rule-success**), back for a rule (type **d-rule-back**), failure for a rule (type **d-rule-fail**), enter/success/back/failure for a group (type **d-group-enter/success/back/fail**), enter/success/back/failure for a concatenation (type **d-con-enter/success/back/fail**), token forward (type **d-token-front**), token backward (type **d-token-back**), next element (type **d-next**), previous element (type **d-previous**), creation/destruction for a list (type **d-list-create/destroy**), creation/destruction for an array (type **d-array-create/destroy**), found error (type **d-error**), success of the analysis (type **d-success**), and end of the analysis (type **d-done**) are distinguished.

2.10 Conclusions

In the second chapter, a conceptual model of a parsing machine is presented, which is suitable for the implementation of various strategies, methods, approaches, and algorithms that are popular in the literature. Also defined is a new type of lexer module functionality that enables the parser module to parse according to the case of the characters in the lexemes and to accept characters according to character ranges.

An addition to the ABNF is proposed, with which the advanced grammars can be defined. A phrase machine model is proposed that categorizes in advance the different phrases in the parser grammar in order to speed up the analysis. A modification of the known in the literature way of operation of the scanner and lexer modules is proposed. New modules, such as a supplier and an optimizer, are defined.

The common architecture of a parsing machine is oriented more towards the types of data that are received and sent by the modules in the machine and less towards how they are processed.

3 Tunnel parsing algorithm

In this chapter, the Tunnel Parsing (TP) algorithm is proposed and discussed as an algorithm that is executed by the parser module in the parsing machine, introduced in the previous chapter.

3.1 Basic concept of the algorithm

The goal of the algorithm is to efficiently execute a push-down automaton having transitions with advanced symbols and countable repetitions, which is represented as connected transition diagrams. For efficient execution, the algorithm groups certain transitions and states of the push-down automaton into “parts”.

The situation is complicated by the fact that in the ABNF grammars there may be a repetition of a reference (for example, **`5*8R`**), with the referenced rule generating an empty word (for example, **`R=0*1"x"`**). In this situation, the TP parses as many repetitions as possible using tokens (for example, **`"xx"`**), and the remaining number of repetitions (**`3`**) up to the minimum required (**`5`**) are performed without tokens. The algorithm does not enter an infinite loop when the repetition is to infinity (for example, **`5*R`**).

3.2 Stacks

The TP algorithm uses an execution stack that consists of elements represented as an n-tuple (c,n) for control state c and the number of archived depth stack elements $n \in \mathbb{N}_0$.

At runtime, the TP algorithm uses a depth stack that consists of segments. One segment contains information about the operations that the parser can perform depending on the current input token. Each segment is represented as an n-tuple (p,n,m,rm,rd,rn) for parent $p \in (N_\Omega \cup G_\Omega \cup N)$, minimum number of repetitions n, maximum number of repetitions m, minimum router rm of the **`r-minimum`** type, inner router rd of the **`r-inner`** type, and next router rn of the **`r-next`** type. If $p \in (N_\Omega \cup G_\Omega)$, then $(n,m)=\mathrm{REPEAT}(p)$, and if $p \in N$, then $n=1$ and $m=1$.

For the different analyses that follow below, a repetition stack with N_0 elements called counters is used. This stack contains the number of repetitions that have already been found (or are in the process of being found) for a given advanced symbol. Repetitions are counted for the symbol $\omega \in \Omega$ when for $(n,m)=\mathrm{REPEAT}(\omega)$ is true that $n>1 \lor (m>1 \land m \neq \infty)$ (the repetition is countable).

At runtime, the TP algorithm can progress backward in the automata. To make this possible, the items in the depth stack are not deleted but are moved to the archive depth stack in an operation called archiving. Similarly, moving an element from the archive depth stack to the depth stack is called restoring.

The archive repetition stack works in a similar way — when the algorithm moves from recognizing a given advanced symbol to the next, the number of repetitions that have been recognized so far for the given element (if any) is archived in the archive repetition stack.

3.3 Automata

For the purpose of analyzing a string of tokens based on an advanced grammar, a sequence of automata is created. Each automaton is created recursively by applying various templates from which states and labeled transitions are created.

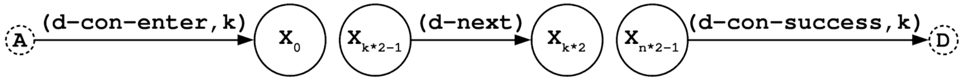

Figure 2: A template with states and transitions created on the basis of a concatenation

For each symbol **(s-concatenation,** ω_0**,…,**ω_{n-1}**)**, states and transitions are created by applying the template in Figure 2. States **A** and **D** are not created, but they are the context in which the template is applied. In Figure 2, **k** is the sequence number of the concatenation in the alternation that uses the concatenation. The pairs of states X_{k*2-1} and X_{k*2}, as well as the transitions between them, for **k∈[1..n−1]**, are created only if **n>1**. Each pair X_{i*2} and X_{i*2+1}, for **i∈[0..n−1]**, is the context in which the template for ω_i is applied.

The <u>parent of the state</u>, which is created when applying a given template for the symbol $\omega|(H_\Omega \cup N_\Omega \cup G_\Omega \cup Y_\Omega)$ is $p|p\in(R\cup G_\Omega)\wedge p\mapsto\alpha\mapsto\beta\wedge(\beta\mapsto\omega\vee\beta\mapsto\gamma\mapsto\omega)$ for **Π(α)=s-alternation**, **Π(β)=s-concatenation** and **Π(γ)=s-repeat**.

From here on, only reduced and well-defined advanced grammars that do not have left recursion are worked with. This removes certain kinds of ε-cycles (but many other kinds remain) that can exist in the automata built based on the grammar.

3.3.1 Finding of the shortest paths

It is chosen that if there are two ε-paths with the same number of ε-transitions that pass through the states created for two different concatenations in the same alternation, then the shortest path is the one that passes through the states for the concatenation with a lower sequence number in the alternation.

During the parser's generation, the shortest ε-path, if any, from the start state to the final state created for each individual rule is first found. The shortest ε-path, if any, from the start state to the final state created for each individual group is then found.

Based on the found ε-paths for rules and groups, the shortest ε-paths for <u>jumping</u> each individual symbol **ω** located in a given concatenation are found. This is done by searching for the shortest ε-path between states X_{i*2} and X_{i*2+1}, which are created when applying the template in Figure 2 for symbol ω_i in concatenation **k=(s-concatenation,**ω_0**,…,**ω_{n-1}**)**, where **i∈[0..n)**. Based on all ε-paths found so far, found are the ε-paths for each symbol ω_i from the state after ω_i to the final state for the rule (or the group) that uses the alternation that uses **k**.

3.3.2 Reachable advanced symbols

Any symbol $\omega\in H_\Omega$ that is the label of a given transition **t**, is called <u>reachable</u> from state **q**, when there exists at least one sequence of zero or more ε-transitions from **q** to the beginning of **t**. A <u>key state</u> is any state that is a start state (of a rule or group) and any state after a given symbol that is created by applying the pattern in Figure 2.

During the parser's generation, a search is performed for the reachable states from each key state. The search is similar to the popular depth-first search, but with a few differences:

1. The only way a search can jump a symbol (to go through the states and transitions that are created for the symbol) is by traversing the symbol's ε-path, if it exists;

2. The transitions `(d-con-enter,k)` with lower $\mathbf{k}$, which are not part of the chosen ε-path (if any) for the parent of $\mathbf{q}$, are used for the search first;
3. When searching from a given state $\mathbf{q}$, the transition at the chosen ε-path (if any) for the parent of $\mathbf{q}$ is used for the search last.

The result of the search for the reachable symbols from state $\mathbf{q}$ is the reachable tree of $\mathbf{q}$, in which the descendants of each node are ordered in the order they were found. The label of a given leaf in the reachable tree is equal to the label of the transition that was used to find the leaf — the symbol $\omega\in H_\Omega$.

From the construction of advanced grammar automata and the definitions of reachable advanced symbols, it follows recursively that each reachable advanced symbol from a given initial state does not conflict with itself in the following situation:

1. The state is created for a group (or for a rule that is referenced by a reference);
2. The group (or the reference) repeats at least two times;
3. For the group (or the rule), there is a chosen ε-path.

3.3.3 Conflicts

In the advanced grammars, there are character, phrase, and eof symbols, as well as countable repetitions for them. As a consequence, different new types of conflicts between symbols become possible during parsing based on these grammars, which do not occur during the work of other parsing algorithms using other grammars.

If in the reachable tree of state $\mathbf{q}$ for $\omega_a\in H_\Omega$ in leaf's label l_a and symbol $\omega_b\in H_\Omega$ in leaf's label $l_b | l_a\neq l_b$ is true that $\Psi(\omega_a)\cap\Psi(\omega_b)\neq\emptyset$, then it is said that ω_a and ω_b are in conflict from $\mathbf{q}$.

The set $\mathbf{E}$ of leaves in the reachable tree of $\mathbf{q}$ is here considered. During the generation of the parser, all the different conflicts E_i are derived from this set $\mathbf{E}$. Each conflict E_i contains an ordered set of leaves in $\mathbf{E}$, in the order they are found, with advanced symbols as labels, as well as information (shown below) about the particular conflict.

Several conflict types are distinguished and are represented as n-tuples, where the first element is the conflict's type, $l\in L$ for `L={l-character,l-sensitive,l-insensitive,l-universal,l-eof}`, and the last element, $\mathbf{s}$, is an ordered set of the leaves in $\mathbf{E}$ labeled with the advanced symbols that are in conflict:

- `(l-character,f,t,s)` — a character conflict for $f,t\in U$ and $f\leq t$, which contains the ordered set $\mathbf{s}$ from leaves in $\mathbf{E}$ with labels from `(s-character,x,y,m)`, such that $f\leq r\leq t$ for at least one $r|r\in\Lambda(x,y,m)$;
- `(l-sensitive,c,n,s)` — a sensitive conflict for a category $c\in C$, a phrase index `n=SENSITIVE(p)` with a phrase $\mathbf{p}$, and an ordered set $\mathbf{s}$ from leaves in $\mathbf{E}$ with labels from:
  - At least one symbol $(\texttt{s-phrase},c,p_n,\equiv)\in PS_\Omega$;
  - Zero or more symbols $(\texttt{s-phrase},c,q,\approx)\in PI_\Omega$, such that `n=SENSITIVE(j)` for at least one $j\in\Lambda(q,\approx)$;
  - Zero or more symbols $(\texttt{s-phrase},c,-,-)\in PA_\Omega$;
- `(l-insensitive,c,n,s)` — a insensitive conflict for category $c\in C$, a phrase index `n=INSENSITIVE(p)` for a lowercase phrase $\mathbf{p}$, and an ordered set $\mathbf{s}$ from leaves in $\mathbf{E}$ with labels from:
  - At least one symbol $(\texttt{s-phrase},c,q_n,\approx)\in PI_\Omega$;
  - Zero or more symbols $(\texttt{s-phrase},c,-,-)\in PA_\Omega$;

- **(l−universal,c,s)** — a <u>universal conflict</u> for a category **c∈C** and an ordered set **s** from leaves in **E** with labels from at least one symbol **(s−phrase,c,−,−)∈PA**$_{\Omega}$;
- **(l−eof,s)** — <u>eof conflict</u> for an ordered set **s** from leaves in **E** with labels from at least one symbol **(s−eof)∈F**$_{\Omega}$.

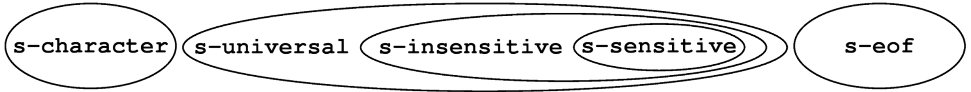

Figure 3: Possible conflicts between advanced symbols

That a given token **h∈H** belongs to a given conflict **E**$_i$ is denoted as **h∈E**$_i$. The cases of a token belonging in conflict are as follows:

- **(t−character,u)∈E**$_i$, if and only if **E**$_i$**=(l−character,f,t,−)** and **f≤u≤t**;
- **(t−sequence,c,e)∈E**$_i$, if and only if:
  - **E**$_i$**=(l−sensitive,c,n,−)** with **n=SENSITIVE(e)**; or
  - **E**$_i$**=(l−insensitive,c,n,−)** with **n=INSENSITIVE(e)**; or
  - **E**$_i$**=(l−universal,c,−)**;
- **(t−eof)∈E**$_i$, if and only if **E**$_i$**=(l−eof,−)**.

3.3.4 Trees with empty nodes

An original syntax tree is <u>ε-condensed</u> when every ε-node in the syntax tree is optimal according to the following optimality criteria:

1. Each ε-node in the tree has as few subnodes as possible, according to the grammar from which it is built;
2. If more than one concatenation in a given alternation can be the basis for the creation of the same number of ε-nodes, then the nodes based on the concatenation with the lower sequence number in the alternation are created;
3. The non-ε-nodes are ordered before the ε-nodes when all these nodes are created based on the same symbol (as a consequence, a possible combinatorial explosion is prevented).

If two different syntax trees built based on the same grammar and for the same input tokens become exactly the same when transformed into ε-condensed form, then they are <u>ε-equivalent</u>. The transformation replaces all ε-nodes with their optimal variant.

3.4 Tunnels

The information contained in each tunnel is about changing the various stacks that the algorithm uses and about the commands that can be sent to the next module. A tunnel is represented as an n-tuple **(t,e,o,d,a)** for the tunnel's <u>type</u> **t∈{τ−first,τ−inner,τ−last,τ−ε−inner,τ−ε−last,τ−ε−direct−front,τ−ε−direct−back,τ−ε−main−front,τ−ε−main−back}**, a sequence of <u>commands</u> **e**, a sequence of <u>operations</u> **o** over **O**, a number of <u>elements to remove</u> **d**$\in\mathbb{N}_0$, which must be removed from the repetition stack, and a number of <u>elements to add</u> **a**$\in\mathbb{N}_0$ (each with a value of one), which must be added to the repetition stack (after **d** number of elements are removed).

During the parser's generation, different types of tunnels are extracted based on the ordered set **s** in a given conflict **E**$_i$, which is an element in a nonempty set of conflicts **E**, which are derived from the leaves of the reachable tree **Z** for key state **q** with traversal of **Z**. For extraction purposes, an extraction stack of segments is created and:

- If **q** is a state for rule **B→α**, then the segment of **B** is added to the extraction stack; or

- If **q** is a state for a group, then the group's segment is added to the extraction stack.

During the extraction of the tunnels, the newly added segments to the extraction stack are only for references or groups. The tunnels for conflict E_i are extracted as follows:

- A tunnel of the **τ-first** type is extracted from **q** to the first element in **s**;
- A tunnel of the **τ-inner** type is extracted between any two adjacent elements in **s** (from one element to the next in the ordered set);
- If no ε-path is chosen from **q** to the final state for the parent of **q**, then a tunnel of the **τ-last** type is extracted from the last element in **s** to **q**;
- If there is a chosen ε-path from **q** to the final state for the parent of **q**, then a tunnel of the **τ-ε-inner** type is extracted from the last element in **s** to the end of the ε-path. For each tunnel of the **τ-ε-inner** type, a tunnel of the **τ-ε-last** type is extracted from the final state for the parent of **q** to **q**.

If, for a key and nonstart state **q**, there is a chosen ε-path to the final state for the parent of **q** and the conflict set for **q** is empty, then a tunnel of the **τ-ε-direct-front** type from **q** to this final state is extracted. In the opposite direction, a tunnel of the **τ-ε-direct-back** type is extracted.

For each rule (or a group) for which an ε-path is chosen, a tunnel of the **τ-ε-main-front** type is extracted from the start to the final state. In the opposite direction, a **τ-ε-main-back** tunnel is extracted.

3.5 Routers

For each key state **q**, an object called a router is created. Routers are intended to contain precomputed information about how, during parsing, the parser can continue the analysis from a given automaton state. A router is represented as an n-tuple (t, p, c_ε) for a router's type **t∈{r-origin, r-minimum, r-inner, r-next}**, an ordered set of paths **p**, and continuation control state c_ε. A path in a router is defined as $E_i \rightarrow c$ for a conflict E_i derived for **q** and a control state (c-state for short) **c**.

The different types of routers are created in the following situations:

- **r-origin** — for each start rule in **S**;
- **r-minimum** — for each symbol $\omega | \omega \in (N_\Omega \cup G_\Omega) \wedge REPEAT(\omega) = (n, m) \wedge n > 1$ (for each reference or group that repeats at least twice);
- **r-inner** — for each symbol $\omega | \omega \in (N_\Omega \cup G_\Omega) \wedge REPEAT(\omega) = (n, m) \wedge n < m \wedge m > 1$ (for each reference or group for which the minimum number of repetitions is less than the maximum number of repetitions and the maximum number of repetitions is more than one);
- **r-next** — for each symbol $\omega | \omega \in (N_\Omega \cup G_\Omega \cup H_\Omega)$ (for all references, groups, and all advanced symbols that directly define tokens).

The paths in the different types of routers are created as follows:

- In **r-origin**, **r-minimum**, and **r-inner**, one path is created per each first leaf in the ordered set of leaves **s** in each conflict E_i that can be derived from the leaves **E** of the rule's reachable tree for the rule's (or group's) start state for which the router was created;
- In **r-next**, one path is created for each first leaf in the ordered set of leaves **s** in each conflict E_i, which can be derived from the leaves **E** of the reachable tree for the state after the symbol for which the router was created.

3.6 Control objects

The TP algorithm uses a set of control objects (c-objects, for short) that use different tunnels and routers. Each control object consists of at least one control state.

The number of control states in a control object depends on the control object. Each control object indicates "where" in the automata the parser reached, and each control state defines "which" operations are to be performed. Each tunnel is executed in a specific context that represents all stacks at a given time.

All the operations that are defined by the different c-states are such that the parser follows exactly the automata that were created for the advanced grammar, always moving along the shortest path from one state to another.

Origin control object

One origin c-object is created for each start rule **J∈S**. Before the parser starts analyzing the input tokens, depending on the selected start rule **J**, the parser adds the created origin c-object (created for J) as the first element in the execution stack. An origin c-object is an n-tuple **(c-origin,{use},r)** for one c-state **use** and a router **r** of the **r-origin** type.

Terminal control object

One terminal c-object is created for each terminal state in the automata. The purpose of this c-object is to increase the execution stack by one element before the parser starts working with the next token. A terminal c-object is an n-tuple **(c-terminal,{use},r)** for one c-state **use** and a router **r** of the **r-next** type.

Token control object

One token c-object is created for each leaf in the ordered set of leaves **s** in each conflict $\mathbf{E_i}$ that can be derived from the leaves **E** of a given reachable tree when the symbol in the leaf's label has a maximum number of repetitions equal to one. A token c-object is an n-tuple **(c-token,{use,used},n,t,τ)** for two c-states (**use** and **used**), a next c-object **n**, a c-object **t** of the **c-terminal** type, and a tunnel **τ** of the **τ-first** or the **τ-inner** type.

Batch control object

The batch c-objects are created similarly to the token c-objects, but when the leaf's label has a maximum number of repetitions greater than one. This c-object has complex functionality because it iterates **ω** both forward and backward without the help of other c-objects. The way the c-states in this object work is that they accept as many **h∈Ψ(ω)** as possible. A batch c-object is an n-tuple **(c-batch,{use,repeat,back,used},n,t,τ)** for the four types of c-states, a next c-object **n**, a c-object **t** of the **c-terminal** type, and a tunnel **τ** of the **τ-first** or the **τ-inner** type.

Epsilon-origin control object

The epsilon-origin c-objects are created for each $\mathbf{c_ε}$ in a router of **r-origin** type, that is created for a given start rule **J∈S**, for which there is a chosen ε-path, because **J⇒*ε**. The purpose of this c-object is to be used when the parser starts the parsing for the start state of **J**, but there is no path in the said router for the first input token. An epsilon-origin c-object is an n-tuple **(c-epsilon-origin,{use,used},τf,τb)** for the two types of c-states, a forward tunnel **τf** of the **τ-ε-main-front** type, and a backward tunnel **τb** of the **τ-ε-main-back** type.

Epsilon-next control object

The epsilon-next c-objects are created for each key state **q** (other than the start of a rule or group), from which there is a chosen ε-path to the final state for **q**'s parent. The only c-state of this type of c-object is used as $\mathbf{c_ε}$ in a router **r** of the **r-next** type, which was created for **q**. Once the

parser is in state **q**, then the parser will search for a path in **r** for the current token. If no path is found, then the single c-state of that object replaces the top of the execution stack. An epsilon-next c-object is an n-tuple **(c−epsilon−next,{use},τf,τb)** for one c-state, a forward tunnel **τf** of the **τ−ε−inner** type, and a backward tunnel **τb** of the **τ−ε−last** type.

Epsilon-fill control object

One epsilon-fill c-object is created for each router of the **r−minimum** type when the rule or the group for which this router is created has an ε-path. The single c-state is used as c_ε in the said router and is never added to the execution stack because it is used by other c-objects. An epsilon-fill c-object is an n-tuple **(c−epsilon−fill,{use},τf,τb)** for one c-state, a forward tunnel **τf** of the **τ−ε−main−front** type and a backward tunnel **τb** of the **τ−ε−main−back** type.

Passage-origin control object

A passage-origin c-object is created at the end of a list of **c−token** or **c−batch** c-objects in a router of the **r−origin** type that is created for a given start rule **J∈S** for which there is a chosen ε-path, because **J⇒*ε**. In order to get to the use of this c-object, the parser has run through all possible ways of parsing the input tokens and progressed backward until it reached the start state for the start rule that has a chosen ε-path. A passage-origin c-object is an n-tuple **(c−passage−origin,{use,used},τf,τb)** for two c-states, a forward tunnel **τf** of the **τ−ε−inner** type, and a backward tunnel **τb** of the **τ−ε−last** type.

Passage-minimum control object

A passage-minimum c-object is created at the end of a list of **c−token** or **c−batch** c-objects in a router **r** (of the **r−minimum** type) that is created for a given start rule **J∈S** for which there is a chosen ε-path, because **J⇒*ε**. In order to get to the use of this c-object, the parser used all **c−token** and **c−batch** c-objects that are arranged in a list of c-objects in the minimum router **r**. A passage-minimum c-object is an n-tuple **(c−passage−minimum,{use},τ)** for one c-state and a tunnel **τ** of the **τ−ε−inner** type.

Passage-next control object

A passage-next c-object is created at the end of the list of c-token or c-batch c-objects in a router of the **r−next** type that is created for a given state **q**, from which there is a chosen ε-path to the final state for the parent of **q**. In order to get to the use of this c-object, the parser has performed all possible ways to parse the input tokens after the given state and progressed backward. A passage-next c-object is an n-tuple **(c−passage−next,{use},τ)** for one c-state and a tunnel **τ** of the **τ−ε−inner** type.

Back-origin control object

A back-origin c-object is created at the end of the list of **c−token** and **c−batch** control objects in a router of type **r−origin** that is created for a given start rule for which no ε-path is chosen. This c-object is similar to the passage-origin c-object with the difference that there is no ε-path selected for the start rule, and for this reason, the parser goes from the state for the last **c−token** or **c−batch** c-object directly to the start state of the start rule. A back-origin c-object is an n-tuple **(c−back−origin,{use},τ)** for one c-state and a tunnel **τ** of the **τ−last** type.

Back-universal control object

A back-universal c-object is created at the end of the list of **c−token** or **c−batch** c-objects in a router of the **r−next** type, which is created for state **q** after symbol $\omega\in(N_\Omega\cup G_\Omega\cup H_\Omega)$, where from **q** there is no chosen ε-path to the final state for the parent of **q** and for $\omega\in(N_\Omega\cup G_\Omega)$ no repetitions are counted or $\omega\in H_\Omega$. When the parser progresses backward and returns to state **q**, then this c-object executes the tunnel from the final state for the parent of **q** to **q**. A back-universal c-

object is an n-tuple **(c-back-universal,{use},τ)** for one c-state and a tunnel **τ** of the **τ-last** type.

Back-countable control object

A back-countable c-object is created at the end of the list of **c-token** or **c-batch** c-objects in router **r** of the **r-next** type, which is created for state **q**, after symbol $\omega \in (N_\Omega \cup G_\Omega)$ with segment **g** from which there is no ε-path to the final state for the parent of **q** and repetitions are counted for **ω**. The functionality of this c-object is similar to that of a back-universal c-object, with the difference that the parser restores one element to the repetition stack and does backward ε-fill if necessary. A back-countable c-object is an n-tuple **(c-back-countable,{use},g,τ)** for one c-state, the said segment **g**, and a tunnel **τ** of the **τ-last** type.

Back-minimum control object

A back-minimum c-object is created at the end of the list of **c-token** or **c-batch** c-objects in an **r-minimum** router that is created for a given rule (or a group) for which no ε-path is chosen. In order to get to the use of this c-object, the parser has performed all possible ways to analyze the input tokens with repetitions (to analyze the minimum number of repetitions) of the given rule or group and is now progressing backward. A back-minimum c-object is an n-tuple **(c-back-minimum,{use},τ)** for one c-state and a tunnel **τ** of the **τ-last** type.

Back-inner control object

A back-inner c-object is created at the end of the list of **c-token** or **c-batch** c-objects in a router of the **r-inner** type that is created for a given reference (or a group) for which no ε-path is chosen. In order to get to the use of this c-object, the parser has performed all possible ways of analyzing the input tokens by a repetition (analyzing more than the minimum number of repetitions) of the given rule (or a group) and is now progressing backward. A back-inner c-object is an n-tuple **(c-back-inner,{use},τ)** for one c-state and a tunnel **τ** of the **τ-last** type.

Unwind control object

Only one global unwind c-object is created in the control layer. During the execution of the operations that are defined by this c-object, the parser archives the depth stack (similar to exiting a function). If the minimum number of repetitions has not been analyzed for the parent of the archived segment, then the parser attempts to repeat the parent. If a repetition cannot be done, then, if possible, the parser performs ε-fill and then tries to continue the parsing after the parent of the segment. If the minimum number of repetitions for the parent of the archived segment has already been analyzed but the maximum number has not, then the parser attempts to repeat the parent, but on failure it does not perform ε-fill. If a repetition is not possible, then the parser tries to continue after the parent of the segment. If continuing past the segment's parent is not possible, the parser will start to progress backward. An unwind c-object is an n-tuple **(c-unwind,{use})** with one c-state.

Restore control object

During the parser's generation, only one global restore c-object is created in the control layer. This c-object attempts to restore one segment at each step and, at the same time, executes a backward tunnel, if necessary, from the final state of the rule (or the group) in which the segment's parent is located to the state after the segment's parent, along with, if necessary, performing a backward ε-fill depending on the number of repetitions of the segment's parent analyzed so far. A restore c-object is an n-tuple **(c-restore,{incomplete,complete})** with two c-states.

3.7 Parsing

Once all the stacks are prepared, all the tunnels are extracted, all the routers and the entire control layer are created, then the parser running the TP algorithm can start parsing input tokens. The parsing starts by placing the origin c-object that is created for the selected start rule **J∈S** and continues until the parser sends a command **(d-done)**. If the parser aims to find only one syntax tree, then parsing ends at the first output of **(d-success,true)**. If the parser aims to parse until it finds the first error in the input tokens, then the parsing ends after the first command **(d-error)** is output.

3.8 Properties of the Tunnel parsing

A parser running with the TP algorithm uses the control objects and their states, the tunnels, and the routers to transition from one internal state to another. According to all of the definitions above:

- The algorithm parses based on any non-left recursive, well-defined, and reduced advanced grammar;
- The memory used by the algorithm is linear to the number of tokens in the worst case;
- The algorithm parses in linear time based on any grammar that can be derived from a deterministic stack automaton, and if there is no recursion in the grammar, the parsing can be performed with a constant amount of memory;
- The parsing time is exponential in the worst case for some inputs that are parsed based on certain grammars. To deal with the exponential parsing time, memoization can be used;
- Some **LL(k>1)** grammars can be parsed with **k-1** token look-ahead by trivial optimization;
- The algorithm can parse deterministically and in linear time based on some ambiguous grammars, and in this case, it outputs the commands to create an ε-condensed tree.

In the TP algorithm, the number of operations performed by the parser at each iterative step is independent of the number of input symbols.

The non-ε-nodes are ordered before the ε-nodes when all of the nodes are created for a given symbol, which is a consequence of how the unwind c-object works, how the tunnels are extracted, and the definitions of the reachable symbols.

3.9 Conclusions

The TP algorithm offers an efficient execution of a push-down automaton with phrase symbols and countable repetitions, which is represented as connected transition diagrams.

In the chapter, it is shown that templates are used to create automata for the rules in an advanced grammar. The analysis can be sensitive or insensitive to the case of the characters in the lexemes. Character ranges are supported directly in the parser grammar. A new subclass of context-free grammars is determined, based on which strings of tokens can be generated, which the TP algorithm can parse in linear time and memory.

4 Profiling of automatically generated parsers

The following practical questions are of interest: how much memory is used by the syntax trees generated by a particular parser, what information they contain, and how long it takes to create and work with them.

In order to avoid the development of a large number of grammars by hand (which can also increase the risk of mechanical errors), a specially designed tool for the purpose of the dissertation is used, and it is called a parser generator profiler (profiler for short). With the help of the profiler, it

becomes possible to perform a large number of tests on different parsing machines, which are generated by different parser generators.

Roughly speaking, a profiler allows the explicit input of one or several grammars or the programmatic output of an aggregation of grammars differing by selected elements. Then, for each of the grammars, the profiler creates executable codes for one or more corresponding parsers using external programs. By running different data through the different parsers, comparative information about the used resources can be obtained. This process is described below in detail.

For the purpose of profiling, template grammars are created that describe one or more grammars in a short form. Each template grammar is represented as a script (a string of characters) that is valid according to a script language specially created for the purpose of the dissertation, and it is called a template grammar language (template language for short). The template grammars are imperatively programmed in the template language.

The grammars in ABNF are directly defined – no additional computations are required to use the grammar. In contrast, template grammars define one or more grammars in a short form, and therefore additional computations (evaluations) are required to derive all of the object grammars that are defined by the given template grammar. The object grammars are grammars for which no evaluation is required.

4.1 Tokens

The lexer module in the parsing machine works like a continuous lexer, as defined above. This continuous lexer outputs a limit token (of **t-limit** type) when the number of characters for which the lexer fails to uniquely determine which rule they belong to is more than $2^{24}-1$, which is more than enough. In Figure 4 the grammar in the lexer's specification is shown.

```
identifier   = ('a'-'z' / 'A'-'Z' / '_')
                 *('a'-'z' / 'A'-'Z' / '_' / '0'-'9')
number       = 1*('0'-'9')
comment      = "//"
```

Figure 4: The lexer grammar in the lexer specification of the template language

4.2 Template grammar language

In the parser grammar, the phrase symbols are used to distinguish the identifiers that are keywords from the others. Keywords in the language are: **rule**, **token**, **template**, **input**, **out**, **define**, **in**, **if**, **then**, and **else**.

A template language allows different templates (similar to functions) to be called at different places in the script. The template call will be executed during an evaluation, and the result of the call will be written directly to the grammar's script at the point of the call.

4.3 Parser generator profiler

The parser generator profiler is a computer program written in C++ that, according to a given template grammar script, first generates a sequence of object grammars and a sequence of input data. The parsing of the script is performed with the TP algorithm. Each object grammar is then automatically translated by the profiler to the form used by various parser generators. These translated grammars are called profile intermediaries. Each profile intermediary becomes an input to the various parser generators, which generate files called profile sources. Each profile source is written in a given programming language and is subsequently compiled with the compiler of the corresponding language with different compilation options (32/64 bits, optimization level, etc.). The compiled files are called profile targets. Each execution of each profile target is called a test. After

the successful completion of all tests, the visualization of the used resources (time and memory) is performed by the user.

```
ECODE PGP_Machine::Run(void) {
  while(true) {
    ECODE ecode = Progress();            // try to do a step
    if (ecode == E_DONE) return E_DONE; // return when done
    if (ecode != E_OK) return ecode;     // return on an error
  }
}
ECODE PGP_Machine::Progress(void) {
  switch(FState) {
    case JS_ERROR: return E_ERROR;   // had an error
    case JS_DONE : return E_DONE;    // already done
    default: if (!Step()){ FState = JS_ERROR; return E_ERROR;}
  }
  return E_OK; // made a progress
}
bool PGP_Machine::Done(void){ FState = JS_DONE; return true;}
```

Figure 5: Functions to control the iteration

The implementation of the language uses iteration [1, p. 43t] for all of its main functionalities, because this allows on the template language to be developed template grammars with a large number of grammar elements. Various iteratively working machines in the profiler inherit the PGP_Machine class, which controls the iteration, as part of that code is shown in Figure 5.

A diagram of the various steps involved in the processing of template grammars that can be executed by the profiler is shown in Figure 6, where a domain is the set of values for the variators in the grammar along with the current template grammar.

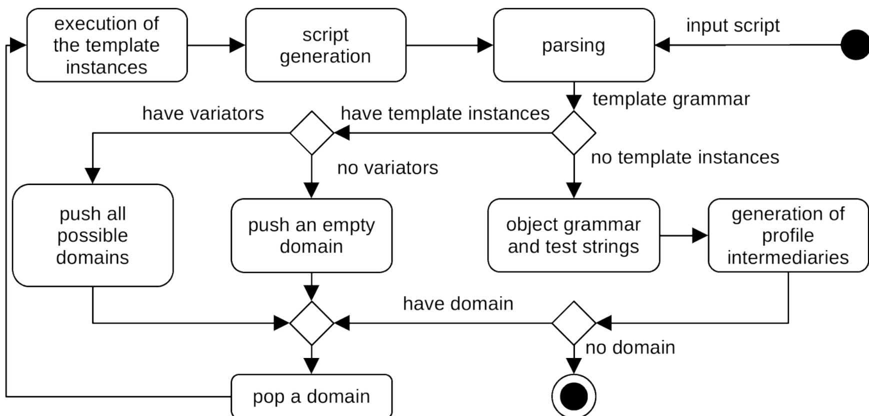

Figure 6: First core functionality of the profiler

4.4 Visualizer

A specially created for the goal utility (visualizer) written in JavaScript, HTML and CSS is used to visualize the test results.

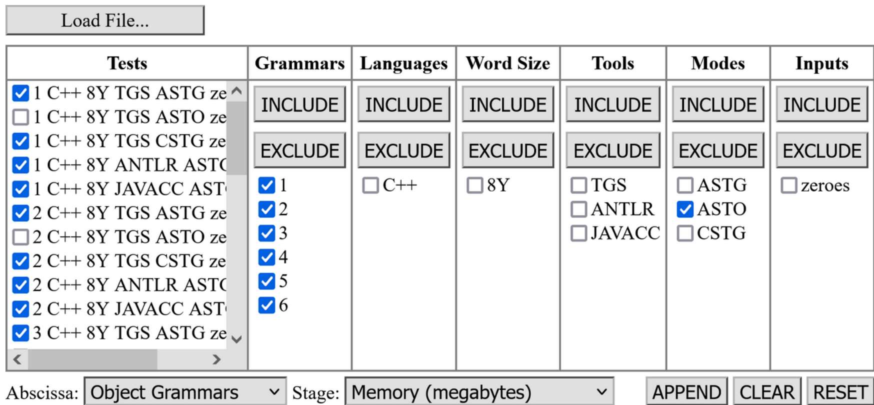

Figure 7: Graphical interface of the visualizer with loaded test results

4.5 Experiments

Measuring the resources that are used by parsing machines at runtime is necessary, because it matters for practical reasons whether the parsing will complete in a second or two [16].

The settings of different parsers, parser generators, and compilers are usually large in number, making it difficult to determine the impact of different combinations of settings on an arbitrary parser that parses based on an arbitrary grammar and analyzes arbitrary data in an arbitrary software environment on arbitrary hardware and is generated and compiled by a specific parser generator and compiler with arbitrary settings. For this reason, experiments are done with as short grammars as possible to eliminate "noise" in the results when there are a large number of different elements in the grammar.

Four experiments are conducted. The design of each experiment aims to measure the resources used (time and memory) by the different parsers generated by different parser generators for different object grammars that are derived from different template grammars.

Experiment 1. Resources during translation

The purpose of the experiment is to give an empirical insight into the effectiveness of different parsers, without taking into account the complexity of the lexer and parser in the parsing machine, and not according to their description in the literature or the documentation of the parser generator, but based on their practical implementations.

Experiment 2. Resources for in depth parsing

The purpose of the experiment is to show how many resources are used for "in depth" parsing and the construction of different types of syntax trees with such a structure.

Experiment 3. Resources for a concatenation

The purpose of the experiment is to show how much the number of elements in a concatenation affects the resources used.

Experiment 4. Resources for skippable elements

The purpose of the experiment is to show the resources used when the grammar has skippable elements that are never found in the input data.

4.6 Conclusions

The fourth chapter presents the design and development of a grammar metaprogramming language in which the template grammars needed to perform the experiments in the dissertation are programmed.

The profiler collects measurements of the resources used during recognition and parsing based on various context-free grammars. The visualization of the collected measurements is done with a special utility program, also created for the purpose of the dissertation.

Conclusion

The dissertation shows lexical and syntactic analyses of data, which are performed by a parsing machine that has a parser module that works with the Tunnel parsing algorithm. A common parsing machine architecture is proposed. The functionalities of the modules in the parsing machine and the objects that are sent and received by the modules are described in detail.

A template language in which template grammars can be defined and/or programmed, and a tool (parser generator profiler) that derives one or more object grammars from a given template grammar are shown. The profiler can experiment with different parsing machines that are generated by different parser generators based on the derived object grammars.

A good property of the iterative parsing is that a pause can be made after each iterative step. Another property of the iterative parsing is that the data the algorithm works with is available in instances of data structures (not in the thread dedicated stack) and can more easily be saved to (restored from) a memory medium when needed.

Contributions

The dissertation contains the following scientific, scientific-applied, and applied contributions:

Scientific contributions:

Н1. A conceptual model of a parsing machine is proposed that is suitable for the implementation of various strategies, methods, approaches, and algorithms, providing some additional capabilities compared to the existing ones;

Н2. Advanced grammars with phrase symbols are defined as composed of rules and advanced symbols, with a close structure to the grammars in augmented Backus-Naur form;

Н3. A phrase machine model is proposed that categorizes in advance the different phrases in the parser grammar in order to speed up the analysis.

Scientific-applied contributions:

НП1. The functionality of the parsing machine is described;

НП2. The functionality of the Tunnel parsing algorithm is described as parsing based on any non-left recursive advanced grammar, with the parsing time being linear for some ambiguous grammars and for any grammar that can be derived from a deterministic push-down automaton (if the derived grammar has no recursion, the parsing can be performed with a constant amount of memory);

НП3. A metaprogramming language for grammars is designed and developed;

НП4. A parser generator profiler is designed, enabling the performance of tests (measuring the resources used during recognition and parsing) with parsing machines created by different parser generators and compilers.

Applied contributions:

П1. A prototype of a software tool – a parser generator profiler (including a module for visualizing the results) is developed, enabling experimentation with a directly entered grammar or with a programmatically derived, by the tool, aggregation of grammars;

П2. Experiments are conducted using the profiler.

Future Work

The dissertation touches on many topics that are related to data parsing, and for this reason there are many possible ways of development, the most notable of which are as follows:

- The list of commands that are sent by the parser module and accepted by subsequent modules can be supplemented with new ones, so that not only syntax trees, but also other syntax structures can be built;
- In connection with the advanced grammars, the possibility can be explored that after using the lexeme in a given sequence token (of the **`t-sequence`** type), the token to be decomposed (in the input data of the specific module) into individual character tokens (of the **`t-character`** type), one for each character in the lexeme, and these tokens to be used for subsequent parsing;
- A natural extension of the current dissertation is the creation of an addition to the Tunnel parsing algorithm that will enable parsing based on left-recursive grammars;
- Based on the dissertation, an algorithm can be created that verifies whether the parsing of arbitrary data with the Tunnel parsing algorithm based on a specific grammar will always be in linear time;
- A possible future improvement of the profiler is adding support for other parser generators and compilers. The currently supported versions of the various parser generators and compilers should not be removed so that the tests can show the development of the technologies related to the automatic generation of parsers over time. This makes the profiler a future-oriented program;
- Another possible development of the profiler is to measure the compilation time of the profile sources and the profile targets, the startup time of the profile targets, and the size of each file.

List of publications on the topic of the dissertation

1. N. Handzhiyski and E. Somova, "Tunnel Parsing with counted repetitions", Journal Computer Science, vol. 21, n. 4, p. 441-462, 2020;
2. N. Handzhiyski and E. Somova, "A parsing machine architecture encapsulating different parsing approaches", International Journal on Information Technologies and Security (IJITS), vol. 13, n. 3, p. 27-38, 2021;
3. N. Handzhiyski and E. Somova, "The Expressive Power of the Statically Typed Concrete Syntax Trees", CEUR Workshop Proceedings, vol. 3061, p. 136-150, 2021;
4. N. Handzhiyski and E. Somova, "Tunnel Parsing with the Token's Lexeme", Journal Cybernetics and Information Technologies, vol. 22, n. 2, p. 125-144, 2022;
5. N. Handzhiyski and E. Somova, "Tunnel Parsing with Ambiguous Grammars", Journal Cybernetics and Information Technologies, vol. 23, n. 2, p. 34-53, 2023;
6. N. Handzhiyski and E. Somova, "Tunnel Parsing", in book "Composability, Comprehensibility and Correctness of Working Software", CEFP 2019, Lecture Notes in Computer Science, vol. 11950, p. 325-343, 2023.

Noted citations

1. N. Handzhiyski and E. Somova, "Tunnel Parsing with the Token's Lexeme", Journal Cybernetics and Information Technologies, vol. 22, n. 2, p. 125-144, 2022:
    - S. A. Qassir, M. T. Gaata, A. T. Sadiq, "SCLang: Graphical Domain-Specific Modeling Language for Stream Cipher", Cybernetics and Information Technologies, vol. 23, n. 2, p. 54-71, 2023.

References

[1] H. Kiskinov, "Introduction to Discrete Mathematics", University of Plovdiv "Paisii Hilendarski" Faculty of Mathematics and Informatics, Plovdiv University Publishing House 2022 (in Bulgarian)

[2] A. V. Aho and J. D. Ullman, "The Theory of Parsing, Translation, and Compiling", Prentice-Hall Inc. 1972.

[3] M. Sipser, "Introduction to the Theory of Computation", 2nd edition, Course Technology 2006.

[4] L. Zheng, S. Ma, Z. Chen and X. Luo, "Ensuring the Correctness of Regular Expressions: A Review", International Journal of Automation and Computing, vol. 18, n. 4, p. 521-535, 2021

[5] D. Crocker and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 5234, 2008.

[6] P. Kyzivat, "Case-Sensitive String Support in ABNF", RFC 7405, 2014, visited at https://www.rfc-editor.org/rfc/rfc7405.html.

[7] N. Chomsky, "On Certain Formal Properties of Grammars", Information and Control, vol. 2, n. 2, p. 137-167, 1959.

[8] A. W. Burks and H. Wang, "The Logic of Automata", Journal of the ACM, vol. 4, n. 2/3, Association for Computing Machinery, p. 193-218/279-297, 1957.

[9] Z. Bednárová and V. Geffert and C. Mereghetti and B. Palano, "Removing nondeterminism in constant height pushdown automata", Information and Computation, vol. 237, p. 257-267, 2014.

[10] Y. Bar-Hillel, M. Perles and E. Shamir, "On Formal Properties of Simple Phrase Structure Grammars", Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung, vol. 14, p. 143-172, 1961.

[11] A. Afroozeh and A. Izmaylova, "One Parser to Rule Them All", in book "2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)", p. 151-170, Association for Computing Machinery 2015.

[12] M. G. J. van den Brand and J. Scheerder and J. J. Vinju and E. Visser, "Disambiguation Filters for Scannerless Generalized LR Parsers", in book "Compiler Construction", p. 143-158, Springer Berlin Heidelberg 2002.

[13] E. R. Van Wyk and A. C. Schwerdfeger, "Context-Aware Scanning for Parsing Extensible Languages", in book "Proceedings of the 6th International Conference on Generative Programming and Component Engineering", p. 63-72, Association for Computing Machinery 2007.

[14] E. Scott and A. Johnstone, "GLL syntax analysers for EBNF grammars", Science of Computer Programming, vol. 166, p. 120-145, 2018.

[15] Unicode® 15.0.0, Unicode Standard, 2022, visited at https://www.unicode.org/versions/Unicode15.0.0/.

[16] B. Dean, "We Analyzed 5.2 Million Desktop and Mobile Pages - Here's What We Learned About Page Speed", Backlinko, 2019, visited at https://backlinko.com/page-speed-stats.