Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2022

MSc THESIS

Hardware-Accelerated PEG Parsing Machine

R.P.M. Bakker

CE-MS-2022-03

Abstract

Information exchange through the countless webservices is central in the current age of technology, which increases the importance for security in these developments. One such security feature is the validation of data based on standardized data structures. The aim of this thesis is to develop a flexible hardware-accelerated text-based recognizer that provides this strict syntax validation.

To this end, a parsing machine architecture was adopted in order to fulfill the flexibility and strict recognition requirements. The parsing machine architecture was developed by formalizing the fundamental PEG expressions and creating a micro-architecture based on these PEG expressions, which led to the specification of the PPEG instruction set architecture. This architecture was then mathematically formalized and a proof for its strict adherence to the formalized PEG behavior was provided. The parsing machine architecture was implemented on an FPGA, a virtualization of the parsing machine was implemented in Python for easy analysis of its behavior, and a PEG compiler and assembler were developed for the PEG-PPEG translation. Finally, a memoization unit was developed as an extension to the parsing machine for an improved parsing throughput.

By running benchmarks for CSV, XML, JSON, and Java files on the PPEG parsing machine implementation, its parsing behavior was analyzed and compared to existing solutions. This showed that the minimum stack sizes depend solely on the size and complexity of the PEG; the percentage of clock cycles spent on jumps in instruction and data memory is substantial, ranging from 18% and 40%; the PPEG-compiled binary code size is relatively small compared to other solutions; and the throughput of the PPEG parsing machine is comparable if not better than other solutions running on faster hardware. Finally, the memoization unit was found to benefit large complex grammars more than small grammars.

Hardware-Accelerated PEG Parsing Machine

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

R.P.M. Bakker
born in Delft, the Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Hardware-Accelerated PEG Parsing Machine

by R.P.M. Bakker

Abstract

Information exchange through the countless webservices is central in the current age of technology, which increases the importance for security in these developments. One such security feature is the validation of data based on standardized data structures. The aim of this thesis is to develop a flexible hardware-accelerated text-based recognizer that provides this strict syntax validation.

To this end, a parsing machine architecture was adopted in order to fulfill the flexibility and strict recognition requirements. The parsing machine architecture was developed by formalizing the fundamental PEG expressions and creating a micro-architecture based on these PEG expressions, which led to the specification of the PPEG instruction set architecture. This architecture was then mathematically formalized and a proof for its strict adherence to the formalized PEG behavior was provided. The parsing machine architecture was implemented on an FPGA, a virtualization of the parsing machine was implemented in Python for easy analysis of its behavior, and a PEG compiler and assembler were developed for the PEG-PPEG translation. Finally, a memoization unit was developed as an extension to the parsing machine for an improved parsing throughput.

By running benchmarks for CSV, XML, JSON, and Java files on the PPEG parsing machine implementation, its parsing behavior was analyzed and compared to existing solutions. This showed that the minimum stack sizes depend solely on the size and complexity of the PEG; the percentage of clock cycles spent on jumps in instruction and data memory is substantial, ranging from 18% and 40%; the PPEG-compiled binary code size is relatively small compared to other solutions; and the throughput of the PPEG parsing machine is comparable if not better than other solutions running on faster hardware. Finally, the memoization unit was found to benefit large complex grammars more than small grammars.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2022-03

Committee Members :

Advisor:	Dr.ir. J.S.S.M. Wong, CE, TU Delft
Chairperson:	Dr.ir. J.S.S.M. Wong, CE, TU Delft
Member:	Dr. C.B. Poulsen, PL, TU Delft
Member:	ir. A.D. Wiersma, Technolution B.V.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AST	Abstract Syntax Tree
BNF	Backus-Naur Form
CFG	Context-Free Grammar
CFL	Context-Free Language
CNF	Chomsky Normal Form
CPU	Central Processing Unit
CSG	Context-Sensitive Grammar
FPGA	Field-Programmable Gate Array
GTDPL	General Top-Down Parsing Language
ISA	Instruction Set Architecture
LIFO	Last-In-First-Out
LL	Left-to-right, Leftmost derivation
LR	Left-to-right, Rightmost derivation
LRU	Least-Recently Used
PEG	Parsing Expression Grammar
PPEG	Parsing machine PEG
TDPL	Top-Down Parsing Language
VPM	Virtual Parsing Machine

Acknowledgements

This thesis marks the completion of my computer engineering studies at TU Delft. The original idea for this project was conceptualized in collaboration with Technolution a year ago from the time of writing, but due to the many interesting possible developments for the project, the process of determining its details and scope was a definite challenge for a long time thereafter. Nevertheless, the opportunity to work on this project as a combination of computer engineering and computer science has been a highlight in my time as a student.

I am extremely grateful for the support I received by many during this project. In particular I would like to thank my supervisors Stephan Wong and Ard Wiersma for their exceptional advice and feedback along the entirety of this journey.

I would also like to thank Technolution for providing the opportunity to work on this topic in a supportive and interesting work environment. Special thanks goes to my colleagues for their support and technical assistance.

Last but not least, I am eternally thankful to my friends and family for their continuous support, patience, and love during my years of study.

R.P.M. Bakker
Delft, The Netherlands
September 23, 2022

Introduction 1

Information exchange through the multitude of today's webservices is central in the current age of technology and interconnectedness and security plays an increasingly important role in its development. Security on the network traffic level is implemented universally in the form of filter applications such as firewalls. These intercept and filter incoming and outgoing network packets based on a set of policies. Using deep packet inspection techniques, the contents of these packets may be inspected and validated at a surface level. However, these applications often implement a pattern matching matching technique, which does not perform exact syntactic and semantic validation of its contents. Moreover, software applications can be slow and are prone to existing exploits of the hardware they run on. Finally, the lack of flexibility has held back the development of hardware-oriented solutions in this space.

This project aims to investigate the possibilities for developing a flexible validation machine for text-oriented data, such as common data structures like XML, JSON, and CSV. Applications for validating or recognizing this type of data based on a set of policies falls under the field of formal grammar and parsing theory, which itself is a field of computer science. Though much research has been done on general parsing algorithms, almost none focus on implementations in hardware and even fewer are easily reconfigurable for different data types.

To this end, this project focuses on the exploration of a new direction in hardware-accelerated recognizers and parsers. Special attention is given to its flexibility. That is, the recognizer must be easily reconfigurable such that different policies may be changed without a complete redesign of the system. Moreover, the design must be extensible in order to potentially enhance performance and add more filter capabilities.

1.1 Problem Statement and Goals

Main Research Question

The main research question that this thesis aims to answer is as follows:

- How can a flexible text-based recognizer be built with digital electronics?

In the context of this thesis, flexibility has two distinct meanings. First, the recognizer needs to be able to be reconfigured for each set of policies without changing the underlying hardware design. Second, minimal changes to the core architecture should be required for adding new recognition functionalities, i.e., optimizations or additional recognition expressivity.

Moreover, this project assumes that the text-based data is ASCII-encoded. However, to facilitate the recognition of raw binary data, the smallest data unit is assumed to be 8 bits.

Additional Research Questions
The list below contains related additional research questions to be answered by this body of work:

- What are the design considerations for choosing a hardware-oriented recognition technique?
- How can performance be enhanced by extending an existing base design?
- How does an implementation in hardware compare to existing software implementations?

Goals
To answer the research questions posed above, the following goals are defined:

1. Define the recognition strategy for a hardware-based recognizer that satisfies flexibility property defined in Section 1.1.
2. Define extensions that enhance the performance of a base design.
3. Implement the defined recognition strategy in hardware.
4. Evaluate the performance of the hardware implementation.

1.2 Methodology

In order to achieve the aforementioned goals, the following steps are taken towards the completion of those goals:

- (Goal 1) Investigate existing implementations in software and hardware and determine their strengths and weaknesses.
- (Goal 1) Select the recognition strategy that best fits the flexibility constraint.
- (Goal 1) Mathematically formalize the selected recognition strategy to allow for a rigid proof of correct operation.
- (Goal 1) Encapsulate the functionality of the envisioned hardware solution in a micro-architecture to allow for extensibility and ease of implementation.
- (Goal 1) Mathematically formalize the functionality of the envisioned hardware solution based on the micro-architecture formalization.
- (Goal 1) Proof that the selected recognition strategy is adhered to in all aspects by the formally defined hardware solution.

- (Goal 1) If time permits, implement a software emulator of the micro-architecture design to perform functional testing and design-space exploration.
- (Goal 2) Investigate possible extensions that enhance performance of the base design.
- (Goal 2) If time permits, implement the aforementioned extensions in the software emulator.
- (Goal 2) Analyze the performance of the design with extensions.
- (Goal 3) Implement the micro-architecture design in an FPGA-based platform.
- (Goal 3) Implement necessary peripherals in the same FPGA-based platform for communication with the implemented recognizer.
- (Goal 4) Define benchmarks that are representative of the use case.
- (Goal 4) Run the benchmarks on the implemented software emulator and design implemented in the FPGA-based platform.
- (Goal 4) Analyze the benchmark results of the hardware design, software emulator, and other similar existing recognition implementations.

Core Hardware Design Principles
Ideally, an engineer has time to iterate through several solutions of a problem in order to ultimately reach a “good” final design. Unfortunately, time does not permit such an extensive iterative design-implementation-measurement process. For this reason, when following the methodology defined in this section, two main design principles are taken into account when designing a first working hardware-accelerated recognizer. These core design principles are as follows:

- Functionality over optimization.
- Simplicity over complexity.

1.3 Thesis Outline

The remainder of this thesis is divided into the five chapters. Chapter 2 presents the necessary background information for understanding design choices and results in subsequent chapters. The end of the chapter contains a detailed analysis of all design considerations, which results in the selection of the best-fit recognition technique which forms the basis of the eventual recognition design. Chapter 3 contains a complete account of the design process for the new hardware-accelerated recognizer. The design is first implemented in software as an emulator, which is part of Chapter 4. This chapter furthermore contains information about possible extensions to the base design and about additional software tools that help development. Chapter 5 analyzes the results that were obtained by running benchmarks on the new design and compares those with existing solutions. Finally, Chapter 6 summarizes the work that was achieved and the results that it produced. Moreover, the main contributions and possible future work are also listed in this chapter.

Background 2

The aim of this chapter is to provide background information on important theory that forms the basis of the rest of this report. Because this work focuses on the analysis of text, the background consists of theory of formal grammars as well as parsing algorithms.

Section 2.1 presents a general explanation of formal grammars and its relation to syntax specification and text generation. Based on these fundamentals, syntax analysis is discussed in Section 2.2 in the form of parsing theory. Here, parse trees and general parsing techniques are the focus. Next, due to a limitation in formal grammars, another type of grammar, namely analytic grammars, are discussed in Section 2.3. Section 2.4 discusses one type of analytic grammar in particular: parsing expression grammars. Because the goal is to create a hardware-oriented recognition implementation, Section 2.5 discusses hardware-accelerated parsers. Finally, Section 2.6 combines the information discussed in previous sections to more precisely focus on a single grammar and parsing technique that fits the requirements as discussed in the Introduction.

This chapter contains more information then strictly necessary for the reader who already has a basic understanding of parsing. Readers already familiar with formal grammars and parsing algorithms are recommended to read Sections 2.4, 2.5, and 2.6.

2.1 Formal Grammars

In the field of formal language theory, a grammar defines syntactic rules that describes how valid strings should be formulated for a given language. More specifically, a formal grammar defines a formal language (valid sets of strings) with finite means, while the language itself may be infinite. Note that no meaning is attributed to components of said language, only their syntax. [6]

Most formal grammars are written in such a way as to define how strings conforming to that grammar must be formed, starting with a so-called "start symbol" from which the formulation process starts. This way of thinking therefore aligns with the concept of a language generator. Depending on the application, however, it might be better to think of a formal grammar as a language recognizer. That is, a grammar can be used to recognize whether a particular string adheres that grammar definition.

A formal grammar G, as defined by Chomsky [7], consists of four components [6] [8]:

1. A finite vocabulary of symbols that appear in strings of the language, also known as terminals. A set of terminals is denoted by Σ.

2. Another finite vocabulary of additional symbols called non-terminals. A set of non-terminals is denoted by N. It holds that no elements of N are present in Σ and vice versa. In mathematical notation: $N \cap \Sigma = \epsilon$, where ϵ is the empty set.

3. A special non-terminal called the start-symbol, which is represented by S, where $S \in N$.

4. And finally a finite set of production rules, denoted by P. In general, these production rules are of the form:

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$$

Note the use of the Kleene closure in the above definition for production rules: a unary operation applied to a set of strings, denoted by V^* with V a set of strings. In this context, it defines a new set containing the empty string and all strings of finite length consisting of one or more arbitrary elements of V. For example, if $V = \{a, b, c\}$, then $V^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, ...\}$. [9]

In total, a grammar is formally defined as a tuple: $G = (\Sigma, N, S, P)$. Moreover, individual elements of the sets contained in this tuple are often represented in literature by the following symbols:

- $a, b, c, ...$ represent terminals.
- $A, B, C, ...$ represent non-terminals.
- $\alpha, \beta, \gamma, ...$ represent strings of terminals and/or non-terminals.

Using these conventions, production rules may be defined such as $\alpha \rightarrow \beta$, which states that string α can be replaced by string β. The application of this rule to a string means that any substring equal to α will be replaced by β. For instance, applying the production rule $\alpha \rightarrow \beta$ to the string $x\alpha y$, produces the new string $x\beta y$, which is formally notated with $x\alpha y \underset{G}{\Longrightarrow} x\beta y$. The operator $\underset{G}{\Longrightarrow}$ thus relates two strings if the second string is obtained when a single production rule defined in grammar G is applied to the first string. If zero or more production rules, defined by the grammar, are applied to one string in order to obtain a second, the $\overset{*}{\underset{G}{\Longrightarrow}}$ operator is used. Similarly, if one or more production rules are applied, the $\overset{+}{\underset{G}{\Longrightarrow}}$ operator is used instead.

Generation of string α, consisting only of terminals in Σ, with grammar G is only possible when starting from start-symbol S, after which a finite sequence of production rules defined in P are applied. The intermediate sequence of strings that originate from S and end with α together form the derivation of α. An example of a derivation of $\alpha = abcdefg$ is as follows.

Given $G = (\Sigma, N, S, P)$, with $\Sigma = \{a, b, c, d, e, f, g\}$, $N = \{S, X, Y, Z\}$,
and $P = \{S \rightarrow XY\ ,\ Xc \rightarrow abc\ ,\ Y \rightarrow Zfg\ ,\ Z \rightarrow de\}$, the generation of α according to $S \overset{*}{\underset{G}{\Longrightarrow}} \alpha$ can be achieved via the following derivation:

$$S \Rightarrow XcY \Rightarrow abcY \Rightarrow abcZfg \Rightarrow abcdefg$$

The complete set of strings that can be generated with grammar G is called the language of G, which is notated as $L(G)$. This can be formally defined as [9]:

$$L(G) = \left\{ w \;\middle|\; w \in \Sigma^* ,\ S \overset{*}{\underset{G}{\Rightarrow}} w \right\} \tag{2.1}$$

In other words, the language $L(G)$ defined by grammar G is the set of all combinations of string w, where w is of finite length consisting only of zero or more arbitrary terminals from Σ_G, and can be derived starting from non-terminal S_G by applying a finite number of production rules from P_G.

2.1.1 Chomsky Hierarchy

In the 1950's, Noam Chomsky introduced formalisms to identify and organize formal languages into four hierarchical levels of complexity [7]. This formulation, called the Chomsky hierarchy, was originally meant to organize natural languages such as English. However, it became apparent that the Chomsky hierarchy could also be used in other fields, like Computer Science, where it is useful to define programming languages with formal grammars as basis, as this can be related to parsing algorithms (see Section 2.2). [6]

The four levels of complexity that were defined by Chomsky are at its core based on the application of increasingly strict constraints to the production rules of formal grammars. From least restrictive to most restrictive we have type-0 grammars to type-3 grammars. Over the years since the introduction of the Chomsky hierarchy, many additional (in-between) grammar types have been proposed [6] [8], but in this section only the original four will be discussed in more detail.

2.1.1.1 Type-0 Grammars

Type-0 grammars, also called recursively enumerable grammars, are completely without constraints on its production rules. As such, the production rules can be defined simply as:

$$P = \left\{ \alpha \to \beta \;\middle|\; \alpha \neq \epsilon \right\} \tag{2.2}$$

From this definition, it is apparent that if a language may be described by any formal grammar, it is also a recursively enumerable language and likewise its grammar can be classified as a recursively enumerable grammar. [6] [9]

2.1.1.2 Type-1 Grammars

The second type of formal grammars is called type-1 grammars. Within this type, two distinct subtypes can be identified, namely monotonic grammars and context-sensitive grammars.

Monotonic grammars, also known as noncontracting grammars, are derived by restricting the type-0 production rules P to the following definition:

$$P = \left\{ \alpha \rightarrow \beta \;\middle|\; \alpha \neq \epsilon\,,\ |\alpha| \leq |\beta| \right\} \tag{2.3}$$

In this context, the notation $|x|$ represents the length of string x or, equivalently, the number of symbols in string x. Monotonic grammars are therefore those grammars whose production rules all have fewer or an equal number of symbols on the left-hand side as on the right-hand side.

The other subtype of type-1 grammars is the so-called context-sensitive grammars, abbreviated by CSG. These have a slightly different set of constraints that apply to their production rules, namely:

$$P = \left\{ \alpha A \beta \rightarrow \alpha \gamma \beta \;\middle|\; A \in N\,,\ \gamma \neq \epsilon \right\} \tag{2.4}$$

In other words, the production rules of a context-sensitive grammar only translate a single non-terminal A into another symbol γ (string of terminals, non-terminals, or both). The type of grammar is aptly named, as it may be noted that the application of the grammar's production rules is dependent on the context. In the case of the above definition, the process of the translation $A \rightarrow \gamma$ is dependent on the context of A, which is α and β on the left and right-hand side of A respectively.

It is worth mentioning that the restrictions imposed on monotonic grammar production rules also implicitly appear in the production rules of CSGs. That is, with regard to production rules, the length of the string on the left-hand side is always equal to or smaller than the length of the string on the right-hand side. CSGs differ from monotonic grammars in that they impose one additional constraint, which is that only a single non-terminal may be rewritten for each production rule. As an example, the following production rule is allowed by both monotonic and context-sensitive grammars:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

But the production rule below is allowed by monotonic grammars only:

$$\alpha A B \beta \rightarrow \alpha \gamma \delta \beta$$

Nevertheless, it can be proven that monotonic grammars and context-sensitive grammars are equally powerful, meaning that each language that can be generated by a monotonic grammar a context-sensitive grammar exists that can express the same language, and vice versa. [8] [9]

2.1.1.3 Type-2 Grammars

As before, the third type of grammars in the Chomsky hierarchy, called type-2 grammars or context-free grammars (CFG), impose even more restrictions on the grammar's production rules. The context-free part of the name of this grammar already gives away the additional restrictions on production rules with respect to context-sensitive grammars. Namely, the production rules of context-free grammars are defined as follows:

$$P = \left\{ A \rightarrow \alpha \,\middle|\, A \in N \,,\ \alpha \neq \epsilon \right\} \tag{2.5}$$

It can be observed that CFGs only allow a single non-terminal A on the left-hand side of production rules, which can be rewritten into a string of terminals and non-terminals shown on the right-hand side. Because no other terminals or non-terminals may surround the non-terminal on the left-hand side of production rules, the process of rewriting a substring by means of applying production rules is not dependent on its surrounding context, thereby being "context-free". A consequence of this is the so-called production independence property of CFGs, which states that whatever a non-terminal is rewritten into is independent of what its neighbors are rewritten into. [8] [9]

The production independence property also benefits the process of recognizing strings as part of a given context-free grammar. This is because a substring can be easily matched to any of the production rules' right-hand side, which produces only a single non-terminal in return without the need for checking the substring's context.

Context-free grammars are often represented in the so-called Backus-Naur Form (BNF). BNF notation compacts multiple alternative production rules for the same non-terminal into a single production rule. For example,

$$\begin{aligned} A &\rightarrow abc \\ A &\rightarrow def \end{aligned}$$

can be written in BNF notation as

$$A \rightarrow abc \mid def$$

Here, the '|' operator signals an unordered choice between multiple possible derivations for the non-terminal on the left-hand side. It must be stressed that this grammar notation does not care about the order of the right-hand side expressions. [10]

2.1.1.4 Type-3 Grammars

The fourth and final type of formal grammars introduced by Chomsky is called type-3 grammars or regular grammars (REG). The most important additional constraint with respect to type-2 grammars is that this type of grammar only allows a single non-terminal on the right-hand side of its production rules. This can be formalized as follows:

$$P = \left\{ A \to a\,,\ A \to aB \;\middle|\; \{A, B\} \in N;\ a \in \Sigma \right\} \tag{2.6}$$

In other words, each production rule in a regular grammar can only rewrite a single non-terminal at a time (left-hand side) into either a single terminal or a single terminal followed by a single non-terminal (right-hand side). Note that the preceding definition only describes right-regular grammars, as opposed to left-regular grammars in which the single non-terminal precedes the single terminal. However, when referred to regular grammars, usually right-regular grammars are intended due to the prevalence of right-regular grammars over left-regular grammars in literature. [8] [11]

Production rules of regular grammars leave out one important property inherent to context-free grammars, thereby making regular grammars easier to convert into parsers (see Section 2.2). This property is the recollection of production rules that came before. An example of this property is shown below. Here, a snippet of a simple context-free grammar is shown. During the process of applying production A, we find ourselves trying to produce output for non-terminal B. Thus, we apply the production rule of non-terminal B. However, when this production rule for B has been successfully applied, we somehow have to recollect our last position in the production rule for A, such that we may continue applying the production rule starting at non-terminal C. By not having to store or recall previous positions in production rules during the process of parsing strings, a simpler parsing algorithm may be used. [8]

$$\begin{aligned} &A \to aBCd \\ &B \to w \end{aligned}$$

2.2 Parsing Theory

Section 2.1 discussed how natural languages were formalized by Chomsky by means of formal grammars. These grammars can be used to generate syntactically valid sentences belonging to the language which the grammar describes. Instead of generating sentences, formal grammars can also be used to recognize or validate whether a given sentence adheres to a particular grammar. Going one step further, analyzing sentences based on a grammar could produce information about how the sentence is constructed from the grammatical elements in the form of a hierarchical data structure. This process is called parsing and the hierarchical data structure that is often used to represent a parsed string based on a formal grammar is called a parse tree.

This section discusses parse trees and how they may be constructed based on a number of different techniques.

2.2.1 Parse Trees

An example of parse trees is as follows. Given the CFG shown in Equation (2.7), we want to parse the string “$a \times (a + a)$”. The result is the parse tree as shown in Figure 2.1. Note that from this tree representation, the derivation order can still be deduced and even the grammar itself may be reproduced.

$$\begin{aligned} &S \rightarrow A \times P \\ &P \rightarrow (B) \\ &A \rightarrow a \\ &B \rightarrow A + A \end{aligned} \tag{2.7}$$

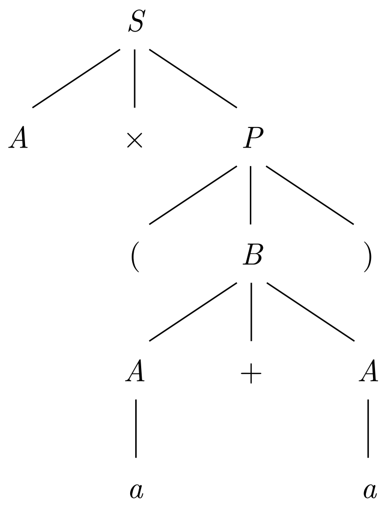
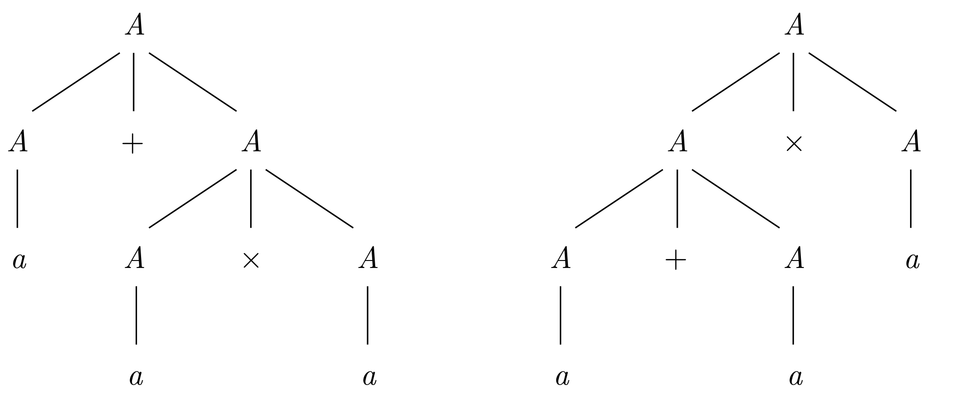


Figure 2.1: Parse tree of input string “$a \times (a + a)$” corresponding to the grammar definition defined in Equation (2.7).

Unlike the previous example, in general there is not necessarily a single parse tree that can be generated for a given grammar and input string. This is exemplified by the grammar shown in Equation (2.8) with which the input string “$a + a \times a$” can be parsed. There are two possible derivations and thus two possible parse trees, as presented in Figure 2.2. The difference between the two derivations is the derivation order. The parse tree in Figure 2.2a is obtained when expanding non-terminals from left to right, which is appropriately called a leftmost derivation. Similarly, expanding non-terminals from right to left is called a rightmost derivation and can be observed in Figure 2.2b. Generally, only a single derivation direction is used in parsers, which would in this case not produce an ambiguous parse. [8] [12]

$$A \to A + A \mid A \times A \mid a \tag{2.8}$$



(a) Leftmost parse tree derivation. (b) Rightmost parse tree derivation.

Figure 2.2: Parse trees of input string "$a{+}a{\times}a$" corresponding to the grammar definition defined in Equation (2.8).

If two or more parse trees can be generated from parsing an arbitrary string with a given grammar, that grammar is said to be ambiguous. Often times, generated parse trees are used to further process some text. Consequently, the generation of only a single parse tree is intended per parse, which warrants the exclusive use of unambiguous grammars.

The problem of unintentional ambiguous grammars can be observed in the infamous "dangling else" construct [12]. Given the context-free grammar below:

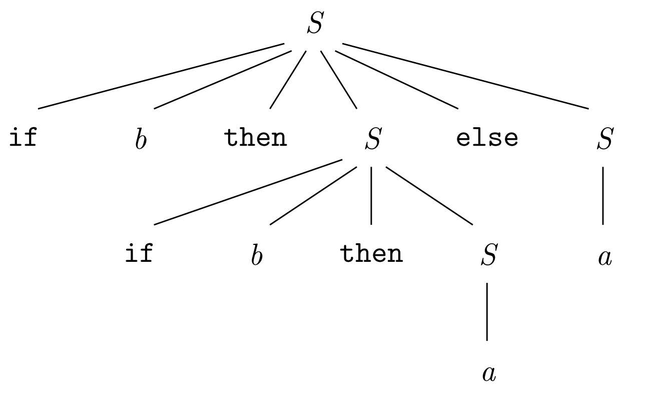
$$S \to \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{if}\ b\ \texttt{then}\ S \mid a \tag{2.9}$$

We want to parse the following text:

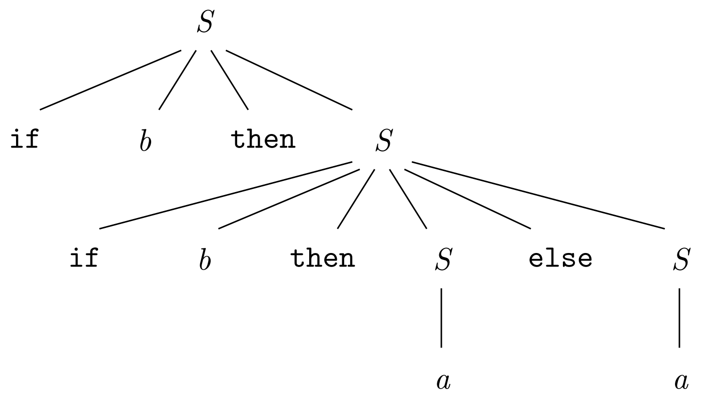
$$\text{“}\texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ a\ \texttt{else}\ a\text{”} \tag{2.10}$$

There are, however, two distinct (leftmost) derivations associated with this CFG, which are represented as the two parse trees shown in Figure 2.3. The difference between the two parse trees is the construct to which the "`else` a"-tail of the input string is bound. In Figure 2.3a the tail is bound to the "outermost" construct / non-terminal S, which is equivalent to the string "`if` b `then` (`if` b `then` a) `else` a". In contrast, in Figure 2.3b the tail is bound to the "innermost" construct / non-terminal S, which is equivalent to the string "`if` b `then` (`if` b `then` a `else` a)".

From this example it becomes clear that a context-free grammar G is ambiguous if there exists a sentence in the set of valid sentences $L(G)$ which can be produced by two or more distinct (left- or rightmost) derivations and thus parse trees. Importantly, it is generally undecidable if a context-free grammar G is ambiguous. [8] [12]



(a) Tail bound to outermost construct.



(b) Tail bound to innermost construct.

Figure 2.3: Parse trees of input string shown in Equation (2.10) corresponding to the grammar definition defined in Equation (2.9).

2.2.2 Parsing Techniques

As previously explained, the job of a parser is to reproduce the derivation of how a sentence is generated given a formal grammar which is ultimately represented as a parse tree. To this end, this section first introduces two basic parsing methodologies: top-down parsing and bottom-up parsing. Thereafter, additional parsing considerations are discussed, such as predictive parsing and memoization techniques.

2.2.2.1 Top-Down Parsing

The first technique follows the same procedure as the example derivation shown in Section 2.1, which started with the start symbol and from there rederived the original sentence by exploring the grammar's production rules. This technique is called top-down parsing, because the parse tree is derived starting at the top (start symbol). [8] [12]

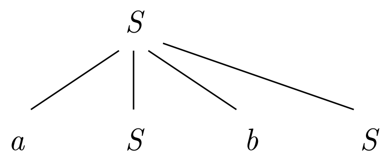
Consider an example of a simple and naive top-down parsing approach, which assumes the following context-free grammar:

$$S \rightarrow aSbS \mid aS \mid c \tag{2.11}$$

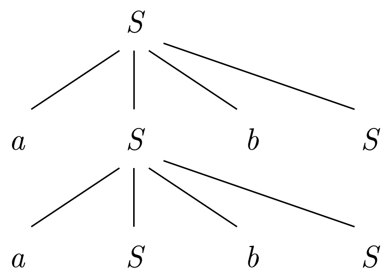
Furthermore, suppose that the input string we would like to parse is "*aacbc*" and that, for this example, the parser always applies the productions for S in the order that they are defined in Equation (2.11). That is, application of non-terminal S leads to first trying to apply the first alternate $aSbS$. If that does not work, try the second aS, and then the third alternate c.

Our top-down parser begins by creating the top node, or root, for the parse tree, namely start symbol S. Moreover, an input pointer is initialized that points to the first character 'a' of the string and follows along the input string during the parsing process. Now, the first production rule for S is invoked, creating nodes for each symbol in the production, thus extending the single-node tree to the new partial parse tree shown in Figure 2.4a. Because the first node of the production $aSbS$ is a terminal, this terminal is directly compared to the first character of the input string. The terminal and character in question match, therefore advancing the input pointer by one character. Next, because the second node of the first production is a non-terminal, the first alternate production of this non-terminal is invoked, thus extending the partial parse tree to that shown in Figure 2.4b. Here, the second character of the input string (as pointed to by the input pointer) is matched to the first node 'a'. Because they match, the input pointer is once again advanced (now pointing to 'c'). The second node is a non-terminal S once more, but note that another invocation of the first alternate would result in a mismatch between terminal a and input character 'c'. The same is true when applying the second alternate, but the third alternate is a correct match between character 'c' and terminal c, as shown in Figure 2.4c, which allows for the advancement of the input pointer to point to character 'b'. This character matches with the next node, namely terminal b. Finally, the top-down parser attempts to match productions for non-terminal S to extend its bottom-right node in Figure 2.4c, which it does by matching character 'c' with the third production rule consisting solely of terminal c. So far, the resulting partial parse tree looks as shown in Figure 2.4d.

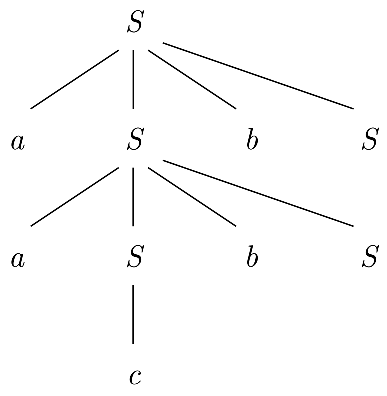
Unfortunately, at this point the input pointer is already all the way at the end of the input string, but the top-right nodes b and S must still be evaluated in Figure 2.4d. Because there is no more input character to match with these nodes, this parse tree is no representation of the input string, and for this reason the parser undoes all steps up until the point that an alternate production can be applied to a non-terminal which has not been applied before. In this case, no alternate productions can be applied correctly until we undo the steps up to the parse tree presented in Figure 2.4a with the input pointer pointing at the second character, 'a'. Instead of applying production rule $aSbS$ for bottom-left non-terminal node S, the parser tries application of production aS, thereby obtaining the partial tree as shown in Figure 2.4e. Here, a match is found, thus advancing the input pointer to character 'c'. This character is correctly matched by applying the third alternate for the next node, non-terminal S, thereby extending the parse tree to that shown in Figure 2.4f and advancing the input pointer to the second-to-last character, 'b'. The parser matches this character with next node in the parse tree, terminal b, and advances the input pointer to once more point at the last character 'c'. The only production that matches with this input character is the third alternate production of S, which is applied to the right-most non-terminal S node in Figure 2.4f. Ultimately, this results in the completed parse tree presented in Figure 2.4g. Because the input pointer has reached the end of the input string and no more parse tree nodes are left unmatched, the top-down parser stops.



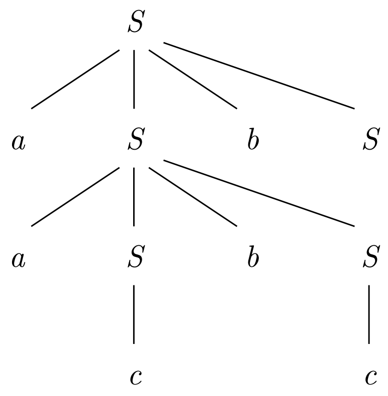
(a)



(b)



(c)



(d)

Notice that the above-described parsing process is exactly the same as the derivation process, if the non-terminals are expanded from left to right. The application of a leftmost derivation is formally defined as:

$$\alpha A \beta \overset{L}{\Longrightarrow} \alpha \gamma \beta$$

With $A \rightarrow \gamma$ a production from the context-free grammar's production rules P, and with $\alpha \in \Sigma^*$. This definition states that a derivation is a leftmost derivation $\overset{L}{\Longrightarrow}$ if A is the leftmost non-terminal in $\alpha A \beta$.

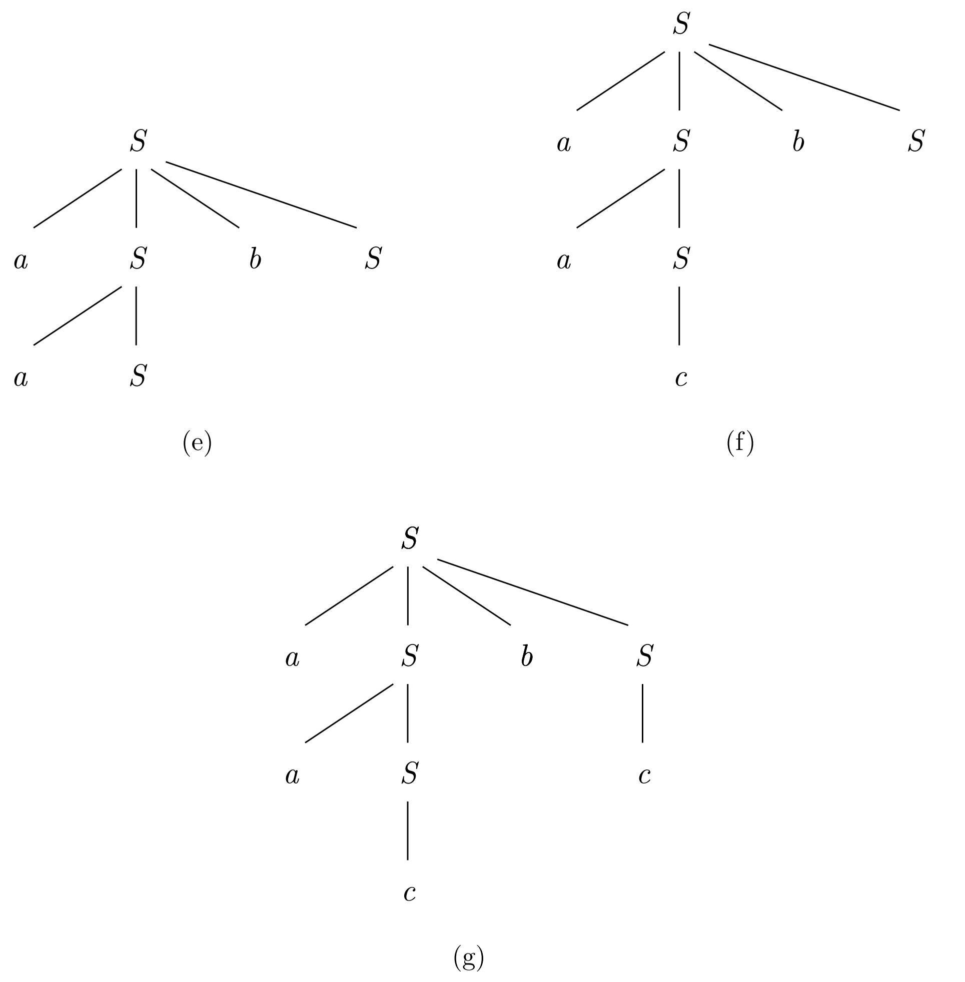


Figure 2.4: Top-down parsing process of input string "*aacbc*" corresponding to the grammar definition defined in Equation (2.11).

In the example top-down parsing process, the parse tree derivation follows the equivalent leftmost derivation process:

$$S \overset{L}{\Longrightarrow} aSbS \overset{L}{\Longrightarrow} aaSbS \overset{L}{\Longrightarrow} aacbS \overset{L}{\Longrightarrow} aacbc$$

In short, the top-down parsing (left-to-right) technique is a process in which the parse tree is derived from the start symbol by recursively deciding which production to apply to the leftmost non-terminal, thus being equivalent to a parse tree derivation by application of left productions. Another common name for these top-down parsers are LL parsers, where the first L stands for the left-to-right reading of the input and the second L stands for the applications of left productions to create the parse tree. [12] [13] [14]

2.2.2.2 Bottom-Up Parsing

The second technique follows a derivation procedure opposite to the top-down parsing algorithm by starting at the leave nodes of the parse tree (i.e., the characters of the input string) and building the tree upwards towards the root of the tree, the start symbol S. Due to the tree building direction, this technique is called bottom-up parsing. [8] [12]

Consider this time an example of a simple and naive bottom-up parsing approach, for which we again assume the context-free grammar presented in Equation (2.11). Furthermore, suppose that we would like to parse the same input string, "*aacbc*". Moreover, this example will explain bottom-up parsing by means of a simple shift-reduce parsing algorithm. That is, at every instant, either the next character is shifted onto a stack or a reduction (reverse of production) is applied to the rightmost input characters on the stack. Note, however, that priority is given to a reduction over a shift if both are possible.

We start by making each input character a separate node of the final parse tree and shift the first input character '*a*' on top of the stack. At this point, since the stack does not match any of the right-hand side productions presented in Equation (2.11), no reduction can be performed. For this reason, the next input character is pushed on the stack, now containing the string "*aa*". Again, there is no reduction possible, thus allowing only a shift of the input character '*c*'. Finally, this allows the parsing algorithm to apply a reduction to the rightmost character on the stack, namely $c \to S$, and a new node is created for non-terminal S as shown in Figure 2.5a. This reduction changes the stack from "*aac*" into "*aaS*". This allows another reduction, namely $aS \to S$. The result is a new tree node S, as shown in Figure 2.5b, and a change in stack value from "*aaS*" to "*aS*". The same reduction can be applied again, finally resulting in the partial parse tree shown in Figure 2.5c and a stack value of "*S*". Note that at this point the remaining characters of the input string are "*bc*". There are no more reductions possible to the stack, so the parser shifts the next input character '*b*' onto the stack. Still no reduction is possible and thus the last character is shifted onto the stack too. Finally, one more reduction, $c \to S$, is possible, changing the stack from "*Sbc*" into "*SbS*".

Unfortunately, no more reductions or shifts are possible and, as the start symbol has still not been reached, no complete parse tree can be derived. Therefore, the parser backtracks to the last reduction, thus returning to the state before the last $c \to S$ reduction. However, as no other reduction is possible, the parser backtracks even further, thereby returning to the partial parse tree as in Figure 2.5b, with a stack containing the characters "*aS*", and the remaining characters in the input string "*bc*". No reduction other than the one applied before ($aS \to S$) can be applied, so the parser shifts the next input character '*b*' onto the stack instead. No reduction is possible on the current stack "*aSb*", so the last character is shifted onto the stack once more resulting in the stack value "*aSbc*". Now, the reduction $c \to S$ can be applied, resulting in the partial parse tree shown in Figure 2.5d. The stack contains the characters "*aSbS*", which is exactly the first alternate production of non-terminal S. Applying the corresponding reduction (i.e., $aSbS \to S$) at last produces the full parse tree as shown in Figure 2.5e. Note that this parse tree is equivalent to the one shown in Figure 2.4g, which was obtained with the top-down parsing algorithm explained in the previous section.

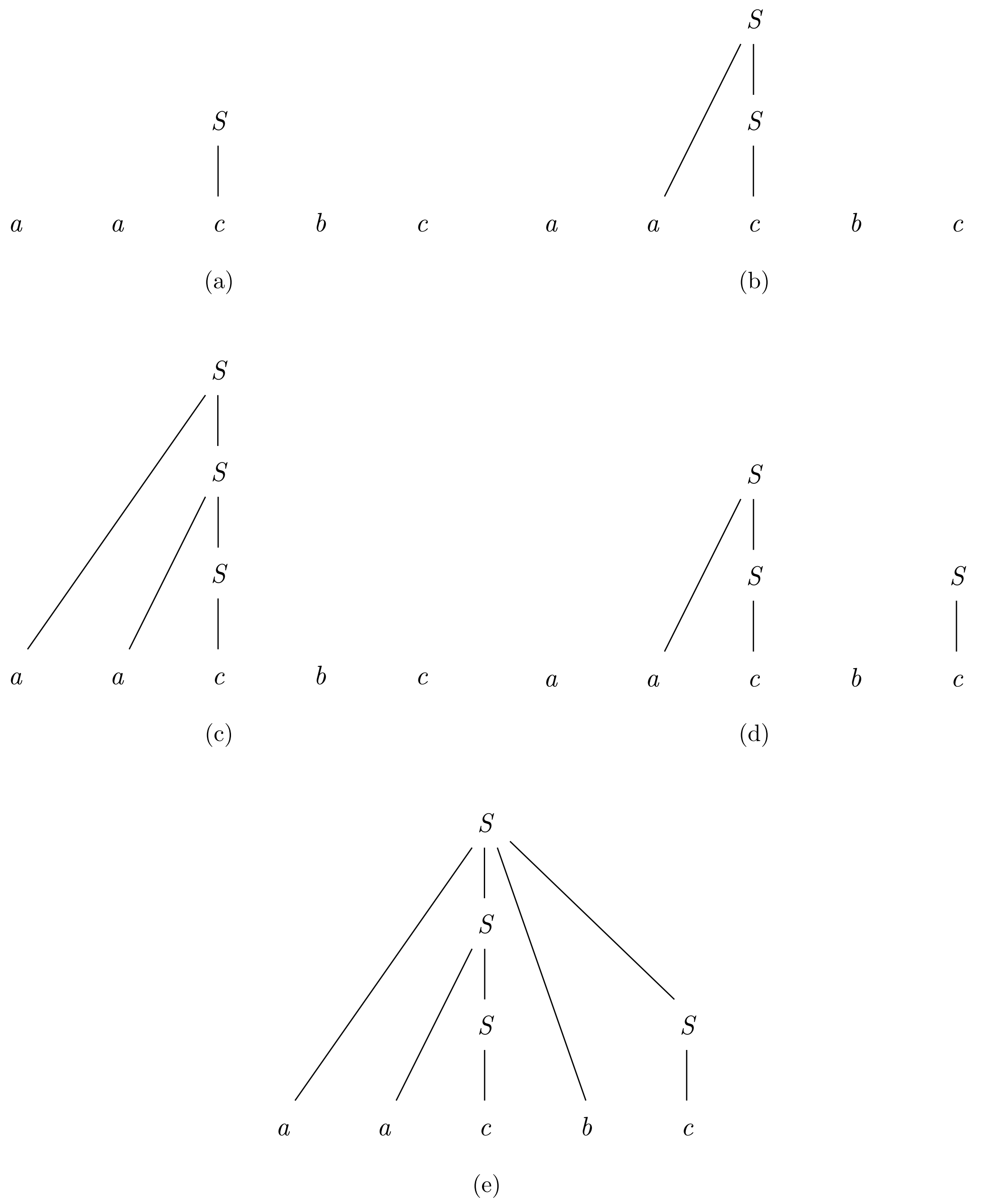


Figure 2.5: Bottom-up parsing process of input string "*aacbc*" corresponding to the grammar definition defined in Equation (2.11).

Notice that the bottom-up process described in this section is exactly the same as the reverse derivation process, assuming that non-terminals are expanded from right to left.

Similar to the leftmost derivation process, application of rightmost derivation is formally defined as:

$$\alpha A \beta \overset{R}{\Longrightarrow} \alpha \gamma \beta$$

With $A \to \gamma$ a production from the context-free grammar's production rules P, and with $\beta \in \Sigma^*$. This definition states that a derivation is a rightmost derivation $\overset{R}{\Longrightarrow}$ if A is the rightmost non-terminal in $\alpha A \beta$.

When applying this formalism to the example provided in this example, the following derivation is obtained:

$$S \overset{R}{\Longrightarrow} aSbS \overset{R}{\Longrightarrow} aSbc \overset{R}{\Longrightarrow} aaSbc \overset{R}{\Longrightarrow} aacbc$$

Reading this derivation from right to left, the steps in this process are equivalent to those in the bottom-up parser description to obtain the parse tree shown in Figure 2.5e.

Summarizing this section, the bottom-up parsing (left-to-right) technique is a process in which the parse tree is derived by applying rightmost reductions according to the grammar's production rules, thus applying exactly the reverse process to a parse tree derivation by application of right productions. Another name for these bottom-up parsers are LR parsers, where the L again stands for the left-to-right reading of the input and the R stands for the applications of reverse right productions to create the parse tree. [13] [12]

2.2.2.3 Predictive Parsing

The examples presented in the previous two parsing methodologies used a naive backtracking approach. The reason for this is that those parsers arbitrarily applied one of multiple non-terminal productions, but then had to backtrack and try an alternate production if the first one failed to produce the correct parse tree. Unfortunately, this backtracking approach results in a worst-case exponential time as a function of the length of the input string. [12] [14]

To reduce backtracking, there are methods that attempt to make deterministic top-down and bottom-up parsers. The idea is to remove the randomness of choosing which production rule to apply and instead look ahead and "predict" which production would match the next few characters. This way, only one correct next step is identified by the parsers at any given time, removing the otherwise non-deterministic behavior.

$$\begin{aligned} S &\to A \mid B \\ A &\to \texttt{if} \\ B &\to \texttt{for} \end{aligned} \tag{2.12}$$

To show how predictive parsing works in practice, consider the context-free grammar shown in Equation (2.12). Using this grammar, we want a top-down parser to parse the string "`for`". The top-down parsing algorithm, as defined in Section 2.2.2.1, would first invoke the production for non-terminal A before needing to backtrack and finally invoke the production of non-terminal B, resulting in a successful parse. However, if this top-down parser would be able to look ahead a single character, it would know that invocation of non-terminal A leads to a failed match with terminal `i` and that invocation of non-terminal B leads to a successful match with terminal `f`. The parser can therefore predict that invocation of non-terminal A is illogical, because it would result in backtracking, and that an invocation of non-terminal B results in successfully parsing at least one character. A top-down parser that can look ahead 1 character would therefore directly invoke non-terminal B.

This 1-character lookahead top-down parser is also called an LL(1) parser. Similarly, a 1-character lookahead bottom-up parser is called an LR(1) parser. This prediction technique could of course be extended to a parser that can look ahead an arbitrary number of characters.

For some grammar it might be necessary to extend the number of lookahead characters to keep the deterministic aspect for the parsing process. For example, the single character lookahead top-down LL(1) parser would still have difficulties if the previous CFG was changed to:

$$\begin{aligned}
&S \rightarrow A \mid B \mid C \\
&A \rightarrow \texttt{if} \\
&B \rightarrow \texttt{for} \\
&C \rightarrow \texttt{foreach}
\end{aligned} \tag{2.13}$$

The original LL(1) parser now cannot "predict" which non-terminal to invoke in order to parse the string "`for`", as both productions for non-terminal B and C start with the same character '`f`'. This could be resolved, however, by a 4-character lookahead top-down parser. In general, a k-character lookahead parser is called an LL(k) or LR(k) parser for top-down and bottom-up parsers respectively.

The need for many-character lookahead parsers can be reduced by means of a lexical analyzer (also known as a lexer). Instead of passing the input string directly to the parser, a lexical analyzer reads the input string first and creates tokens of substrings that match with predefined string patterns before passing those to the parser. As an example, consider the context-free grammar as defined in Equation (2.13). If a lexical analyzer defines the keywords in the CFG as lexical tokens $t_1 = \texttt{if}$, $t_2 = \texttt{for}$, and $t_3 = \texttt{foreach}$, the grammar can then be rewritten to make use of these tokens as follows:

$$\begin{aligned} S &\rightarrow A \mid B \mid C \\ A &\rightarrow t_1 \\ B &\rightarrow t_2 \\ C &\rightarrow t_3 \end{aligned} \tag{2.14}$$

Then, if the input string “`for`” needs to be parsed based on this grammar, the string is first passed to the lexical analyzer which converts “`for`” into the token t_2. This token is finally passed to the parser. Instead of a 1-character lookahead parser, a 1-token lookahead parser can be used to immediately predict that invocation of non-terminal B results in a successful parse. In this case it can be said that the language specified by the lexical analyzer in combination with the token-based context-free grammar can be deterministically parsed by a 1-token lookahead parser. [12] [14]

In order to predict which non-terminal can be applied successfully without invoking said non-terminal, a so-called parse table is constructed for each non-terminal. For a k-character lookahead parser, each parse table contains all possible character sequences (up to k character in length) which can be derived from the associated non-terminal [8] [12]. For example, in the case of a LL(1) parser for the CFG in Equation (2.12), the parse table for S contains sequences ‘`i`’ and ‘`f`’, non-terminal A has only ‘`i`’, and likewise non-terminal B has only ‘`f`’.

The method by which to construct parse tables is by finding the so-called FIRST and FOLLOW sets for each non-terminal in the context-free grammar. However, this process is quite involved and is therefore outside the scope of this chapter. Interested readers are referred to sections 8 and 9 of [8] for a detailed account of deterministic top-down and bottom-up parsers and the creation of parse tables by means of FIRST and FOLLOW sets.

2.2.2.4 Tabular Parsing

As the name implies, tabular parsing methods implement a parsing strategy that makes use of storing intermediate information in a tabular fashion. In this section, a general top-down tabular parsing approach and two well-known general tabular parsing approaches are discussed in short.

Top-Down Tabular Parsing
With the naive top-down parsing algorithm discussed in Section 2.2.2.1 it could happen that a non-terminal production is applied at some position in the input string, which had already been successfully applied some time before. For example, consider the case of the following context-free grammar:

$$\begin{aligned}
S &\to A\ B \mid A\ C \\
A &\to a \\
B &\to b \\
C &\to c
\end{aligned}$$

Then, if the simple string “*ac*” is to be parsed by the top-down parser, it would first attempt to parse the text by applying production $S \to A\ B$. Indeed, application of non-terminal A does indeed parse the first character ‘*a*’, but the parser proceeds to backtrack once it fails in applying non-terminal B. Next, the alternate production for S is applied, i.e., $S \to A\ C$. Notice that this requires the application of non-terminal A once more at exactly the same position in the input string. For this rudimentary example it might not require much additional time, but this situation can obviously occur for non-terminals of any complexity.

Reapplication of a previous successfully applied non-terminal can be resolved by keeping track of all successful non-terminal applications at all input string positions. Then, instead of reapplying that non-terminal, a lookup operation can retrieve the associated partial parse tree from a lookup table. This technique of storing results of computation in a lookup table to save later recomputation is known as memoization. [8] [15]

Consider the previous example, where the initial successful application of non-terminal A may thus be stored in a lookup table. Then, once non-terminal A is encountered once more in the second production for non-terminal S, a lookup operation may be performed based on the current non-terminal and the current position in the input string. This returns the partial parse tree consisting of non-terminal A connected to child node terminal a.

Note that the this lookup table implementation requires $n \times m$ memory elements, with n the length of the input string and m the number of non-terminals in the used context-free grammar.

Cocke-Younger-Kasami Parsing

The Cocke-Younger-Kasami (CYK) parsing algorithm – named after J. Cocke, D. H. Younger, and T. Kasami, who independently developed the algorithm – consists of two stages. During the first stage, the algorithm constructs a parse table that, for all possible substrings of input string s, computes which non-terminal, if any, can derive it. Once this table has been constructed, the algorithm steps through the table, starting at the non-terminal that can derive the complete input string, and constructs the parse tree accordingly. [8] [16]

The process of constructing the CYK parse table starts with finding non-terminals that match with individual characters in the input string, and then moves on to finding derivations for larger and larger substrings. For this reason, the CYK parsing algorithm is classified as a non-directional bottom-up parser.

Although construction of a CYK parse table is possible for a general context-free grammar, the time complexity of such an algorithm is far too great to be of practical use. For this reason, the context-free grammar is often restricted to the so-called Chomsky Normal Form (CNF) grammars. Production rules of CNF grammars may only be of one of two formats: $A \rightarrow BC$; $A \rightarrow a$. [16]

The worst-case time complexity of the CYK parsing method is $O(n^3)$, with n the length of the input string. Moreover, because the CYK parse table needs to store data for each substring, the required number of data elements is in the order of $O(n^2)$ (space complexity). [16]

Earley Parsing

The Earley parsing algorithm – Named after its inventor [17] – is based on another tabular parsing method. Earley recognized that tabular bottom-up parsers (e.g., CYK parsers) that compute derivations for all substrings of an input string s are rather inefficient. This is because many of these derivations could never be derived from the start non-terminal by a general top-down parser, and thus would not be a part of the parse tree. [8] [12] [17]

Earley parsing instead steps through the input string from left to right and applies a bottom-up parsing algorithm by applying any non-terminal production that could possibly match with the next character at each step. However, based on a top-down parsing approach, at each step only non-terminal productions from the previous step are considered, along with new productions that can be derived from those existing ones. Each step produces a list of productions that are considered, which are stored in so-called Earley items. After the last character has been parsed, the algorithm traverses the Earley items from end to start, thereby constructing the parse tree. [17]

The Earley parsing process uses a combination of bottom-up and top-down parsing to more efficiently parse a text, by only analyzing and storing those productions that could possibly help in deriving the complete text. Therefore, Earley described his algorithm as a breadth-first top-down parser with bottom-up recognition. [17]

Space complexity of the Earley parsing algorithm is also $O(n^2)$ and despite the added efficiencies, the Earley parsing still has a worst-case time complexity of $O(n^3)$, with n the length of the input string. However, in contrast to the CYK parsing algorithm, Earley's algorithm has $O(n^2)$ time complexity for unambiguous context-free grammars and even $O(n)$ time complexity for deterministic context-free grammars. It is also worth considering that Earley parsers do not require the grammar to be transformed into Chomsky Normal Form, thereby possibly reducing the number of grammar productions and thus parse time. [8] [17]

2.3 Analytic Grammars

As discussed in Section 2.1, Chomsky formalized natural languages with a hierarchy of formal grammars, such as context-free grammars and regular grammars. These types of formal grammars can be classified as being of a generative type, meaning that they define rules for how to generate syntactically valid strings, which forms the grammar's language $L(G)$. However, a side effect of the expressiveness provided by for example context-free grammars is that there may be multiple derivations for the same sentence. As shown in Section 2.2, grammars with this property are known as ambiguous grammars. Ambiguities in a grammar G become a problem when parsing strings, because for some sentences in the language $L(G)$ there is a non-unique parse tree, which may not be the intention of the grammar's author. This problem is compounded by the fact that it is generally undecidable whether a given context-free grammar contains ambiguities [18]. Handling of ambiguities, also known as disambiguation, is therefore key in parsing applications.

2.3.1 Disambiguation

There are multiple techniques to handle ambiguities in the context of parsing. The most common three disambiguation methods are as follows [14]:

- Let the parser produce all possible parse trees and filter out the best parse tree based on a set of rules or list of illegal parse trees.
- Change the parsing algorithm to deal with the ambiguities at run-time based on a set of disambiguation rules.
- Rewrite the formal grammar to exclude any possibility for encountering ambiguities.

The first of these techniques is very compute intensive, as all possible parsing paths have to be followed to the end. Moreover, the process of matching illegal parse trees can be quite complex as well. These problems become especially troublesome when ambiguities are abundant in the grammar. [14]

The second disambiguation method is to let the parser choose a parsing path based on disambiguation rules during runtime. These disambiguation rules can either be implicit or explicit. Implicit disambiguation rules are directly integrated into the parsing algorithm. In contrast, explicit disambiguation is achieved by annotating production rules of formal grammars with declarative disambiguation rules. This provides the language engineer with more control over parsing paths that are explored by the parser. [14]

A well-known example of both implicit and explicit disambiguation rules can be observed in the parser generator tool called Yacc [19]. With Yacc, one can specify explicit disambiguation rules such as precedence and associativity associated with production rules, while it handles all remaining encounters of ambiguities with a set of implicit disambiguation rules. Similar techniques are also applied by the ANTLR4 parser generator tool [20]. Another example of the use of declarative disambiguation rules can be observed by SDF [21], which includes explicit rules for both lexical and context-free syntax. In general, however, there is no one standard for defining declarative disambiguation rules like there is for formal grammars. Furthermore, because there exists no algorithm to definitively determine if a given CFG is unambiguous, there is no guarantee that a set of disambiguation rules make a certain CFG unambiguous [18].

Finally, one could also rewrite the formal grammar G_1 to one without ambiguities G_2 while accepting the same language: $L(G_1) = L(G_2)$. This might prove to be a difficult exercise for complex grammars containing many ambiguities. [14]

However, the third option may be expanded upon by introducing a new type of grammar. One which contains inherent disambiguation rules, such that it cannot be used to define ambiguous grammars. The first instance of such a grammar was proposed and conceptualized by Gilbert in 1966 when he noticed that the types of formal grammars from the Chomsky hierarchy were difficult to work with when designing parsers [18]. Instead of using ad hoc techniques in parsers to deal with any encountered ambiguities, Gilbert wanted a new type of formal grammar that was specifically targeted for parsing. In other words, this new type of formal grammar is purposely written not in a generative form, but instead with the recognition paradigm in mind. Gilbert termed this type of formal grammars as analytic grammars. [18] [12]

The difference between the two types of grammars is exemplified by the difference in notation. Whereas generative grammars use the $A \rightarrow a$ notation, analytic grammars use the notation $A \leftarrow a$ instead. Notice that the direction of the arrows represent the flow of information. The $\rightarrow$ operator signals an expansion from non-terminal A to a string a of terminals. Conversely, the $\leftarrow$ operator signals a reduction from a string a of terminals to a single non-terminal A. [10]

2.3.2 Examples of Analytic Grammars

This section serves to provide insights into how analytic grammars may differ from Chomsky grammars by discussing several examples along with historical context. One of the most recently introduced analytic grammars is presented in its own section (see Section 2.4), as it is central to the rest of this body of work.

2.3.2.1 TDPL

The top-down parsing language, or TDPL, is an early adaption of the analytic grammar paradigm that was first introduced by Birman and Ullman in 1970 [22]. Only, at the time, this grammar was called TS (TMG recognition scheme), because it was based on the 1965 TMG parser generator [23]. The TS grammar formalism was later refined by Aho and Ullman [12] and renamed to TDPL, which aptly describes the underlying theory of the grammar as it is based on a limited backtracking top-down parsing algorithm.

The TDPL grammar is formally defined by the tuple (Σ, N, S, P) (see Section 2.1). Here, the set of production rules P is formally defined below.

$$P = \left\{ A \leftarrow a\,,\ A \leftarrow BC \mathbin{/} D \;\middle|\; \{A, B, C, D\} \in N\,,\ a \in \{\Sigma, \epsilon, f\} \right\} \tag{2.15}$$

The first production rule in this definition represents the ability to match a string with the specified terminals in Σ. In addition, a includes the empty string ϵ, which allows the unconditional reduction of a to A without matching to anything in particular, and a symbol f representing parsing failure, which unconditionally fails the reduction of a to A.

The second production rule consisting only of non-terminals exemplifies the ability of TDPL to express an ordered choice between a set of right-hand side expressions [22] [12]. The second production rule consisting only of non-terminals exemplifies the deterministic and algorithmic aspect of the TDPL grammar. That is to say, as the name implies, a TDPL grammar essentially describes a deterministic top-down parsing algorithm. The determinism part can be seen in the way that TDPL defines the order of how two possible production rules are evaluated by means of the ordered choice operator '/' shown in this production rule. This is in contrast to a context-free grammar, whose choice operator | does not specify the order of evaluation of possible production rules, which could thus result in ambiguities.

Another important aspect of the TDPL grammar is that it exactly defines how the parsing algorithm must backtrack if a failed match is encountered. As discussed in Section 2.2.2.1, given the right-hand side of a rule $AB \mid C$, a normal top-down parsing algorithm will backtrack to A to look for an alternative parse of A when it fails to evaluate B. Only when no other successful parse is found will non-terminal C be evaluated. In contrast, the parsing algorithm described by TDPL will not re-evaluate A if evaluation of B fails, but will instead directly look at alternative rules, in this case C. This mechanism is called limited backtracking, which is where the parsing algorithm will never look for an alternative parse of a non-terminal if it was successfully evaluated before.

$$\begin{aligned} S &\leftarrow A\ c \\ A &\leftarrow a\ /\ ab \end{aligned}$$

The peculiarity that is limited backtracking can be readily observed from the fact that the above example TDPL grammar cannot be used to successfully parse the string "*abc*". Starting with start symbol S, the algorithm jumps to the definition of non-terminal A. Here, due to the ordered choice operator, first the terminal a will be matched with the first character of the input string "*abc*". This evidently succeeds, and thus the algorithm returns back to non-terminal S, where it will then try to match terminal c with the next character in the input string, namely b. This clearly fails, and thus the process of parsing string "*abc*" fails entirely, as the TDPL algorithm will not re-evaluate non-terminal A using its alternate production rule.

If instead a normal top-down parsing algorithm and an equivalent context-free grammar was employed by replacing '/' with the unordered choice operator '|', the same input string would be successfully parsed. This is because, unlike TDPL, a CFG in combination with a normal top-down parsing algorithm has complete backtracking capabilities. This means that when the parsing process fails trying to match input character b with terminal c in the definition of non-terminal S as shown above, the CFG-based top-down parsing algorithm will backtrack and try to match the alternate definition of non-terminal A, namely ab.

In order to parse the string "*abc*", the rule for non-terminal A in the previous TDPL program needs to be changed to the following:

$$A \leftarrow ab \mathbin{/} a$$

The only change with respect to the definition before is the order of the ordered choice operands. In this case, a match for ab will be checked first, before checking the more general case of only a. This example shows that the more general matching instances should be at the end of the ordered choice operator and more specific instances at the start. [12] [10]

Another example that clearly shows the difference between context-free grammars and TDPL is presented by means of the infamous dangling else construct. Below is the associated CFG in BNF format:

$$S \rightarrow \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{if}\ b\ \texttt{then}\ S \mid a$$

To see why this grammar is ambiguous, the sentence below is presented.

$$\texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ a\ \texttt{else}\ a$$

There are two distinct possibilities for deriving this string. With the CFG as defined above, this string can be derived as:

$$\begin{aligned}
&S \Rightarrow \\
&\texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \Rightarrow \\
&\texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \Rightarrow \\
&\texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ a\ \texttt{else}\ S \Rightarrow \\
&\texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ a\ \texttt{else}\ a
\end{aligned}$$

or as:

$$\begin{aligned}
&S \Rightarrow \\
&\texttt{if}\ b\ \texttt{then}\ S \\
&\texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \Rightarrow \\
&\texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ a\ \texttt{else}\ S \Rightarrow \\
&\texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ a\ \texttt{else}\ a
\end{aligned}$$

As this example proves, there are two possible parse trees that can be derived from this one sentence. Moreover, without any additional disambiguation rules implemented in the grammar or parser, this sentence cannot be parsed deterministically. However, directly converting the grammar to a TDPL equivalent grammar would result in a practical non-ambiguous grammar. In that case, the TDPL grammar would parse the sentence as in the second derivation.

Both of the examples presented in this section show another important property of top-down parsing language grammars, namely the natural support for the longest-match disambiguation policy. That is, TDPL always tries to match as many terminals as possible given any production rule. Due to this property, a TDPL grammar definition as the one below would never correctly parse a string of a's, as the invocation of non-terminal A would always greedily consume all a. When finally no more a's can be consumed, the TDPL algorithm returns to the invocation of S at the start and try to consume one last a. This would unfortunately always fail, as all a's have consumed by the non-terminal A in front of it. [10]

$$S \leftarrow A\ a$$
$$A \leftarrow a\ A \ /\ a$$

2.3.2.2 GTDPL

The generalized top-down parsing language, or GTDPL, is another analytic grammar paradigm adapter that was introduced in the same paper as TDPL by Birman and Ullman [22]. Similar to TDPL, this grammar formalism was called gTS (generalized TMG recognition scheme), but was instead based on another parser generator tool called META II [24]. gTS was later refined by Aho and Ullman [12] and renamed to GTDPL, as it was also designed for a limited backtracking top-down parsing algorithm.

The GTDPL grammar formalism is quite similar to the TDPL grammar discussed before. Below the exact definition of production rules of the GTDPL grammar is presented.

$$P = \Big\{ A \leftarrow a\,,\ A \leftarrow B\,[C, D] \ \Big|\ \{A, B, C, D\} \in N\,,\ a \in \{\Sigma, \epsilon, f\} \Big\} \tag{2.16}$$

The only difference between TDPL and GTDPL is the restriction on the second type of production rule. Invoking A causes non-terminal B to be evaluated first. If B is successfully evaluated, the GTDPL algorithm first tries to evaluate non-terminal C to try and match the remaining input characters that are unconsumed by B. If C is then successfully evaluated, the complete evaluation of A succeeds. On the other hand, if evaluation of non-terminal C fails, the evaluation of A itself fails too. Finally, in the case that B fails to match on the input string, the algorithm instead invokes non-terminal D on the same input string as was tried for B, such that at that point successful evaluation of A depends entirely on the successful evaluation of D.

In short, the invocation of non-terminal D in the TDPL production rule $A \leftarrow BC\ /\ D$ is independent of whether the evaluation of B or C failed, but instead only depends on whether the sequence BC in its entirety failed. In contrast, the invocation of non-terminal D in the GTDPL production rule $A \leftarrow B\,[C, D]$ is dependent only on whether the matching of B failed. This allows different evaluation results of non-terminal A depending on whether B failed, or B succeeded and C failed, which increases the expressive power of the grammar. [22] [12]

The original paper that introduces both TDPL and GTDPL [22] also shows that GTDPL is at least as powerful as TDPL by means of a proof that any TDPL grammar can be rewritten as a GTDPL grammar. Moreover, because of the additional level of expressiveness of GTDPL, it is thought that GTDPL is more powerful than standard TDPL. However, this conjecture has not yet been proven. [10]

Another interesting property of both TDPL and GTDPL grammars is that every deterministic context-free language (a proper subset of unambiguous context-free languages) can be recognized by both TDPL and GTDPL grammars [22]. Moreover, it has been proven that any pushdown automaton can be simulated by TDPL and GTDPL, meaning that they can recognize any $LL(k)$ and $LR(k)$ language. [10]

Finally, although not formally proven, it appears to be the case that the set of languages expressible by CFGs and TDPLs are incomparable. That is, there exist CFLs not expressible by a TDPL grammar and likewise there exist TDPL languages that are not expressible by a CFG. An example of the latter assertion is presented below. This language is expressible by a TDPL grammar, but not by any CFG. [22] [12] For an example of the former assertion refer to Section 6.1.3 of [10].

$$L = \left\{a^n b^n c^n \middle| n \geq 1\right\}$$

2.4 Parsing Expression Grammars

As was mentioned previously in Section 2.3, formal grammars of the generative type were adopted relatively early by programming language engineers, because of its use and research by the field of linguistics. This, despite the fact that a programming language compiler is by definition based on the analysis and not generation of text. Because of this misalignment, various disambiguation techniques are employed by parser generators such as Yacc [19], ANTLR4 [20], and SDF [21]. The recognition parsing paradigm in the form of analytic grammars has only recently become of more frequent research and use. One type of analytic grammar in particular has sparked much interest, namely Parsing Expression Grammars (PEG) introduced by Ford [3].

2.4.1 PEG Definition and Properties

PEG is most similar to TDPL grammars as discussed in Section 2.3.2.1. Parsing Expression Grammars essentially describe a deterministic limited-backtracking top-down parsing method. However, in contrast to both TDPL and GTDPL which are highly limited in their production rule format, PEGs feature a rich expressiveness in their production rules. A more in-depth discussion of PEG, parsing expressions, and its other unique properties are summed up in the subsequent subsections.

2.4.1.1 PEG and Parsing Expressions

Mirroring the notation of formal grammars, a parsing expression grammar is defined by the tuple (Σ, N, S, P), with the set of terminals Σ, the set of non-terminals N, the start symbol $S \in N$, and production rules P. The latter have the format $A \leftarrow e$, where e is called an parsing expression. For any non-terminal in N there exists one parsing expression, such that $A \leftarrow e \in P$. [3]

Parsing expressions are fundamental to PEGs. They can be constructed with a number of operations. Below follows an explanation of these operations and how they are interpreted by PEG [3]. A more formal approach of parsing expressions is discussed later in Section 3.2. Moreover, a PEG that defines the PEG syntax has been defined by Ford in [3] of which a copy can be found in Appendix B.

“s” or ‘s’ – Literal String
The parsing expression “s” represents any literal string, where $\{s_0, s_1, \ldots, s_n\} \in \Sigma$. If the string starting at the current character position matches with s, the characters are consumed and the character position is updated accordingly.

$[s]$ – **Character Class**
The parsing expression $[s]$ represents any character class, where $\{s_0, s_1, \ldots, s_n\} \in \Sigma$. If the character at the current character position matches with any of the character elements $s_0, s_1, \ldots, s_n$, that character is consumed and the character position is incremented by one. s can include both individual characters and character ranges c_1-c_2.

. – **Any Character**
The parsing expression . (full stop or period character) represents any character in set N. As long as the end of the string that is currently parsed has not been reached, consume the next character and increment the character position by one.

(e) – **Grouping**
The parsing expression (e) is simply a grouping of expression e, which is used to give precedence to the evaluation of e before evaluating any expression outside the parentheses, if present.

$e?$ – **Optional**
Parsing expression $e?$ is evaluated by evaluating expression e at the current character position. Though evaluation of expression e can fail, evaluation of expression $e?$ does not. Characters are only consumed and the character position increased if evaluation of e succeeded.

$e*$ – **Zero-Or-More**
Parsing expression $e*$ is evaluated by repeating the evaluation of expression e as many times as possible starting at the current character position. On each successful evaluation, consume the parsed characters and increase the character position.

$e+$ – **One-Or-More**
Similar to expression $e*$, parsing expression $e+$ is evaluated by repeated evaluation of expression e as many times as possible from the current character position. Similarly, on each successful evaluation, consume the parsed characters and increase the character position. However, at least one successful evaluation of e is expected, otherwise evaluation of expression $e+$ fails. Parsing expression $e+$ is equivalent to $e\,e*$.

$\&e$ – **And-Predicate**
Parsing expression $\&e$ is evaluated simply by evaluating expression e. However, on successful evaluation of e, no characters may be consumed and the character position thus must remain unchanged. Failure to evaluate expression e results in a failure to evaluate $\&e$.

$!e$ – **Not-Predicate**
Parsing expression $!e$ is also evaluated by evaluating expression e. If evaluation of e fails, evaluation of $!e$ succeeds, but, similar to expression $\&e$, no characters may be consumed and the character position must remain unchanged. If evaluation of e succeeds, evaluation of $!e$ fails.

e_1e_2 – **Sequence**
As the name suggests, the sequence parsing expression e_1e_2 is evaluated by evaluating first expression e_1 and then, if successful, evaluating expression e_2. Evaluation of e_1e_2 only succeeds if the back-to-back individual evaluations succeed.

e_1 / e_2 – **Prioritized Choice** Parsing expression e_1 / e_2 is evaluated by first trying to evaluate expression e_1. Only if evaluation of e_1 failed at the current character position is expression e_2 evaluated at the same character position. If either succeed, expression e_1 / e_2 succeeds, but if both fail evaluation, e_1 / e_2 as a whole fails as well.

2.4.1.2 Unambiguous Grammars

A core idea of PEG is to remove the possibility to create ambiguous grammars. This is achieved by changing the grammar specification to include disambiguation rules. The result is that a PEG essentially describes how a string can be deterministically parsed with a top-down parsing approach. This is in stark contrast with context-free grammars, which leave out any description about how exactly to interpret a string, but instead how to generate one, thereby possibly introducing ambiguities in the interpretation of strings.

The non-deterministic description of context-free grammars is exemplified by allowing multiple production rules to be defined for a single non-terminal, all with equivalent evaluation priorities. For example, context-free production rules $A \rightarrow a \mid ab$ and $A \rightarrow ab \mid a$ are functionally equivalent, because the BNF production separator '|' does not impose a specific order of evaluation. In contrast, the PEG production rules $A \leftarrow a \,/\, ab$ and $A \leftarrow ab \,/\, a$ are not functionally equivalent, because the prioritized choice operator '/' does impose an order of evaluation, as discussed in Section 2.4.1.1. [3] [10]

2.4.1.3 Lexical Syntax Definition

Expressing the structural syntax of texts is key to context-free grammars and parsing expression grammars alike. However, the latter additionally provides a means to express lexical syntax in a stylistic likeness to regular expressions with Extended Backus-Naur Form (EBNF) notation [25]. The integration of lexical syntax specification tools provides PEGs with much more expressiveness than CFGs. Consequently, where CFGs required a separate lexical analyzer that precedes the syntax analyzer or parser, PEG combines the two.

Character classes, greedy repetition, and predicate expressions can be effectively combined to define lexical syntax. For example, combining character classes and greedy repetition, a PEG rule for integers can be easily constructed as shown below. It consists of a leading non-zero digit followed by a greedy repetition of decimal digits.

```
Integer <- [1-9] [0-9]*
```

An example that shows how repetition and predicate expressions can be combined is shown below. Here, the syntax of an XML tag is defined, which consists of a string of arbitrary length and characters between the begin and end chevrons.

```
XmlTag <- "<" (!">" .)+ ">"
```

Finally, the PEG syntax specification as a PEG in Appendix B show a good example of how larger grammars may readily combine lexical and the hierarchical elements.

2.4.1.4 Localized Backtracking

An important aspect of PEGs are its use of a localized backtracking strategy as opposed to a global backtracking strategy assumed by CFGs. To see the difference, consider the grammar shown in Equation (2.17). Here, a common start non-terminal S is defined, but the production rule for non-terminal A has a CFG implementation and an "equivalent" PEG implementation. Assume furthermore that a top-down parsing approach is employed for both cases.

Consider the CFG-case, which can readily parse the string "*abc*" as follows: first it attempts to evaluate non-terminal A, thereby trying to match with alternate '*a*', which succeeds; having successfully evaluated non-terminal A, it returns to non-terminal S and tries to match terminal '*c*' with character '*b*' which obviously fails; it therefore backtracks to re-evaluate non-terminal A by trying to match the alternate rule "*ab*"; this succeeds, such that evaluation of A succeeds again; finally, terminal '*c*' is successfully matched with character '*c*', thereby completing the parse of the entire string "*abc*".

Now consider the PEG-case, which cannot parse the string "*abc*": it attempts to evaluate non-terminal A the same way as the CFG-case, thus initially succeeding matching with the first alternate '*a*'; it returns to non-terminal S and then fails to match terminal '*c*' with character 'b'. However, unlike the parser based on the CFG, the PEG states that a previous successfully evaluated expression may not be re-evaluated. Therefore, in this case the parser may not backtrack to non-terminal A in order to try alternate "ab", but instead must fail evaluation of non-terminal S entirely.

$$\begin{aligned} & S \leftarrow A\ c \\ \text{CFG: } & A \leftarrow a \mid ab \\ \text{PEG: } & A \leftarrow a \ / \ ab \end{aligned} \tag{2.17}$$

The CFG-versus-PEG example shows how a context-free grammar assumes that an exhaustive search is performed through all production alternatives in order to find an alternate successful parse (global backtracking strategy). In contrast, PEG does not allow such an exhaustive search and only looks for local alternate expressions (localized backtracking strategy), i.e., once e_1 in e_1 / e_2 succeeds, e_2 will never be tried. [10] [26]

The localized backtracking strategy employed by PEG was inherited from the TDPL grammar specification as discussed in Section 2.3.2.1. It also makes a memoization-based PEG parser easier to implement, which is one of the reasons that PEG was originally created to be used by a technique called packrat parsing (see Section 2.4.2.2) [10]. This is one of the major reasons that suggests that the set of languages expressible by PEG are incomparable with those of CFG [10]. That is, there are some languages expressible by PEG and not by CFG, and vice versa.

A final remark on the disadvantage of localized backtracking is that PEGs can exhibit a phenomenon called language hiding [27]. This can occur when an expression e_2 is prevented from ever being evaluated in an expression of the form e_1 / e_2. For example, the expression a / aa could never be used to parse the string "aa", as only the first alternate is ever evaluated.

2.4.2 PEG Parsing Techniques

There are two common parsing techniques for implementing PEG-based parsers. These are the normal naive top-down backtracking approach and a technique called packrat parsing and are explained in more detail in the following sections.

2.4.2.1 Top-Down PEG Parsing

As stated earlier, PEGs essentially describe a deterministic top-down parsing approach, which makes implementing a top-down parser for PEGs relatively straightforward. At its core, the top-down parsing approach for PEGs is identical to that of CFGs as discussed in Section 2.2.2.1. The exception to this is the ability to evaluate PEG-specific parsing expressions and to implement localized backtracking rather than global backtracking (see 2.4.1.4). Unlike CFGs, which only have the ability to express literal strings as terminals, sequences of terminals and non-terminals, and (non-prioritized) choices, PEG parsers additionally need to support the evaluation of character classes, repetitions, and syntactic predicates. Section 2.4.1.1 describes a general top-down parsing approach for each parsing expression.

2.4.2.2 Packrat Parsing

Packrat parsing is a tabular top-down parsing algorithm developed specifically for TDPL and PEG grammars [10]. Its advantage over a regular recursive-descent top-down parsing approach is that it guarantees linear-time parses, but at the cost of linear space complexity $O(n)$, where n length of the input string.

Packrat parsing achieves linear-time parses by employing memoization to its extreme. As discussed in Section 2.2.2.4, memoization can be used to prevent the re-evaluation of previous successfully evaluated non-terminals at a specific character position. In short, to achieve this, a table of $m \times n$ cells are needed, with m the number of non-terminals in the grammar and n the number of characters in the input string. During the top-down parsing process, whenever a non-terminal is correctly evaluated starting at some character position, the cell corresponding to those unique parameters (i.e., non-terminal identifier and character position) is filled with the parse result. At the same time, before evaluating a non-terminal, the cell corresponding to that non-terminal and current character position is checked: if empty, the non-terminal is evaluated as normal; otherwise, the parse result stored in the cell is used to skip re-evaluation of the non-terminal.

For context-free grammars the above approach poses a problem as ambiguous evaluations are a real possibility. Therefore, in the case of ambiguous grammars, the parse result for a non-terminal and character position pair can have a non-unique value, which conflicts with memoization table having a single unique cell per non-terminal and character position pair. PEGs, however, do not have such a limitation, as the grammar is always unambiguous and therefore guarantees that packrat parsing always generates unique parse results for each cell. [10]

It has to be emphasized that linear space complexity of packrat parsing can be a real problem, especially with limited memory environments. The actual storage requirements are proportional to both the string length n and the number of PEG non-terminals m, which is in contrast to the memory requirements of LL and LR parsers, which simply grows with the non-terminal call stack depth. Packrat parsing thus proves great for smaller texts when memory is limited, but is rather inefficient at parsing large quantities of flat texts, such as is common with data structures (e.g., XML, JSON, YAML, CSV, etc.). [10] [28]

2.5 Hardware-Accelerated Parsers

Because the aim of this project is to develop a hardware-accelerated parser based on grammars, it is worth exploring existing hardware-oriented parsing solutions. The following sections summarize the most important existing research in this field. Note that most of the development on parsing theory is done in software rather than hardware, though especially simple recognizers are popular among high-throughput data streaming applications where such designs are used for data-validation and security applications.

2.5.1 Pattern Matching Engines

Many of the hardware-accelerated parsers are not true parsers, but rather pattern matching engines that try to superficially verify compliance of a character stream to a (context-free) grammar or otherwise extract some semantic information from a character stream. This type of hardware development is usually aimed at high-speed data processing for deep packet extraction or similar applications.

For example, Cho et al. [29] implemented a hardware-accelerated pattern matching engine based on context-free grammars. The intention of the paper was to develop a hardware component that can detect and extract semantic information from incoming data streams. Rather than basing the pattern matching engine on regular expressions, CFGs were used to deal with more complex syntactic structures. However, it simply converts the CFG to a state machine, moving from state to state based on characters in the input stream in order to determine when in reached text that needs to be extracted. It therefore does not verify if the stream complies with the context-free grammar, but rather assumes it complies to begin with. A similar hardware implementation was developed by Moscola et al. [30], but their pattern matching engine is based on regular expressions to extract semantic information.

The hardware-based recognizers discussed thus far are fixed for a particular grammar, and cannot be reprogrammed. In contrast, a recognizer design, called the B-FSM, by Lunteren et al. [31] is actually reprogrammable and is based on a state machine. The B-FSM architecture consists of a fixed control unit that moves through a reprogrammable states based on their specified input conditions and corresponding state transitions. Although it is not able to exactly parse strings based on context-free grammars, the programmability of the machine allows for an easily scalable recognizer solution.

2.5.2 Tabular Parsers

Tabular parsing approaches discussed in Section 2.2.2.4 are well-suited for parallelization and thus for hardware-acceleration.

For example, multiple versions of an FPGA-based implementation of the CYK parsing algorithm have been implemented by Ciressan et al. [32] [33]. Their implementations feature a 1D-array of processors, which reduces the worst-case time complexity from $O(n^3)$ (see Section 2.2.2.4) to $O(n^2)$, with n the length of the input string. 2D-parsing arrays do exist [34] and provide a constant parse time of $O(n)$, but such implementations are even more difficult to scale for larger grammars and inputs. Unfortunately, as is the case for any implementation of the CYK parsing algorithm, a space complexity of $O(n^2)$ remains.

There have also been developments in hardware-oriented tabular parsing methods based on Earley's parsing algorithm. A major contributor in this setting was Chiang et al. [35], who developed a parallel Earley algorithm that can be implemented as a 2D-array VLSI architecture. This particular design allows for $O(n)$ parse time complexity as opposed to he original $O(n^3)$ complexity for the original algorithm (see Section 2.2.2.4. However, as with the CYK-based hardware implementation, the $O(n^2)$ space complexity makes it difficult to scale in practice.

2.5.3 Virtual Machines

There also exist a number of virtual machine implementations that implement some parsing algorithm. These virtual machines resemble conventional computer architectures, consisting of instruction and data memory, a decoder and control unit, etc. The instructions are generally specific to a certain parsing algorithm and a grammar is converted to a program consisting of said parsing instructions.

Šaikūnas [36] presents one such virtual machine, which is based on the Earley parsing algorithm. The so-called Earley Virtual Machine (EVM) consists of 6 basic instructions, with additional instructions that can be added for more functionality outside normal Earley parsing. A translation process is used to convert a grammar to a set of these instructions, which can be sequentially executed by the EVM.

There exists also virtual machine implementations for executing regular expressions, such as that presented by Cox [37]. This regular expression virtual machine supports execution of 4 simple instructions, but is able to execute quite the subset of regular expressions.

Finally, there are a number of PEG parsers based on virtual machines, such as MiniNez [4], GPEG [5], and LPEG [38]. Of these, the most influential one is the virtual machine implemented in the Lua programming language by Medeiros et al. called LPEG [38] [39]. The most basic version of the virtual machine consists of 8 instructions, with additional instructions for some optimizations.

Note that the virtual machines discussed here are emulations of an equivalent hardware-based machine, but are themselves implemented in software. However, for some, if not all, implementation of the core in digital hardware might be feasible.

2.6 Design Considerations

From this chapter, it can be observed that there are many options for the implementation of a text-based recognizer. In this section, based on the information detailed in the other sections of this chapter, a single parsing technique is selected that will form the basis for a new hardware-oriented parsing architecture that is explored in the rest of this report. The following paragraphs explain the design considerations that ultimately lead to the selection of a single parsing technique.

Grammar-Based vs. Handwritten Parsers
In general, parsers can either be written by hand or generated based on a formal grammar. The first provides much more control to the author of the handwritten parser, and can be more efficient than automatically generated parsers. However, the latter provides

a formalized and deterministic method for generating parsers based on an easy-to-write and high-level description of a language to be parsed.

The use case for this project specifies that the recognizer must be reconfigurable with a set of policies. This can be achieved by means of a grammar, such that policies are synonymous with grammar production rules. In contrast, having to write a completely new parser for each possible set of policies is rather inefficient. For this reason, a grammar-based implementation is warranted.

Grammar Selection

There are a variety of grammars that are continued to be researched and which allow the generation of parsers. First, consider the set of Chomsky formal grammars discussed in Section 2.1. Regular grammars and regular expressions do not provide the necessary means for syntax analysis. On the other end of the spectrum, recursively enumerable grammars and context-sensitive grammars are too expressive and are therefore difficult to translate into parsers. That leaves context-free grammars (CFG), which are able to express hierarchical syntax and, to some extent, lexical syntax. Moreover, there is a lot of research about context-free grammars with respect to parsing theory. The other type of grammars that were discussed in Section 2.3 is analytic grammars, the most prominent of which is parsing expression grammars (PEG) which is described in detail in Section 2.4.

Though incomparable, CFG and PEG look similar and the set of languages they can express overlaps greatly. A great advantage of CFG is that it has been the subject of research for much longer and has been extensively used for generating grammars (see Section 2.2). Moreover, PEG can be rather unintuitive due to a combination of localized backtracking and repetition operators (see Section 2.4.1.4). However, one advantage of PEG is its inherent disambiguation techniques as opposed to the undecidable ambiguous nature of CFG (see Section 2.4.1.2). The use of PEG removes the need for ad hoc explicit disambiguation rules and possibly ill-specified implicit disambiguation rules. Secondly, PEG features a set of operators, such as repetition and syntactic predicates, which makes the lexical syntax specification much simpler than it would be with CFG (see Section 2.4.1.3). Moreover, if the parser is built to support these operators, no external lexical analyzer component is needed. This in turn reduces the parser architecture greatly, as otherwise two separate parsing components need to be designed along with their communication interface.

For the reasons specified in this paragraph, PEG is chosen as the basis for the eventual parsing architecture.

Parsing Technique Selection

Finally, an appropriate parsing technique must be selected that satisfies the goals defined in Section 1.1. Section 2.5 details three main types of hardware-accelerated parsers.

First, pattern matching engines, even those based on grammars, are not sufficient for true syntax analysis. They are extremely fast, often being able to extract semantic information in real-time from data streams, but parse a superset of the valid sentences specified by a grammar.

Second are hardware-based tabular parsers. These too are generally relatively fast due to parallelization and validate input exactly as the grammar specified. However, this is all at the cost of at minimum linear space complexity. Moreover, tabular parsers are quite rigid and therefore not well-suited for potential extensions.

Lastly, there are the parsers implemented as virtual machines. Though non of the virtual machines discussed before have been implemented in hardware, there is a definite potential for developing such a machine in hardware by designing it based on what is and what is not possible in hardware. Moreover, due to the similarities between such machines and conventional computer architectures, similar tools and hardware extensions could be developed. Finally, the requirement to reconfigure the parser without a new design is inherently satisfied by the use of instructions which can be stored in common memory hierarchies.

Based on the arguments in the previous paragraphs, a parser implementation based on virtual machines is chosen. However, rather than being virtual, the implementation explored by this work is in hardware and is referred to as a parsing machine. Though there are multiple parsing algorithms that may be employed by a parsing machine (e.g., top-down, bottom-up, Earley, etc.), PEG's inherent limited backtracking top-down parsing algorithm is chosen for this new parsing machine design. The reason for this is because this top-down parsing algorithm has been studied extensively and has been used for several other virtual parsing machines.

2.7 Conclusion

This chapter explained the information required to get a good understanding of the fundamentals on which the rest of this report builds forth. It walked through key elements of formal grammars, which are used to define the syntax of a language with a varying level of constraints and expressiveness (see Section 2.1). Context-free grammars especially strike a good balance between the imposed constraints and its expressiveness, thereby being often used by parser generators (see Section 2.2).

Unfortunately, the expressiveness of a context-free grammar still poses a problem when two or more correct parses exist for any sentence in its associated language. For this reason, analytic grammars might prove a better type of grammar, as it cannot define ambiguous grammars (see Section 2.3). The most prominent analytic grammar is PEG (parsing expression grammar), which additionally has the ability to define lexical syntax and, unlike context-free grammars, is based on a limited backtracking top-down parsing algorithm (see Section 2.4).

Though much research has been carried out for parsing techniques that are implemented in software, few have been studied for implementation in hardware, namely: pattern matching engines, tabular parsers, and virtual machines. However, the first does not perform exact syntax validation and the second requires memory sizes proportional to the input string length. Only virtual machines, which model a conventional computer architecture, has the potential to satisfy the required goals (see Section 2.5).

By studying existing grammars, parsing techniques, and hardware-oriented implementations, a combination of grammar and parsing technique was selected based on a best fit with regard to the goals listed in Section 1.1. This lead to the decision to base the recognizer design on parsing machines that implement a limited backtracking top-down parsing approach based on parsing expression grammars (see Section 2.6).

PPEG Architecture Design 3

Based on the design considerations discussed in Section 2.6, this chapter aims to design a new parsing machine architecture that can syntactically analyze text based on parsing expression grammars, but which is specifically optimized for implementation with digital logic. The resulting design is called the "Parsing machine PEG", or simply PPEG, architecture.

Section 3.1 first introduces the reader to fundamental parsing machine concepts and components required to implement the top-down PEG parsing algorithm. Next, Section 3.2 provides a formalization for evaluating PEG expressions. This is followed by an architectural description of the PPEG parsing machine in terms of microcode as presented in Section 3.3. The microcode description is used in Section 3.4 to derive 13 PPEG instructions with which any PEG expression may be evaluated. Finally, Section 3.5 explains how PEG expressions translate to PPEG code. This section also includes proofs that the PPEG parsing machine behaves exactly as defined by the PEG formalization defined earlier in Section 3.2.

The aim of this chapter is not merely to show the finalized architecture and simply explain how it works, but rather to walk the reader through the complete design process with step-by-step explanations of architectural details. However, if only the finalized PPEG parsing machine architecture is of interest, the reader is suggested to read Section 3.1 and Section 3.4 in order to get an understanding of its design.

3.1 Parsing Machine Components

One of the advantages of using a parsing machine (see Section 2.5.3) to implement the limited backtracking top-down PEG parsing algorithm is that it has many parallels with the conventional Von Neumann computer architecture [40]. For example, the function of a parsing machine is not hard-wired, but instead is dependent on an externally stored set of instructions and data.

With conventional computer architecture terminology in mind, the components of the PPEG parsing machine architecture developed in this project are discussed in more detail in this section.

3.1.1 Memory Components

The memory architecture of the PPEG parsing machine is more akin to a Harvard computer architecture in that it separates instruction memory and data memory instead of having one unified memory component. This solves the so-called Von Neumann bottleneck, which is when a single shared memory component is used to store both instructions and data, such that either an instruction or data can be read at any one time [41].

The following subsections describe the instruction memory and data memory components in more detail.

3.1.1.1 Instruction Memory

The instruction memory component contains a single parsing machine program consisting of a set of instructions. In a parsing machine, a program is directly associated with a single grammar definition. The instructions in such a program tell the parsing machine what operations must be performed with respect to the input string in order to parse or recognize it as specified by the grammar.

The instruction memory component has two separate address and data buses: one for reading and one for writing. The read address bus sets the memory location whose contents (a single parsing machine instruction) is put on the read data bus and feeds into the control unit (see Section 3.1.3). The value on this address bus therefore determines what instruction is currently executing on the parsing machine. The write address bus sets the memory location whose contents are overwritten by the value on the write data bus. This pair of buses is used to write a new parsing machine program to memory. Note that the write address and write data buses are off-limits by the parsing machine itself, as it may otherwise change its own program behavior. From the parsing machine point of view, instruction memory is therefore read-only.

3.1.1.2 Data Memory

The data memory component in the PPEG parsing machine architecture only stores the input string that needs to be parsed by the parsing machine. It stores the string in order, starting at data memory address zero. Moreover, the string characters are stored as 8-bit words. This facilitates the 7-bit ASCII encoding that is assumed for each character (see assumptions in Section 1.1), but also allows any raw binary string composed of 8-bit bytes to be stored for parsing.

The interface to the data memory component is constructed exactly as in the instruction memory component, save for the data bus width. The read address bus sets the memory location whose contents (a single 8-bit word) is put on the read data bus. The read address bus value is therefore equal to the character position value and the read data bus value is the character corresponding to that position. The write address bus sets the memory location whose contents are overwritten by the value on the write data bus. Similar to the instruction memory component, this bus pair is used to write a new string to memory. Again, this cannot be under the control of the parsing machine itself, as then it may change the input string during the parsing process. Data memory is therefore also read-only and can only be written to externally.

3.1.2 Stack Components

There are two important fundamental actions that must be supported by a parsing machine, namely non-terminal invocation and (limited) backtracking [13]. This is achieved by the use of a return stack and backtrack stack respectively. These components are discussed in the following subsections.

3.1.2.1 Return Stack

Because a parsing machine uses a top-down parsing approach for implementing parsing expression grammars, support for non-terminal invocation (i.e., applying production rules associated with the non-terminal) needs to be added. In many respects, there are definite parallels between the call to a non-terminal and the call to a function.

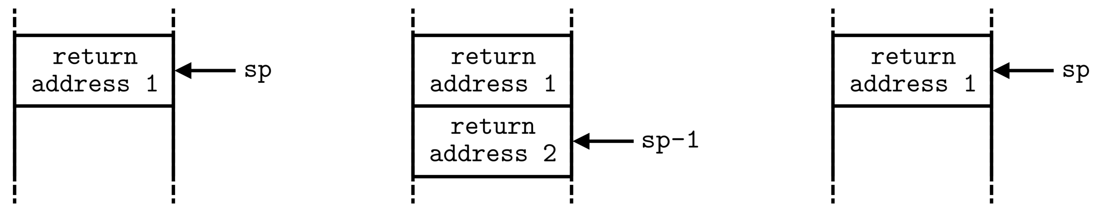
Taking the x86 instruction set architecture as an example, A function call generally uses the `call` instruction, which lets the CPU jump to the first instruction associated with the called function. However, before this jump, the address of the instruction after `call`, also known as the return address, is stored onto a stack. When the CPU has finished executing the function, the return address is fetched from the stack after which the CPU jumps to said address.

A stack, classified as a last-in-first-out (LIFO) queue, is a convenient component for keeping track of return addresses. This is because it effectively models the sequence of function calls, without the need for the user program to keep track of memory addresses. This is accomplished by so-called push and pop operations. As the name suggests, a push operation pushes a value to the top of the stack. A function call thus pushes the return address onto the stack. In contrast, a pop operation returns and removes the top-most value on the stack.

Internally, a stack needs to keep track of the location in memory of its top-most entry. For this reason, a stack pointer is used, which contains the address of the top stack entry. The procedure for the push and pop operations can be readily observed in the example shown in Figure 3.1. Figure 3.1a shows a stack with the top-most entry represented as return address 1, which is pointed to by stack pointer `sp`. If a new return address is then pushed onto the stack, `sp` is decremented to $\texttt{sp} - 1$ so that return address 2 is stored at the new address pointed to by the new stack pointer value. The resulting stack after the push operation is shown in Figure 3.1b. Next, if a pop operation is initiated, the value at address $\texttt{sp} - 1$, the top-most stack entry, is retrieved from the stack and the stack pointer is incremented to its original value `sp`, as shown in Figure 3.1c.

Note that, by historical precedent, pushing items onto a stack actually grows the stack from higher addresses to lower addresses or in other words from top to bottom. For this reason the stack pointer is decremented on a push and incremented on a pop in the example. Also note that in a true stack, push and pop operations only operate on the top stack entry.

Getting back to parsing machines, invocation of non-terminals is implemented the same as invocation of functions described in this section. That is, every call to a non-terminal



(a) Original stack. (b) Stack after push operation. (c) Stack after pop operation.

Figure 3.1: Example of stack operations.

in a PEG expression pushes the address of the next instruction, the return address, onto a return stack. Thereafter, the parsing machine jumps to the address of the first instruction of the called non-terminal. Once the non-terminal has been evaluated successfully, the top stack entry is popped and its value is used to jump to the instruction after the successful non-terminal call.

3.1.2.2 Backtrack Stack

Similar to non-terminal invocation, the limited backtracking property of PEG also lends itself well to a stack-based implementation. Consider the prioritized choice PEG expression below. As described in Section 2.4.1.4, the state of the parsing machine must be stored before executing the first operand expression e_1. Then, if a point of failure is encountered during execution of e_1, the last stored parsing machine state is used to restore the machine and resume execution at the second operand of the ordered choice operation e_2.

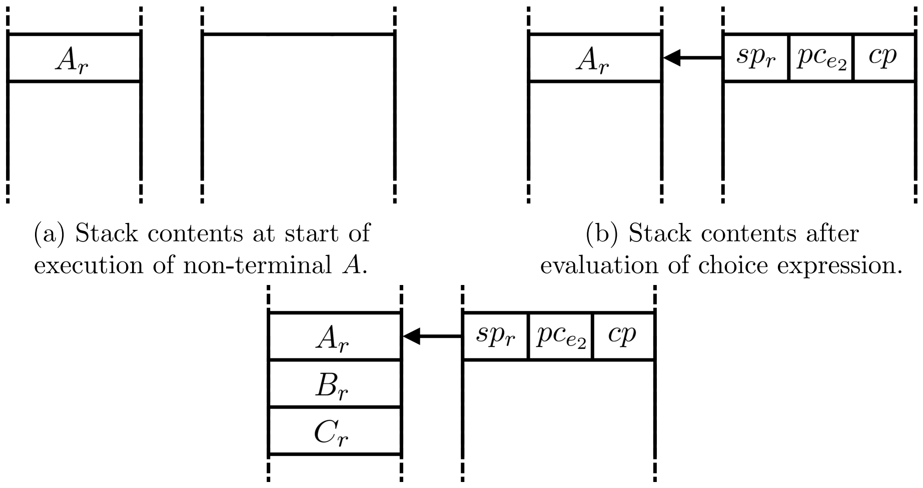
$$A \leftarrow e_1 \ / \ e_2$$

The parsing machine state that needs to be stored in the backtrack stack consists of the following elements:

- **Character position**: If a point of failure is encountered during execution of some expression, all characters that have been parsed since the last state store must be undone in order to be reparsed via some other alternative expression. To that end, the character position value, which is used as data memory read address as discussed in Section 3.1.4, must be stored as part of the parsing machine state.
- **Program counter**: At a point of failure, execution must resume at some alternative expression. Taking the prioritized choice $A \leftarrow e_1 \ / \ e_2$ as an example, if execution of expression e_1 fails, the parsing machine must resume execution at alternative expression e_2. For this reason, the address of the first instruction after backtracking must be stored as part of the machine state.
- **Return stack pointer**: Any active calls to non-terminals during execution of some expression must be exited immediately when a point of failure is encountered. This is achieved by storing the return stack pointer value as part of the parsing machine state.

To clarify backtracking by the parsing machine by means of a backtrack stack, an example of this operation is provided. For this example, consider the PEG grammar as follows:

$$\begin{aligned}
&1.\ A \leftarrow B\ /\ e_2\\
&2.\ B \leftarrow C\\
&3.\ C \leftarrow e_1
\end{aligned} \tag{3.1}$$



(a) Stack contents at start of execution of non-terminal A.

(b) Stack contents after evaluation of choice expression.

(c) Stack contents at start of execution of non-terminal C.

Figure 3.2: Return stack and backtrack stack contents during backtrack operation assuming the PEG grammar presented in Equation (3.1).

Figure 3.2a shows both the return stack (left) and backtrack stack (right) contents at the start of execution of non-terminal A.

Then the current parsing machine state is stored as on the backtrack stack, because the first operation of the expression belonging to non-terminal A is a prioritized choice operation. The resulting stack contents are shown in figure 3.2b. Here, cp represents the current character position; pc_{e_2} represents the program counter value (instruction address) of the first instruction of expression e_2; and sp_r represents the current return stack pointer, which points to the return address associated with the call to non-terminal A as visualized by the arrow.

Next, the subroutine associated with non-terminal B is called, which in turn calls the subroutine associated with non-terminal C. Their return addresses (B_r and C_r respectively) are sequentially stored in the return stack as shown on the left in Figure 3.2c.

If at this point execution of expression e_1 fails, a backtrack operation is required. This is achieved by popping the top backtrack stack entry and using its contents to restore three key registers of the parsing machine (see Section 3.1.4 for a more detailed account of the registers inside the parsing machine). Considering the contents of the backtrack stack presented in 3.2c, the character position register is restored to *cp*, the program counter register is set to pc_{e_2}, and the return stack pointer register is set to sp_r.

After this backtrack operation, the stack contents have been reverted back to that at the start of execution of non-terminal A, which can be seen in Figure 3.2a. However, execution now resumes at the alternative expression e_2 of the prioritized choice.

Note that in the presented example that a backtracking operation directly impacts the value of return stack pointer. Because a backtrack operation can only cause active non-terminal calls to be exited, the change in return stack pointer value is always positive (stack grows down). It can therefore be stated that a backtrack operation can cause zero or more return stack entries to be popped at once.

3.1.3 Control Unit

The control unit is one of the most important components in a computer architecture, and so too in a parsing machine architecture. This component controls the flow of all data within the parsing machine. It does so with only the currently executing instruction as input. Each type of instruction has a unique code in a section of the instruction called the operation code or opcode. During execution, this is decoded by the control unit and all control signals that leave the control unit are configured, such that all data flows from and to the parsing machine's components as necessary.

3.1.4 Register File

Unlike the register file in a conventional computer architecture, this parsing machine has no general-purpose registers, but only the following status registers:

- `$pc`: program counter register; directly connected to the read address bus of the instruction memory component and therefore holds the address of the currently executing instruction (see Section 3.1.1.1).
- `$cpos`: character position register; directly connected to the read address bus of the data memory component and therefore holds the address of the character currently being parsed (see Section 3.1.1.2).
- `$fs`: file size register; holds a constant positive integer representing the length of the input string currently stored in data memory.
- `$rsp`: return stack pointer; holds the address of the top return stack entry (see Section 3.1.2.1).
- `$bsp`: backtrack stack pointer; holds the address of the top backtrack stack entry (see Section 3.1.2.2).

The file size register `$fs` is of special interest for the correct functioning of the parsing machine architecture. From the point of view of the parsing machine, this is actually a read-only register and contains the length of the input string. Part of its function is to protect the parsing machine from reading characters outside the bounds of the input string. Furthermore, the same way that files are stored on a Unix-based operating system, there is no end-of-file character that marks the end of the input string. Instead, the length of the file is stored in this register, which can be checked by a grammar to determine if the complete string has been parsed or only a substring. More implementation details regarding register `$fs` are discussed in Section 3.3.

3.2 Operational Semantics of PEG

The aim of the following sections is to construct a formalism for translation of PEG grammars to parsing machine program assuming the components described in Section 3.1 are used in the parsing machine architecture.

First, however, this section aims to formalize Parsing Expression Grammar (PEG) in terms of the computational behavior of a top-down recognizer. This is achieved by defining operational semantics of PEG in the form of a `match` function. These formalisms are used in later sections to verify correct PEG behavior of the parsing machine architecture.

Section 3.2.1 defines the necessary prerequisites for the formalisms of the operation semantics. Finally, Section 3.2.2 defines and describes the operational semantics for each fundamental PEG expression.

3.2.1 `match` Function Declaration

The operational semantics of PEG is defined as a function called `match`. This function is formally declared as seen in Equation (3.2). Here, the sets before the arrow represent the domain (function arguments) and the sets after the arrow represent the codomain (function return values). The domain consists of the set of PEG grammars $\mathcal{G}$; the set of PEG expressions $\mathcal{E}$; the set of possible strings of terminals with arbitrary length Σ^* (see Section 2.1); and the set of natural numbers $\mathbb{N}$ (i.e., 0, 1, 2, ...). The latter represents the character position at the start of the `match` operation. The result of `match` operation, or the codomain, is either the new character position (set of natural numbers) if the operation was successful, otherwise the result is $\varnothing$ (null).

$$\texttt{match}: \ \mathcal{G} \times \mathcal{E} \times \Sigma^* \times \mathbb{N} \rightarrow \mathbb{N} \cup \{\varnothing\} \tag{3.2}$$

Equation (3.3) to Equation (3.9) show the exact definition of the set of expressions $\mathcal{E}$. The first four equations define the fundamental elements that are used in all expressions of any size. These include the empty string ϵ, the "any character" element '.', character element c, and the character class $[m]$. Equation (3.7) shows how the set of expressions also contains the result of the five unary PEG operations applied to its elements. In order, these are the "optional" match expression $e?$; the "zero-or-more" expression $e*$ and "one-or-more" match expression $e+$; and the "and-predicate" and "not-predicate" match expressions $\&e$ and $!e$ respectively. Next, Equation (3.8) defines the set of expressions contained in $\mathcal{E}$ resulting from application of binary PEG operations on its elements. In order, these are the "sequence" match expression e_1e_2 and the "prioritized choice" match operation e_1/e_2. Finally, the last type of expression, as defined in Equation (3.9), is a non-terminal match expression A_k, assuming A_k is an element of the set of non-terminals N of grammar G.

$$\epsilon \in \mathcal{E} \tag{3.3}$$
$$\text{`.'} \in \mathcal{E} \tag{3.4}$$
$$\text{`}c\text{'} \in \mathcal{E} \tag{3.5}$$
$$[m] \in \mathcal{E} \tag{3.6}$$
$$\big\{e?\,,\ e*\,,\ e+\,,\ \&e\,,\ !e \bigm| e \in \mathcal{E}\big\} \in \mathcal{E} \tag{3.7}$$
$$\big\{e_1e_2\,,\ e_1/e_2 \bigm| \{e_1, e_2\} \in \mathcal{E}\big\} \in \mathcal{E} \tag{3.8}$$
$$\big\{A_k \bigm| A_k \in N\,,\ N \in G\big\} \in \mathcal{E} \tag{3.9}$$

3.2.2 `match` Function Definition

In this section, the declaration of the `match` function as defined in the previous section is expanded to include a proper definition. Definition of the `match` function that properly describes the parsing behavior of any PEG grammar is achieved by means of an inductive definition. That is, for each type of expression (as in Equation (3.3) to Equation (3.9)), the `match` function is defined based on a simpler set of definitions or rules, which together form a so-called rule of inference. [42]

Equation (3.10) shows the form that is used for rules of inference. Here, P_1 to P_n form the premises and C represents the conclusion of the rule of inference. The equation can be read as stating that the statement C holds true if and only if all statements $P_1, \ldots, P_n$ hold true. [42]

$$\frac{\begin{matrix} P_1 \\ \cdots \\ P_n \end{matrix}}{C} \tag{3.10}$$

All concluding statements C of the inductive definitions for the `match` function have the format $\texttt{match}(G, e, s, i) = o$, conforming to its declaration in Equation (3.2). Here, G is any PEG grammar, e is the expression under evaluation, s is the input string that is to be parsed with the given grammar, and i is the character position in string s from where the current evaluation occurs. Finally o represents the result of the function, which can either be of the form $\varnothing$ or $i+n$. The first is true if the `match` operation fails, whereas the latter is true if the operation succeeds. Note that n can only be a non-negative number, such that for any successful match, the character position advances forward only.

Furthermore, as will become apparent in the subsequent definitions, some PEG expressions are defined as recursive statements. This is to say that one or more premises contain an invocation of `match` themselves. In this case, it is important that those rules of inference avoid infinite regress. In other words, such recursive statements must eventually terminate for the conclusion to hold true.

The rest of this section defines the `match` function for each PEG expression.

3.2.2.1 Empty String

The evaluation of any "empty string" PEG expression succeeds unconditionally, as no characters are consumed. For this reason, the single rule of inference for an empty string expression ϵ simply evaluates to the original character position i, as can be seen in Equation (emp.1).

$$\frac{e = \epsilon}{\texttt{match}(G, e, s, i) = i} \tag{emp.1}$$

3.2.2.2 Literal String

A literal string can be split up into a sequence of single character expressions. Therefore, here only a literal string consisting of a single character is assumed.

The success of the `match` function for a single character expression 'c' depends on two main premises: the current character position i and the character at position i of the input string s. For the first, assuming zero-based string indexing, if the character position i is equal or larger than the length of input string s, the `match` operation must fail and evaluate to $\varnothing$, as the character position i is out of bounds of string s. This can be observed in Equation (lit.3).

However, when i is within the bounds of input string s, the success of `match` depends on the equality of character c and the currently indexed character of string s. If the equality holds true, the character position is advanced by a single character. On the other hand, if no equality holds, the operation fails and still evaluates to $\varnothing$. This is presented in Equation (lit.1) and Equation (lit.2) respectively.

$$\frac{\begin{array}{c} e = \text{`}c\text{'} \\ i < |s| \\ s[i] = \text{`}c\text{'} \end{array}}{\texttt{match}(G, e, s, i) = i + 1} \tag{lit.1}$$

$$\frac{\begin{array}{c} e = \text{`}c\text{'} \\ i < |s| \\ s[i] \neq \text{`}c\text{'} \end{array}}{\texttt{match}(G, e, s, i) = \varnothing} \tag{lit.2}$$

$$\frac{\begin{array}{c} e = \text{`}c\text{'} \\ i \geq |s| \end{array}}{\texttt{match}(G, e, s, i) = \varnothing} \tag{lit.3}$$

3.2.2.3 Character Class

Where the expression 'c' can only represent a single character, the "character class" expression $[m]$ matches with any input character contained in string m.

In this report, the character class PEG expression is formally defined as in Equation (3.11). As can be seen, string m is composed of characters c_x and so-called character ranges or sets $c_{x,1}$-$c_{x,2}$. Recall from Section 3.1.1.2 that a character c is represented by an 8-bit value, such that c has a range of $[0, 255]$. A character range $c_{x,1}$-$c_{x,2}$ is therefore equivalent to a sequence of characters with values ranging from $c_{x,1}$ to $c_{x,2}$, e.g., 17-145 = 17 18 $\cdots$ 144 145.

$$\begin{aligned} &m = C_1 C_2 \cdots C_n \\ &\text{where } C_x = c_x \text{ or } C_x = c_{x,1}\text{-}c_{x,2} \text{ given } c_{x,1} \leq c_{x,2} \\ &\text{and } 1 \leq x \leq n \end{aligned} \tag{3.11}$$

Ultimately, $[m]$ is defined as the set of characters contained in string m. Consider for example a character class $[m]$ = [.0-9A-Za-z] representing ASCII-encoded horizontal space character, full stop character, and alphanumeric characters. Using Equation (3.11), m can be deconstructed into individual characters $C_1 =$ ' ' and $C_2 =$ '.', and character ranges $C_3 =$ '0'-'9', $C_4 =$ 'A'-'Z', and $C_5 =$ 'a'-'z'. Then, writing out the ranges C_3 to C_5 into their equivalent sequence of characters, it holds that $m =$ " .0123456789abc$\cdots$xyzABC$\cdots$XYZ". Finally, as stated earlier that $[m]$ is the set of characters contained in string m, it follows that $[m] = \{$' ', '.', '0', '1', $\ldots$, '8', '9', 'a', 'b', $\ldots$, 'y', 'z', 'A', 'B', $\ldots$, 'Y', 'Z'$\}$, which, if these elements are converted to their equivalent ASCII-code [43], is equivalent to $[m] = \{32, 46, 48, 49, \ldots, 56, 57, 65, 66, \ldots, 96, 97, 98, \ldots, 121, 122\}$.

Similar to the literal string expression, the success of the `match` function for a character class expression depends on both the character position i and the check of whether the currently indexed character is contained in the set of $[m]$. The result for failure of the out of bounds check for character position i can be observed in Equation (cls.3). The second is formalized in Equation (cls.1) and Equation (cls.2). Equation (cls.1) states that the `match` operation applied to a character class expression $[m]$ advances the character position by one character if the currently indexed character of the input string s is present in the set of characters represented by $[m]$. Equation (cls.2) defines an evaluation to $\varnothing$ if the indexed character is not an element of $[m]$.

$$\frac{e = [m] \quad i < |s| \quad s[i] \in [m]}{\texttt{match}(G, e, s, i) = i + 1} \tag{cls.1}$$

$$\frac{e = [m] \quad i < |s| \quad s[i] \notin [m]}{\texttt{match}(G, e, s, i) = \varnothing} \tag{cls.2}$$

$$\frac{e = [m] \quad i \geq |s|}{\texttt{match}(G, e, s, i) = \varnothing} \tag{cls.3}$$

3.2.2.4 Any Character

As the name implies, evaluation of the `match` function with as argument the "any character" PEG expression '.' increments the character position i by one character, regardless of the value of the currently indexed character of the input string s.

However, the success of the `match` function still depends on the value of i. If the character position value is lower than the input string length, the `match` function successfully evaluates to the incremented character position value, as seen in Equation (any.1). Otherwise, a null value is returned as in Equation (any.2).

$$\frac{e = \text{'.'} \quad i < |s|}{\texttt{match}(G, e, s, i) = i + 1} \quad \text{(any.1)} \qquad \frac{e = \text{'.'} \quad i \geq |s|}{\texttt{match}(G, e, s, i) = \varnothing} \quad \text{(any.2)}$$

3.2.2.5 Optional

The "optional" match expression is the result of the unary PEG operation '?' applied to some expression e'. In short, an expression to which the optional PEG operation is applied, does not need to be matched successfully itself. That is, if the `match` operation on expression e' fails, the `match` operation on $e'?$ still succeeds, but simply evaluates to the same character position i as it started with. This situation is formally defined in Equation (opt.2).

If evaluation of the `match` operation applied to expression e' does succeed, thereby advancing the character position by j characters, then the optional match expression $e'?$ succeeds too and evaluates to a new character position value $i + j$. This can be seen in Equation (opt.1).

$$\frac{e = e'? \quad \texttt{match}(G, e', s, i) = i + j}{\texttt{match}(G, e, s, i) = i + j} \quad \text{(opt.1)} \qquad \frac{e = e'? \quad \texttt{match}(G, e', s, i) = \varnothing}{\texttt{match}(G, e, s, i) = i} \quad \text{(opt.2)}$$

3.2.2.6 Zero-Or-More

The "zero-or-more" match expression is another result of a PEG unary operation, namely by the '$*$' operator applied to some expression e'. Similar to the optional expression, the expression e' need not be matched successfully itself. That is, if e' fails, the `match` operation applied to $e'*$ still succeeds, thereby returning the initial character position i, as shown in Equation (zom.2)

If the `match` operation applied to e' does succeed, it is assumed that it increases the character position i by j characters. Thereafter, a new `match` operation is applied to the same zero-or-more expression $e'*$, only this time starting from character position $i + j$. Assuming this operation increments the character position by another k characters, the complete `match` operation succeeds and evaluates to the new character position $i+j+k$. This situation is presented in Equation (zom.1).

Note that the recursive `match` operation in Equation (zom.1) must eventually terminate for any finite-length string s. Recursion terminates when the `match` operation applied to e' eventually evaluates to null for some character position $i' \geq i$ according to Equation (zom.2).

$$\frac{\begin{array}{l} e = e'* \\ \texttt{match}(G, e', s, i) = i + j \\ \texttt{match}(G, e'*, s, i + j) = i + j + k \end{array}}{\texttt{match}(G, e, s, i) = i + j + k} \tag{zom.1}$$

$$\frac{\begin{array}{l} e = e'* \\ \texttt{match}(G, e', s, i) = \varnothing \end{array}}{\texttt{match}(G, e, s, i) = i} \tag{zom.2}$$

3.2.2.7 One-Or-More

The "one-or-more" match expression is another repetition-based unary PEG operation similar to the zero-or-more expression. This type of expression is indicated by the ' + ' postfix operator applied to an expression e'. Unlike the zero-or-more operation, the one-or-more PEG operation requires at least one successful match operation applied to expression e' in order for the complete expression $e'+$ to succeed. This can be readily observed in Equation (oom.2).

However, when the `match` expression applied to e' does succeed the first time and thereby increments the character position by j characters, a subsequent match operation applied to the zero-or-more expression $e'*$ is invoked. This case is presented in Equation (oom.1).

$$\frac{\begin{array}{l} e = e'+ \\ \texttt{match}(G, e', s, i) = i + j \\ \texttt{match}(G, e'*, s, i + j) = i + j + k \end{array}}{\texttt{match}(G, e, s, i) = i + j + k} \tag{oom.1}$$

$$\frac{\begin{array}{l} e = e'+ \\ \texttt{match}(G, e', s, i) = \varnothing \end{array}}{\texttt{match}(G, e, s, i) = \varnothing} \tag{oom.2}$$

3.2.2.8 And-Predicate

The "and-predicate" match expression is one of two predicate-based unary PEG operations and is identified by the '&' prefix operator applied to some expression e'. The predicate-based operations are unique in that they do not advance the character position i when evaluated successfully.

The and-predicate match expression succeeds simply when evaluation of expression e' succeeds. However, if that successful evaluation increments the character position by j characters, this is not propagated to the evaluation result of $\&e'$. Instead, the character position is never advanced on successful evaluation of e', which is the main property of predicate-based operations. This behavior is shown in Equation (and.1). Moreover, failure to evaluate expression e' directly translates to a null-evaluation of $\&e'$, which is shown in Equation (and.2).

$$\frac{\begin{array}{l} e = \&e' \\ \texttt{match}(G, e', s, i) = i + j \end{array}}{\texttt{match}(G, e, s, i) = i} \quad \text{(and.1)} \qquad \frac{\begin{array}{l} e = \&e' \\ \texttt{match}(G, e', s, i) = \varnothing \end{array}}{\texttt{match}(G, e, s, i) = \varnothing} \quad \text{(and.2)}$$

3.2.2.9 Not-Predicate

The "not-predicate" match expression is the only other predicate-based PEG operation and is represented by the '`!`' prefix operator applied to an expression e'.

The not-predicate match expression behaves opposite to that of the and-predicate expression. Successful evaluation of $!e'$ occurs only when expression e' itself evaluates to $\varnothing$. That is, $!e'$ succeeds when e' fails, which is formalized in Equation (not.1). When application of the `match` operation to e' succeeds, however, the evaluation of expression $!e'$ fails, as can be observed in Equation (not.2).

$$\frac{\begin{array}{l} e = !e' \\ \texttt{match}(G, e', s, i) = \varnothing \end{array}}{\texttt{match}(G, e, s, i) = i} \quad \text{(not.1)} \qquad \frac{\begin{array}{l} e = !e' \\ \texttt{match}(G, e', s, i) = i + j \end{array}}{\texttt{match}(G, e, s, i) = \varnothing} \quad \text{(not.2)}$$

3.2.2.10 Sequence

The "sequence" match expression is the result of one of two binary PEG operations, which concatenates the `match` operation behavior of two contiguous expressions e_1 and e_2.

The only path to successful evaluation of a sequence expression is if both expressions independently evaluate to success. That is, assuming evaluation of e_1 results in a j-character increment of the character position, and subsequent evaluation of e_2, starting at character position $i + j$, results in a k-character increment of the character position, then the expression e_1e_2 successfully evaluates to $i + j + k$. This case is presented in Equation (seq.1).

There are two distinct ways for any sequence expression to fail evaluation. First, assuming expression e_1 successfully evaluates to $i + j$, failure to evaluate e_2 to any natural number leads to a null-evaluation of the sequence expression e_1e_2 (see Equation (seq.2)). Second, if e_1 evaluates to $\varnothing$, expression e_2 need to be evaluated, and instead the `match` operation applied to the sequence expression e_1e_2 directly evaluates to $\varnothing$ (see Equation (seq.3)).

$$\frac{\begin{array}{l} e = e_1e_2 \\ \texttt{match}(G, e_1, s, i) = i + j \\ \texttt{match}(G, e_2, s, i + j) = i + j + k \end{array}}{\texttt{match}(G, e, s, i) = i + j + k} \quad \text{(seq.1)} \qquad \frac{\begin{array}{l} e = e_1e_2 \\ \texttt{match}(G, e_1, s, i) = i + j \\ \texttt{match}(G, e_2, s, i + j) = \varnothing \end{array}}{\texttt{match}(G, e, s, i) = \varnothing} \quad \text{(seq.2)}$$

$$\frac{\begin{array}{l} e = e_1e_2 \\ \texttt{match}(G, e_1, s, i) = \varnothing \end{array}}{\texttt{match}(G, e, s, i) = \varnothing} \quad \text{(seq.3)}$$

3.2.2.11 Prioritized Choice

The "prioritized choice" match expression is the second binary PEG operation and is identified by the '/' operator in the expression e_1 / e_2. As the name implies, its behavior matches that of a choice between either successful evaluation of e_1 or alternatively of e_2 if the first fails.

There are two distinct paths to successful evaluation of application of the `match` operation to the prioritized choice expression e_1 / e_2. First, if evaluation of e_1 results in the advancement of the character position by j characters, then evaluation of the prioritized choice expression succeeds with the same increase in character position (see Equation (prc.1)). Second, if evaluation of e_1 fails, and the `match` operation applied to e_2 does succeed, thereby advancing the character position by k characters, then evaluation of e_1 / e_2 succeeds also with a character position increase of k characters (see Equation prc.2)).

The only case in which evaluation of the `match` operation applied to e_1 / e_2 does indeed produce a null-result, is when, starting with character position i, evaluation of both expression e_1 and expression e_2 independently fail (see Equation prc.3)).

Note from the inductive definition of the prioritized choice match expression that the order of evaluation of expressions e_1 and e_2 matters. That is to say, evaluation of e_1 has priority over evaluation of e_2. This is unlike the normal choice operator ' | ' in BNF-syntax introduced in Section 2.1.1.

$$\frac{\begin{array}{l} e = e_1 \,/\, e_2 \\ \texttt{match}(G, e_1, s, i) = i + j \end{array}}{\texttt{match}(G, e, s, i) = i + j} \tag{prc.1}$$

$$\frac{\begin{array}{l} e = e_1 \,/\, e_2 \\ \texttt{match}(G, e_1, s, i) = \varnothing \\ \texttt{match}(G, e_2, s, i) = i + k \end{array}}{\texttt{match}(G, e, s, i) = i + k} \tag{prc.2}$$

$$\frac{\begin{array}{l} e = e_1 \,/\, e_2 \\ \texttt{match}(G, e_1, s, i) = \varnothing \\ \texttt{match}(G, e_2, s, i) = \varnothing \end{array}}{\texttt{match}(G, e, s, i) = \varnothing} \tag{prc.3}$$

3.2.2.12 Non-Terminal

The final PEG expression to be inductively defined is the "non-terminal" expression, indicated by A_k. In the context of PEG, a non-terminal expression is nothing more than a label attached to some expression e [3].

In order to recursively match an expression by means of the `match` function, all expressions listed in Equation (3.7) to Equation (3.9) must in the end reduce to one of the fundamental expressions listed in Equation (3.3) to Equation (3.6). Therefore, any non-terminal expression A_k must map to an expression e as defined in its right-hand side of its associated production rule in grammar G. That is, given a PEG production rule $A_k \leftarrow e$, a `match` operation applied to the expression A_k is equal to the `match` operation applied to e. To this end, a partial function `ntmap` is declared as in Equation (3.12). This can be read as stating that the function `ntmap` evaluates to the right-hand side expression e of the production rule for non-terminal A_k, assuming this production rule is defined in grammar G.

$$\texttt{ntmap}: \mathcal{G} \times N \to \mathcal{E} \tag{3.12}$$

$$\texttt{ntmap}(G, A_k) = \{e \mid (A_k \to e) \in G\} \tag{3.13}$$

Using the mapping function `ntmap`, application of the `match` operation to the non-terminal expression A_k successfully evaluates if and only if evaluation of expression e' succeeds, where e' is the result from the mapping of non-terminal A_k given grammar G. Moreover, assuming evaluation of expression e' resulted in a j-character increase of the character position, evaluation of A_k results in the same increase. This is formalized in Equation (nte.1).

Likewise, failure to evaluate expression e' to a natural number results in an evaluation of the non-terminal expression to $\varnothing$, as seen in Equation (nte.2).

$$\frac{\begin{array}{l} e = A_k \\ \texttt{ntmap}(G, A_k) = e' \\ \texttt{match}(G, e', s, i) = i + j \end{array}}{\texttt{match}(G, e, s, i) = i + j} \quad \text{(nte.1)} \qquad \frac{\begin{array}{l} e = A_k \\ \texttt{ntmap}(G, A_k) = e' \\ \texttt{match}(G, e', s, i) = \varnothing \end{array}}{\texttt{match}(G, e, s, i) = \varnothing} \quad \text{(nte.2)}$$

3.2.2.13 Conclusion

In this section, the operational semantics of all PEG expressions were defined by means of the `match` function. Assuming grammar $G = (\Sigma, N, S, P)$ and input string s, the behavior of a top-down PEG parser must follow the operational behavior as determined by the evaluation of $\texttt{match}(G, S, s, 0) = i$. Here, S represents the start-symbol of grammar G and i represents the final character position after the successful parse of string s. However, note that, unless specified in the grammar, it does not necessarily hold that $i = |s|$, but instead more generally $0 \leq i \leq |s|$.

3.3 Microcode of Parsing Machine Architecture

In the previous section a mathematical definition was provided for top-down PEG parsing behavior. This section aims to describe the parsing behavior of the PPEG parsing machine architecture for each of the PEG expressions formally defined in Section 3.2. These behavioral descriptions are later used in subsequent sections to derive a dedicated instruction set architecture.

3.3.1 Microcode Terminology

With the introduction of the main PPEG architectural components in Section 3.1, the behavior of the parsing machine architecture can be described in terms of the movement of data between its various components. This is achieved by introducing a concept called microcode [44].

In the context of this report, microcode consists of so-called micro-operations, or μops, which are single atomic operations that operate on some data stored in a register, memory, stack, or constant. A μop is represented by a single statement with a C-like syntax (C, the programming language). There are two types of operations that can be represented by μop. First is the comparison operation, indicated by the operators '`==`' (equal-to), '`<=`' (smaller-than-or-equal-to), and '`>=`' (greater-than-or-equal-to). Second is the data transfer operation, indicated by the '`=`' operator, which represents a transfer of data at right-hand side to the component indicated on the left-hand side.

The various PPEG components discussed in Section 3.1 are represented in microcode by the following keywords:

Registers:

- `$pc`: program counter
- `$cpos`: character position
- `$fs`: file size
- `$rsp`: return stack pointer
- `$bsp`: backtrack stack pointer

Memory:

- `imem`: instruction memory
- `dmem`: data memory

Stacks:

- `rs`: return stack
- `bs`: backtrack stack

The addressing of memory and stack components is achieved by the index operation notated by `x[i]`. Here, `x` is the component that is addressed and `i` is the address. For example, the currently executing instruction is obtained by addressing instruction memory as `imem[$pc]`. A special case of the index operation is the addressing of a single backtrack stack entry. As discussed in Section 3.1.2.2, a backtrack stack entry consists of a character position value, program counter value, and return stack pointer value. Given backtrack stack entry `x`, in order to address these values individually, the index operations `x[0]`, `x[1]`, and `x[2]` are used respectively.

Microcode programs can also include single-line labels. These are used as alias for an address of the micro-operation on that same line. Any reference to a specific micro-operation is therefore achieved by referring to its label. For example, the microcode in this report may set the program counter register value to a label that references the next micro-operation to be executed. The use of labels is highlighted in subsequent sections.

The microcode programs of many of the fundamental PEG expressions need to refer to the microcode of other expressions. For example, in a subsequent section the microcode for the optional PEG expression $e?$ is provided. This code, however, also includes the code of expression e. To refer to the microcode program for any expression e, a built-in function `c(e)` is used.

Finally, although the C-like syntax might give the impression that microcode programs are by definition executed sequentially, fundamentally it describes hardware behavior.

As such, a sequence of contiguous micro-operations may in practice be implemented in parallel. Which micro-operations happen in parallel is discussed in more detail in Section 3.4.

3.3.2 PEG Expressions to Microcode

In the subsections hereafter, each PEG expression as defined in Equation (3.3) to Equation (3.9) is translated to a microcode program. The program is constructed in accordance with the operational semantics of the PEG expressions defined in Section 3.2.2.

3.3.2.1 Empty String

The empty string PEG expression, as functionally defined in Section 3.3, has zero lines of equivalent microcode. The reason for this is that such an expression succeeds unconditionally and does not change any of the status registers.

3.3.2.2 Literal String

Similar to the operational semantic definition of the literal string PEG expression, the microcode translation of the literal string expression only focuses on a single-character length string. Microcode for a literal string of arbitrary length "$c_1c_2 \cdots c_n$" is equivalent to microcode for a PEG sequence expression of single-character literal strings 'c_1' 'c_2' $\cdots$ 'c_n'.

Listing 3.1 presents the microcode implementation for a single-character literal string PEG expression 'c'. The first thing to notice is the greater-than-or-equal-to comparison between the contents of the character position register and file size register in line 1. This comparison is a translation from the operational semantics definition in Equation (lit.3), where the character position i is compared against the input string length $|s|$. As dictated by the operational semantics, if the character position, stored in `$cpos`, is equal to or larger than the input string length, stored in `$fs`, a backtrack operation must be invoked.

The microcode for a backtracking operation can be seen in lines 2 to 6 of Listing 3.1. In line 2, the top backtrack stack entry value `bs[$bsp]` is temporarily stored in a variable `t0`, after which the backtrack stack pointer `$bsp` is incremented in line 3. These two micro-operations together are equivalent to a stack pop operation as discussed in Section 3.1.2.1. The character position, return stack pointer, and program counter are then set to the values contained in the popped backtrack stack entry as seen in lines 4, 5, and 6 respectively. The setting of the program counter in line 6 results in a jump to the instruction address stored in the popped backtrack stack entry.

If the comparison in line 1 fails, the microcode in lines 9 to 19 is executed. Line 9 introduces yet another comparison, this time between the current character of the input string `dmem[$cpos]`, addressed by the character position register `$cpos`, and the character `c` from the expression 'c' under evaluation. If the comparison proves true, the parsing machine behavior should adhere to the PEG definition specified in Equation (lit.1). This entails incrementing the character position by a single character (line 10) and jumping to the first micro-operation after the microcode for this expression at the address labeled `L0` (line 11).

If the comparison in line 9 fails too, lines 14 to 18 are executed and directly reflect the PEG behavior defined in Equation (lit.2). That is, a backtrack operation is invoked, because the active character of the input string `dmem[$cpos]` does not match character `c`. The backtracking microcode is exactly the same as that in lines 2 to 6 discussed previously.

```
 1     if ($cpos >= $fs) {
 2         t0 = bs[$bsp]
 3         $bsp = $bsp + 1
 4         $cpos = t0[0]
 5         $rsp = t0[2]
 6         $pc = t0[1]
 7     }
 8     else {
 9         if (dmem[$cpos] == c) {
10             $cpos = $cpos + 1
11             $pc = L0
12         }
13         else {
14             t1 = bs[$bsp]
15             $bsp = $bsp + 1
16             $cpos = t1[0]
17             $rsp = t1[2]
18             $pc = t1[1]
19         }
20     }
21 L0: ...
```

Listing 3.1: Microcode implementation for literal string PEG expression ‘c’.

3.3.2.3 Character Class

As explained in Section 3.2.2, the character class PEG expression $[m]$ represents a set of valid characters that can match with the input character at that moment. One way to implement the microcode for this expression is to replicate the code for the literal string expression in Listing 3.1 and extend it to compare all elements of string m with the current input character. This approach can be observed in Listing 3.2.

Lines 1 to 7 are exactly the same as in Listing 3.1 in that the input string length is compared against the current character position value, which causes a backtrack operation if proven equal.

In lines 9, 13, and 18 the comparison between the active character `dmem[$cpos]` and each character in string m can be observed. Note that the number of comparisons is equal to the number of elements n in string m. If any of these comparisons result in a positive match, the character position `$cpos` is incremented and the parsing machine jumps to the first micro-operation after the microcode for this character class expression, which has the address labeled `L0` (line 28). This parsing machine behavior is in accordance with the operational semantics presented in Equation (cls.1).

On the other hand, if none of the comparisons result in a positive match, the microcode in lines 22 to 26 is executed, which represents the backtrack routine seen earlier in lines 2 to 6. This case is a translation of the operational semantics of the character class expression shown in Equation (cls.2). In short, only if the character class comparison succeeds is the character position incremented and the program resumed at the next micro-operation. Otherwise a backtrack operation is initiated.

```
1        if ($cpos >= $fs) {
2            t0 = bs[$bsp]
3            $bsp = $bsp + 1
4            $cpos = t0[0]
5            $rsp = t0[2]
6            $pc = to[1]
7        }
8        else {
9            if (dmem[$cpos] == m[0]) {
10               $cpos = $cpos + 1
11               $pc = L0
12           }
13           else if (dmem[$cpos] == m[2]) {
14               $cpos = $cpos + 1
15               $pc = L0
16           }
17           ...
18           else if (dmem[$cpos] == m[n-1]) {
19               $cpos = $cpos + 1
20               $pc = L0
21           }
22           t1 = bs[$bsp]
23           $bsp = $bsp + 1
24           $cpos = t1[0]
25           $rsp = t1[2]
26           $pc = t1[1]
27       }
28   L0: ...
```

Listing 3.2: Microcode implementation for character class PEG expression $[m]$.

One other microcode implementation, which may sometimes reduce the number of character comparisons, is to change the single-value comparison to a range comparison. Recall from Section 3.2.2.3 that string m is defined as a sequence of arbitrary length consisting of individual characters and character ranges (see Equation (3.11)). Therefore, instead of comparing the active character value `dmem[$cpos])` against each element in set $[m]$, one may check if this value is either equal to any of the individual characters or if it is within the range of any of the character ranges.

This can be achieved by substituting the comparisons in lines 9, 13, and 18 of Listing 3.1 by the following range comparison:

```
dmem[$cpos] >= c_1 && dmem[$cpos] <= c_2
```

Consider the alphanumeric character class expression [0-9A-Za-z] as an example to show how this new comparison reduces the number of comparisons. Instead of having 62 equality comparisons in microcode to check against each character in the character class, this could be reduced to 3 range comparisons as follows: '0' $\leq$ `dmem[$cpos]` $\leq$ '9', 'A' $\leq$ `dmem[$cpos]` $\leq$ 'Z', and 'a' $\leq$ `dmem[$cpos]` $\leq$ 'z'.

The practicality of the two microcode implementations is discussed further in Section 3.4.3.8.

3.3.2.4 Any Character

Like the microcode programs for the literal string and characters class PEG expressions, the microcode for the any character PEG expression presented in Listing 3.3 starts with a check for the character position value in line 1. If this check fails, a backtrack operation is initiated by executing lines 2 to 6, as dictated by operational semantics defined in Equation (any.2).

If the current character position value is below the file size value, the any character expression automatically succeeds as defined in Equation (any.1). In microcode, this entails incrementing the character position value and moving to the microcode for the next PEG expression in the sequence, which can be observed in lines 9 and 10 respectively of Listing 3.3.

```
 1     if ($cpos >= $fs) {
 2         t0 = bs[$bsp]
 3         $bsp = $bsp + 1
 4         $cpos = t0[0]
 5         $rsp = t0[2]
 6         $pc = t0[1]
 7     }
 8     else {
 9         $cpos = $cpos + 1
10         $pc = L0
11     }
12 L0: ...
```

Listing 3.3: Microcode implementation for any character PEG expression '.'.

3.3.2.5 Optional

Parsing machine behavior for the optional PEG expression $e?$ involves the unconditional execution of microcode associated with expression e, which is similar to how its operational semantics are defined as the result of the `match` function applied to expression e (see Equation (opt.1) and Equation (opt.2)).

The main property of the optional expression $e?$, however, is that it evaluates successfully regardless of the successful or unsuccessful evaluation of e. Unfortunately, only executing the microcode for expression e could invoke a backtrack operation, which would make the parsing machine jump to some ‘unknown’ instruction address. Instead, if evaluation of e fails, the parsing machine should simply continue its execution with the next PEG expression in the sequence, while retaining the original character position and return stack before execution of the microcode for e.

The above description for correct parsing machine behavior is achieved in the microcode presented in Listing 3.4. Before execution of microcode for expression e, lines 1 and 2 show that a new backtrack stack entry is pushed onto the backtrack stack, consisting of the current character position value, return stack value, and also the address of the microcode for the next in-sequence PEG expression indicated by label `L0`.

After the new backtrack stack entry, microcode for expression e is executed, which is represented by the built-in function `c(e)` employed in line 3. If execution fails here, the recently added and top-most backtrack stack entry is popped from the backtrack stack and used to restore the character position, program counter, and return stack pointer to the values shown in line 2 of Listing 3.4, which results in a jump by the parsing machine to the next instruction in line 5. If execution of `c(e)` succeeds, backtrack stack entry added in line 2 is popped from the backtrack stack, but without using any of its stored values. This is achieved in line 4 by simply incrementing the backtrack stack pointer. Thereafter, the machine resumes execution as normal in line 5.

As defined in the operational semantics of the optional PEG expression, the microcode for such an expression never fails and always ends up resuming execution at microcode for the next in-sequence expression. In other words, the parsing machine always ends up in line 5 of Listing 3.4.

```
1     $bsp = $bsp - 1
2     bs[$bsp] = {$cpos, L0, $rsp}
3     c(e)
4     $bsp = $bsp + 1
5 L0: ...
```

Listing 3.4: Microcode implementation for optional PEG expression $e?$.

3.3.2.6 Zero-Or-More

Like the optional PEG expression, the zero-or-more PEG expression cannot fail evaluation. For this reason, the first 3 lines of microcode for the zero-or-more expression in Listing 3.5 are the same as that of Listing 3.4. If execution of the microcode for expression e fails in line 3, the backtrack stack entry pushed in lines 1 and 2 cause a backtrack, which jumps program execution to line 7. This behavior is in accordance with the operational semantics defined in Equation (zom.2).

The difference in microcode between that of the zero-or-more expression and optional expression how successful execution of expression e is handled. Rather than simply executing the next expression in the sequence, Equation (zom.1) dictates the repeated evaluation of expression e, until evaluation finally fails. This is mirrored in lines 4 to 6 of Listing 3.5. Here, the top backtrack stack entry, which was pushed in lines 1 and 2, is updated to reflect the change in character position value after successful execution of expression e. Line 4 obtains the top backtrack stack entry; line 5 overrides the top backtrack stack entry with the new character position, but keeps the same program counter value as the original backtrack stack entry (address at label `L1`); and line 6 sets the program counter to the address of the first instruction of microcode for expression e, whose address is labeled as `L0`.

In short, the microcode for expression e is executed repeatedly until finally a parse fail is encountered, at which point the parsing machine jumps to the microcode for the next expression in the sequence.

```
1      $bsp = $bsp - 1
2      bs[$bsp] = {$cpos, L1, $rsp}
3 L0:  c(e)
4      t0 = bs[$bsp]
5      bs[$bsp] = {$cpos, t0[2], $rsp}
6      $pc = L0
7 L1:  ...
```

Listing 3.5: Microcode implementation for zero-or-more PEG expression $e*$.

3.3.2.7 One-Or-More

Microcode for the one-or-more PEG expression $e+$ as shown in Listing 3.6 consists solely of the microcode for expression e in sequence with the microcode for the zero-or-more expression $e*$. The reason for this is because the behavior of the PEG expression $e+$ is equivalent to that of $ee*$, which can be observed from its inductive definition in Equation (oom.1) and Equation (oom.2).

Note that, in contrast to the zero-or-more expression, the one-or-more expression can fail evaluation and cause a backtrack. This is due to the unprotected execution of expression e in line 1 of Listing 3.6.

```
1      c(e)
2      c(e*)
```

Listing 3.6: Microcode implementation for one-or-more PEG expression $e+$.

3.3.2.8 And-Predicate

Behavior for the and-predicate PEG expression execution must comply to the operational semantics in Equation (and.1) and Equation (and.2). Execution of the microcode translation of $\&e$ always starts by saving the current machine state. This is important for two reasons: saving the character position and saving the instruction address in case of failure to evaluate expression e.

If a execution of microcode for expression e in line 3 fails, the machine jumps to label `L0` in line 8. Here, it initiates another backtrack sequence, thereby failing execution of expression $\&e$ entirely.

If execution of expression e is successful, microcode in line 4 and 5 is executed, which pops the top backtrack entry and uses it only to restore the character position. This is in line with Equation (and.1) that states that evaluation of an and-predicate expression cannot advance the character position. Finally, the micro-operation in line 7 lets the parsing machine jump to the next in-sequence expression.

```
 1      $bsp = $bsp - 1
 2      bs[$bsp] = {$cpos, L0, $rsp}
 3      c(e)
 4      t0 = bs[$bsp]
 5      $cpos = t0[0]
 6      $bsp = $bsp + 1
 7      $pc = L1
 8  L0: t1 = bs[$bsp]
 9      $bsp = $bsp + 1
10      $cpos = t1[0]
11      $rsp = t1[2]
12      $pc = t1[1]
13  L1: ...
```

Listing 3.7: Microcode implementation for and-predicate PEG expression &e.

3.3.2.9 Not-Predicate

The microcode of a not-predicate PEG expression !e behaves similar to that of an and-predicate expression, although the result is the inverse. Lines 1 to 3 of Listing 3.8 are the same as that of Listing 3.7: push a new backtrack entry and attempt to execute expression e. However, if the latter fails, execution simply continues at microcode for the next expression in the sequence labeled by `L0`, thereby restoring the character position as specified in Equation (not.1).

If execution of expression e in line 3 fails, a double backtrack sequence is initiated. That is, the top backtrack entry that was pushed in lines 1 and 2 is removed in line 4 by incrementing the backtrack stack pointer. Next, a regular backtrack operation is invoked in lines 5 to 9. This mirrors the semantic definition in Equation (not.2).

```
 1      $bsp = $bsp - 1
 2      bs[$bsp] = {$cpos, L0, $rsp}
 3      c(e)
 4      $bsp = $bsp + 1
 5      t0 = bs[$bsp]
 6      $bsp = $bsp + 1
 7      $cpos = t0[0]
 8      $rsp = t0[2]
 9      $pc = t0[1]
10  L0: ...
```

Listing 3.8: Microcode implementation for not-predicate PEG expression !e.

3.3.2.10 Sequence

Microcode for the sequence PEG expression e_1e_2 is shown Listing 3.9 and simply consists of two consecutive microcode blocks of expression e_1 followed by expression e_2. If the microcode of both expressions is executed successfully, program execution continues with the microcode following expression e_2, which complies to the operational semantics defined in Equation (seq.1).

If execution of microcode for expression e_1 fails or if execution of expression e_1 succeeds but that of expression e_2 fails, then the sequence expression as a whole fails, which matches Equation (seq.3) and Equation (seq.2) respectively.

```
    c(e1)
    c(e2)
```

Listing 3.9: Microcode implementation for sequence PEG expression e_1e_2.

3.3.2.11 Prioritized Choice

If the sequence PEG expression e_1e_2 is analogous to an and-operation between execution of expressions e_1 and e_2, then the prioritized choice expression e_1 / e_2 is analogous to an or-operation between execution of expressions e_1 and e_2. That is, the prioritized choice expression only results in a backtrack operation if the microcode for neither expression e_1 nor e_2 can be successfully executed.

Lines 1 and 2 of Listing 3.10 save the current state of the parsing machine, before attempting to execute expression e_1 in line 3. If successful, line 4 removes the entry pushed in lines 1 and 2, and line 5 initiates a jump to the microcode of the next expression in the sequence. This execution path is in accordance with the operational semantics in Equation (prc.1).

If execution of microcode for expression e_1 fails, the backtrack entry pushed in lines 1 and 2 is used to restore the character position and return stack pointer, after which the program resumes execution in line 6 labeled by `L0`. If execution of microcode for expression e_2 then succeeds, the prioritized choice expression still succeeds as defined in Equation (prc.2). If not, another backtrack operation is invoked such that evaluation of the expression as a whole fails, which happens as defined in Equation (prc.3).

```
1     $bsp = $bsp - 1
2     bs[$bsp] = {$cpos, L0, $rsp}
3     c(e1)
4     $bsp = $bsp + 1
5     $pc = L1
6 L0: c(e2)
7 L1: ...
```

Listing 3.10: Microcode implementation for prioritized choice PEG expression e_1 / e_2.

3.3.2.12 Non-terminal

The microcode for a non-terminal PEG expression A_k with production rule $A_k \leftarrow e$ is shown in Listing 3.11. As explained in Section 3.1.2.1, a non-terminal call is functionally equivalent to a function call in conventional architectures. For example, lines 1 and 2 push a new stack entry onto the return stack containing the address of the first micro-operation, which is labeled by `L0`, after the non-terminal call. Line 3 then initiates a jump to the first micro-operation of expression e labeled by `Ak`. Line 6 represents the microcode of expression e, which is pointed to by `ntmap(`G`, `A_k`)` as defined in Equation (3.13).

If execution of the microcode of non-terminal A_k in line 6 succeeds, lines 7 to 9 are executed. These pop the top entry from the return stack and uses it to jump back to the micro-operation labeled by `L0` after the non-terminal call of line 3. The character position advancement resulting from this successful execution matches the behavior detailed in Equation (nte.1).

If execution of the microcode of non-terminal A_k in line 6 fails, the top backtrack entry is used in order to backtrack to some previously saved state. This is in accordance with the operational semantics defined in Equation (nte.2).

```
1      $rsp = $rsp - 1
2      rs[$rsp] = L0
3      $pc = Ak
4 L0: ...
5      ...
6 Ak: c(ntmap(G, Ak))
7      t0 = rs[$rsp]
8      $rsp = $rsp + 1
9      $pc = t0
```

Listing 3.11: Microcode implementation for non-terminal PEG expression A_k.

3.3.2.13 Grammar

In the previous sections the microcode for all PEG expressions were outlined. These can be used to form the microcode of a complete PEG grammar definition, which results in the microcode of Listing 3.12. The microcode for a grammar is divided into two parts: lines 1 to 9 contains the setup code, and lines 10 to 22 contain the non-terminal definitions.

The microcode for any grammar starts by pushing a partially empty backtrack entry apart from an instruction address labeled by `L1`. This is the first backtrack entry, which is only used in case the grammar as whole did not match with the input string. Next, a call to non-terminal A_1 is prepared by pushing the address of the micro-operation after the call onto the return stack pointer in line 2. Line 3 initiates the jump to the microcode associated with non-terminal A_1 starting at line 10.

In the case that the grammar is successfully evaluated with respect to the input string, program execution returns to the microcode starting at label `L0` in line 4. Here, a status flag indicating its failure to parse is reset and a flag indicating its success in parsing the input string is set, after which the parsing machine ends up in an infinite loop in line 6. At the opposite end of the spectrum, if the grammar fails the match with the input string, the bottom backtrack stack entry created in line 1 is used to jump to the address labeled by `L1` in line 7. Here, the parsing machine is instructed to set a status flag indicating its failure to parse the input string and reset the flag indicating a successful parse. Thereafter, the parsing machine still ends up in an infinite loop in line 9. Note that the infinite loops in lines 6 and 9 can only be exited by an external reset of the parsing machine.

The microcode for all non-terminal productions are listed from line 10 onward. This microcode listing assumes a grammar G with n non-terminals A_1, A_2, ..., A_n and their associated production rules. The non-terminals are translated to microcode in no particular order. However, do note that the start non-terminal is represented as non-terminal A_1, which is the first non-terminal that is called in line 3. The microcode for each non-terminal is the same as presented earlier in Listing 3.11: first the microcode for the expression belonging to the respective non-terminal and then three micro-operations that pop the top return stack entry to jump back to after the respective non-terminal call.

The complete microcode mirrors the expected behavior when evaluating `match`$(G, S, s, 0)$ as discussed in Section 3.2.2.13. Here it is assumed that the start symbol S equates to non-terminal A_1 and that the character position register `$cpos` is reset before execution of Listing 3.12 starts.

```
 1      bs[$bsp] = {null, L1, null}
 2      rs[$rsp] = L0
 3      $pc = A1
 4 L0:  fail = 0
 5      success = 1
 6      $pc = $pc
 7 L1:  fail = 1
 8      success = 0
 9      $pc = $pc
10 A1:  c(ntmap(G, A1))
11      t0 = rs[$rsp]
12      $rsp = $rsp + 1
13      $pc = t0
14 A2:  c(ntmap(G, A2))
15      t0 = rs[$rsp]
16      $rsp = $rsp + 1
17      $pc = t0
18      ...
19 An:  c(ntmap(G, An))
20      t0 = rs[$rsp]
21      $rsp = $rsp + 1
22      $pc = t0
```

Listing 3.12: Microcode implementation for an arbitrary PEG grammar.

3.4 PPEG Instruction Set Architecture

So far, the architectural components, formalized PEG behavior, and microcode programs for each PEG expression have been explained in detail in previous section. These are finally combined in this section in order to develop a coherent PPEG instruction set architecture or ISA.

This section consists of three parts. Section 3.4.1 lays out the design principles that form the basis of the PPEG ISA. Next, Section 3.4.2 explains the instruction encoding that is followed by the PPEG instructions. Lastly, Section 3.4.3 defines the instructions and how they fit into the microcode programs from Section 3.3.

3.4.1 ISA Design Principles

In order to get from the microcode programs introduced in Section 3.3 to an ISA, several design principles are needed in order to construct a coherent instruction set architecture. These principles are only guidelines and may reflect conflicting goals. Resolution of these conflicts is based on assumptions that are explained when encountered. Although a better design may be possible with an iterative design-implementation-measurement process, time constraints limit an iterative practice.

As stated in the goals (see Section 1.1), this new PEG parsing machine architecture is developed with the foremost goal to be implemented on hardware. Consequently, the functionality and simplicity core principles, as listed in Section 1.2, of the PPEG architecture and ISA is key and forms the basis for how these design principles were derived.

The following five ISA design principles are used:

1. The logic required for fetching and decoding instructions must be kept simple.
2. An instruction must have its fetch-, decode-, execute-, and write-back-phase finish in a total of 1 clock cycle.
3. Only a single memory read and write operation per instruction is allowed for each individual memory component, which includes stack components.
4. Only contiguous micro-operations can form a single instruction.
5. An instruction must pack as much functionality as possible.

Principle 1 mirrors the simplicity over complexity core design principle as stated in Section 1.2. Although speed and design footprint are not the primary design goals, these can be welcome consequences by reducing the complexity of both the fetching and decoding process.

Principle 2 is a consequence of the simplicity core design principle. Although a pipelined architecture may be realized to gain speed, the increase of complexity is unwarranted for this architecture. Moreover, as will become apparent later, the number of jumps in parsing machine programs due to backtracks and non-terminal calls make it much harder to reap the benefits of a simple pipeline due to the excessive number of required flushes.

The reason for inclusion of Principle 3 is because of a combination of hardware limitations and an extension of Principle 2. The hardware that is used to implement this parsing machine architecture on has internal memory components which in turn have synchronous write ports. That is, these components allow only a single write per component per clock cycle. The result is that a read-write-read operation cannot occur in a single clock cycle.

Principle 4 is based on the microcode definitions of each PEG expression listed in Section 3.3. The micro-operations have been written sequentially, as the parsing machine is expected to execute these operations in order. For this reason, when grouping the micro-operations from these listings, only contiguous operations may form a single PPEG instruction.

Principle 5 may appear counter to the core simplicity principle, but is meant as guideline to be followed only when the other design principles are adhered to. This principle only serves to prevent the creation of more instructions than is absolutely necessary.

3.4.2 Instruction Encoding

The instruction encoding format is core of any instruction set architecture, as it determines the complexity of the fetching and decoding units in the architecture. Therefore, Principle 1 drives the choices hereafter behind the instruction encoding formats for the PPEG architecture.

First, in order to facilitate a simple instruction fetching process, it is determined that all instructions must have the same length. Although variable length instructions might result in more efficient storage of instructions in memory and would enable the existence for long instructions with many operands, it would certainly require more complex logic. For example, instruction alignment and multi-cycle fetch phases are problems that would arise if variable-length instructions are realized.

Second, extending the previous point, it is determined that PPEG instructions must all be 32 bits in length. The reason for this seemingly arbitrary number is mostly due to the careful balance between functionality and memory cost saving. Specifically, some instructions will need to store an absolute instruction address for a branching operation. If those instructions have few bits for an address, the number of total instructions in a compiled PEG is severely limited and therefore the length of the PEG itself. If instructions are too long, however, many bits are wasted by instructions that have few or no operands at all. Furthermore, it is common for memory components to have data bus widths of a power-of-two number of bits. For these reasons, the choice for 24 bits is chosen as address length. This allows the PPEG programs to at most consist of $2^{24} \approx 16.8 \times 10^6$ instructions, or 2^{24} instructions $\times\, 32\frac{\text{bits}}{\text{instruction}}/8\frac{\text{bits}}{\text{B}} = 64$ MiB of instruction memory.

Finally, three distinct types of instruction encoding format are introduced: E-type (Section 3.4.2.1), I-type (Section 3.4.2.2), and J-type (Section 3.4.2.3). The two most significant bits are used to identify the format of any instruction, i.e., the binary sequence 00 identifies the E-type, 01 the I-type, and 10 the J-type. Bits 27 up to 29 that precede the two format bits identify the function of an instruction, thereby uniquely identifying an instruction within a specific encoding format. Together these 5 bits represent the opcode of an instruction. These bits are used by the decoder to determine which instruction is currently executing and what values its output control lines should have.

3.4.2.1 E-Type

The E-type format or empty format is one that contains no operand fields. Only the mandatory five opcode bits contain useful information. The instructions that use this format either do not operate on any data or the data is implicitly specified in the instruction.

31 30	29 27	26 0
`00`	`func`	`-`

3.4.2.2 I-Type

The I-type format or immediate format consists of three separate 8-bit fields. The first eight bits are to be used for any instruction that requires an address offset operand, such as used for relative jumps. The two consecutive 8-bit fields can be used for storing any immediate value. Note too that there are three bits (bit 24 to 26) left between opcode and operand fields which are reserved.

31 30	29 27	26 24	23 16	15 8	7 0
`01`	`func`	`-`	`immediate1`	`immediate0`	`offset`

3.4.2.3 J-Type

The J-type format or jump format consists of a single 24-bit address field. As explained earlier in this section, this is to be used by instructions that require an absolute instruction address for direct or future jumps in program execution.

31 30	29 27	26 24	23 0
`10`	`func`	`-`	`address`

3.4.3 ISA Definition

With all ISA design principles and instruction encoding information introduced in the previous sections, this section provides the complete definition and explanation of the 13 PPEG instructions. Their definition is threefold. First, each instruction has its behavior defined in terms of micro-operations explored in Section 3.3. Second, a logic diagram is presented that shows the logic required to implement the instruction in digital hardware. Finally, an inductive definition is provided that defines the instruction in terms of a change in parsing machine state.

The inductive instruction definition follows the notation introduced earlier in Section 3.2. Above the horizontal line are the premises, and, if they all hold true, the conclusion below the line then holds true too. The first premise and the conclusion are the initial and next parsing machine state respectively; the second premise is always the instruction that is defined; and all other premises (if any) are additional conditions for a certain conclusion to hold true.

The parsing machine state $\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle$ is declared as $\mathbb{N} \times \mathbb{N} \times \mathcal{R}^* \times \mathcal{B}^* \times \mathcal{T}$. The first two natural numbers represent the current program counter value pc and character position value cp respectively. The return stack rs is declared as $\mathcal{R}^*$, which represents the set of possible return stack entry sequences. Each entry is notated as a single-element tuple consisting of a natural number (pc_r), which represents the return address pc_r associated with that entry. The backtrack stack bs is declared as $\mathcal{B}^*$, which represents the set of possible backtrack stack entry sequences. Each entry is notated as a 3-element tuple (pc_b, cp_b, rs_b) declared as $\mathbb{N} \times \mathbb{N} \times \mathcal{R}^*$, which consists of a program counter and character position value and a return stack. Finally, the state variable t is declared as $\mathcal{T} = \{-1, 0, 1\}$. This variable represents the status of the parsing machine, i.e., $t = -1$ translates to “failed parse” status, $t = 1$ translates to “successful parse” status, and $t = 0$ translates to “active parse” status. Before the parsing machine starts execution, it is assumed that this state variable is externally reset to $t = 0$.

Note that the parsing machine state does not include the return stack and backtrack stack pointers. Instead, these are implicit to the return stack and backtrack stack entry sequences rs and bs, as it would otherwise be much more difficult to read. A consequence of this decision is that, instead of explicitly storing only a return stack pointer in a backtrack stack entry, now the total return stack entry sequence is seemingly stored in such an entry. In practice, however, still only the return stack entry is stored.

Lastly, one final comment about the written-form of the instruction operands. If a PPEG instruction uses the character position register `$cpos` in any way, it is listed as an operand for that instruction, in contrast to any of the other status registers. The reason for this is that, in the context of parsing text, only a change in character position is assumed to be of interest to the user. To this end, the format of the character position register as operand reflects the type of operation performed on that register.

The following subsections define the PPEG instruction set architecture.

3.4.3.1 `set_fail`

As the name implies, the function of the `set_fail` instruction is to set the parsing machine status to indicate failure. Its microcode definition in Listing 3.13 is the result of grouping lines 4 to 7 in the microcode for a PEG grammar in Listing 3.12. The choice for grouping these specific lines into an instruction is based on design principles 4 and

5: the micro-operation before these lines has unrelated behavior and micro-operations after these lines cannot be executed due to the infinite loop before them.

```
fail = 1
success = 0
$pc = $pc
```

Listing 3.13: Microcode for the `set_fail` instruction.

The instruction is implemented in terms of digital logic as is shown in Figure 3.3. The only interesting part is the setting and setting of the "fail" and "success" blocks in the logic diagram, which each represent a single bit register or flip-flop that can be set, reset, or left unchanged.

The inductive definition of the `set_fail` instruction is presented in Equation (set_fail.1). Note that the only state change is the change in value of the parsing machine's status to $t = -1$, representing a failed parse, which is indeed the intended function of this instruction.

Finally, as can be seen in the second premise of its inductive definition, the written instruction format has no apparent operand, such that the instruction encoding follows the E-type format (see Section 3.4.2.1). The instruction's opcode consists of the following binary format and function values: `fmt = 00`, `func = 001`.

$$\frac{\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \qquad \texttt{imem}[pc] = \texttt{set\_fail}}{\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ -1\rangle} \tag{set\_fail.1}$$

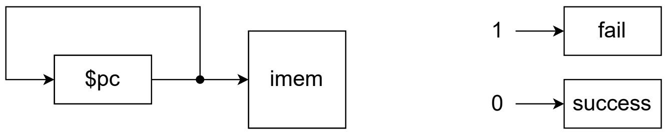


Figure 3.3: Logic diagram for the `set_fail` instruction.

3.4.3.2 `set_success`

The instruction `set_success` is almost entirely the same as `set_fail`, except for indicating a successful parse as opposed to a failed parse. Its microcode definition in Listing 3.14 is result of grouping lines 7 to 9 of Listing 3.12, which is based on entirely the same principles and reasons as outlined in Section 3.4.3.1.

```
fail = 0
success = 1
$pc = $pc
```

Listing 3.14: Microcode for the `set_success` instruction.

The digital logic diagram in Figure 3.4 and inductive definition in Equation (set_success.1) are also extremely similar to those of the `set_fail` instruction, with only the value for the status flags reversed.

The `set_success` instruction also follows the E-type instruction encoding format (see Section 3.4.2.1). Moreover, the instruction's opcode consists of the following binary format and function values: `fmt = 00`, `func = 010`.

$$\frac{\begin{array}{c}\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{set\_fail}\end{array}}{\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ 1\rangle} \tag{set\_success.1}$$

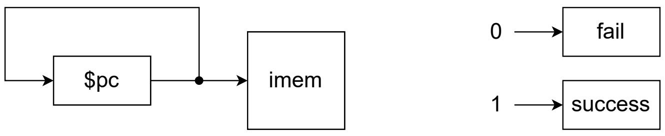


Figure 3.4: Logic diagram for the `set_success` instruction.

3.4.3.3 `ret`

The `ret` instructions is used as a successful return from non-terminal call operation, which is achieved by jumping to the return address at the top of the return stack. Its definition in terms of microcode is shown in Listing 3.5 and is the result of grouping together lines 7 to 9 from the non-terminal PEG expression microcode in Listing 3.11. The reason for grouping these micro-operations is again a consequence of principles 4 and 5: unlike these grouped micro-operations, the microcode preceding these operations are independent and unrelated.

```
t0 = rs[$rsp]
$rsp = $rsp + 1
$pc = t0
```

Listing 3.15: Microcode for the `ret` instruction.

The microcode definition translates into the digital logic seen in Figure 3.5. Here, it can be observed that the current return stack pointer addresses the return stack for the top return address, which is then directly fed into the program counter register. Moreover, the return stack pointer register is increased by 1 by means of a adder component.

Equation (ret.1) presents the inductive definition of the parsing machine state translation for the `ret` instruction. Note that the first premise presents the return stack entry as $(pc_r) : rs$, which is a concatenation of the top return stack entry with return address pc_r and the rest of the return stack rs. If the two premises hold, the parsing machine state changes, such that the new program counter value is equal to pc_r and the return stack is now only equal to rs, which is the result of popping the top entry.

Similar to the previous two instructions, the written format of the `ret` instruction has no operands as shown in the second premise of Equation (ret.1). This instruction thus uses the E-type instruction encoding format (see Section 3.4.2.1), whose opcode consists of the following binary format and function values: `fmt = 00` , `func = 011`.

$$\frac{\langle pc\,,\ cp\,,\ (pc_r) : rs\,,\ bs\,,\ t\rangle \qquad \texttt{imem}[pc] = \texttt{ret}}{\langle pc_r\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle} \tag{ret.1}$$

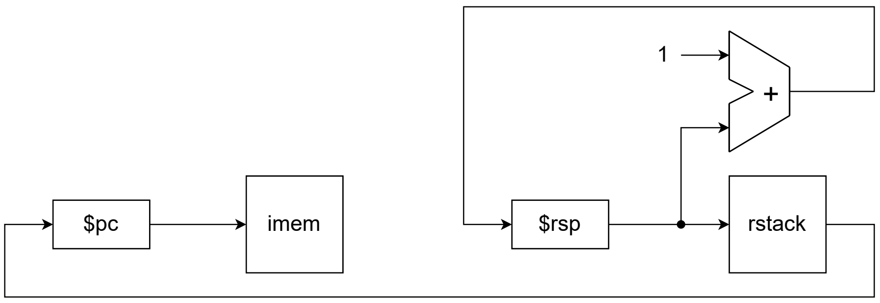


Figure 3.5: Logic diagram for the `ret` instruction.

3.4.3.4 `any`

As the name suggests, the `any` instruction is a one-to-one mapping of the any PEG expression, thereby able to correctly parse any character as long as the character position is within the bounds of the input string length. Its microcode in Listing 3.16 is the result of grouping all lines of the any expression microcode in Listing 3.3, which is a direct result of applying Principle 5: packing as much functionality in a single instruction.

```
 1 if ($cpos == $fs) {
 2     t0 = bs[$bsp]
 3     $bsp = $bsp + 1
 4     $cpos = t0[0]
 5     $rsp = t0[2]
 6     $pc = t0[1]
 7 }
 8 else {
 9     $cpos = $cpos + 1
10     $pc = $pc + 1
11 }
```

Listing 3.16: Microcode for the `any` instruction.

Despite the relatively many micro-operations in its microcode, the `any` instruction should still be able to execute in a single single clock-cycle as stated by Principle 2. This is achieved by the digital logic presented in Figure 3.6. It should be observed that the values on the right hand-side of all micro-operations in Listing 3.16 are all obtained in parallel by this logic. However, the writing these values depends on the comparison in line 1. This comparison is manifests itself into a equal-or-greater-than comparator, which takes the current character position and file size values as input. The single bit result is then fed into four multiplexers, which route the correct values to the input of their respective registers.

The parsing machine state changes are inductively defined in Equation (any.1) and Equation (any.2), where the former present the state change for a successful character parse and the latter for a backtrack operation after a failed character parse. If successful, both the program counter and character position registers are incremented as in Equation (any.1). If unsuccessful, the top backtrack stack entry (cp_b, pc_b, rs_b) is popped and the program counter, character position, and return stack are set to the values within, as is shown in Equation (any.2).

Unlike the previously defined instructions, the `any` instruction does have an operand, namely the `$cpos` register. However, as explained in the introduction of this section, the reason for this is because the `any` instruction operates on `$cpos` register. Still, no other operands are used by the instruction and therefore it too is encoded with the E-type encoding format (see Section 3.4.2.1). The associated opcode consists of the following binary format and function values: `fmt = 00`, `func = 100`.

$$\frac{\begin{array}{l}\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{any \$cpos} \\ cp < fs\end{array}}{\langle pc+1\,,\ cp+1\,,\ rs\,,\ bs\,,\ t\rangle} \tag{any.1}$$

$$\frac{\begin{array}{l}\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{any \$cpos} \\ cp \geq fs\end{array}}{\langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle} \tag{any.2}$$

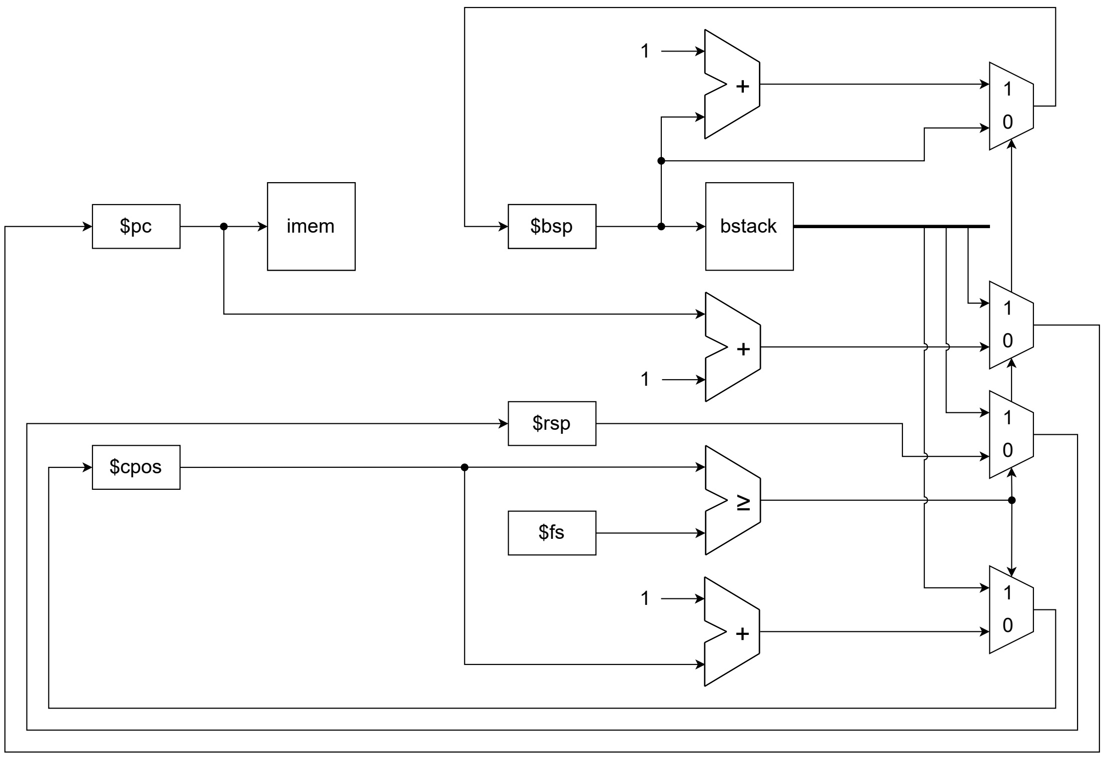


Figure 3.6: Logic diagram for the `any` instruction.

3.4.3.5 `s_bktr`

As can be observed in the microcode definition for the `s_bktr` instruction in Listing 3.17, this instruction simply performs a single backtrack operation. These micro-operations together are a common occurrence in microcode programs for PEG expressions, but can often be combined with other micro-operations to create another instruction, such as the case of the `any` instruction. However, this is not the case for the same group of micro-operations in lines 8 to 12 of Listing 3.7 for the and-predicate PEG expression, where the preceding micro-operation is a jump and the micro-operation after is independent of these micro-operations. The five micro-operations in Listing 3.17 therefore pack the maximum functionality that is needed according to Principle 5.

```
t0 = bs[$bsp]
$bsp = $bsp + 1
$cpos = t0[0]
$rsp = t0[2]
$pc = t0[1]
```

Listing 3.17: Microcode for the `s_bktr` instruction.

The logic that is required to implement the `s_bktr` instruction is presented in Figure 3.7. It simply shows that the backtrack stack pointer addresses the top backtrack stack entry, whose three values are fed into the character position, program counter, and return stack pointer registers. At the same time, the backtrack stack pointer itself is incremented and fed back into itself.

This behavior can also be observed in the inductive definition shown in Equation (s_bktr.1), which is the same as Equation (any.2), except for the removal of the character position condition as a third premise.

Finally, as the backtrack instruction also operates on the character position register, it is included as only operand in the written format of said instruction. The instruction encoding is therefore also of the E-type format (see Section 3.4.2.1), with the following binary format and function values: `fmt = 00`, `func = 101`.

$$\frac{\begin{array}{l}\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{s\_bktr \$cpos}\end{array}}{\langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle} \tag{s\_bktr.1}$$

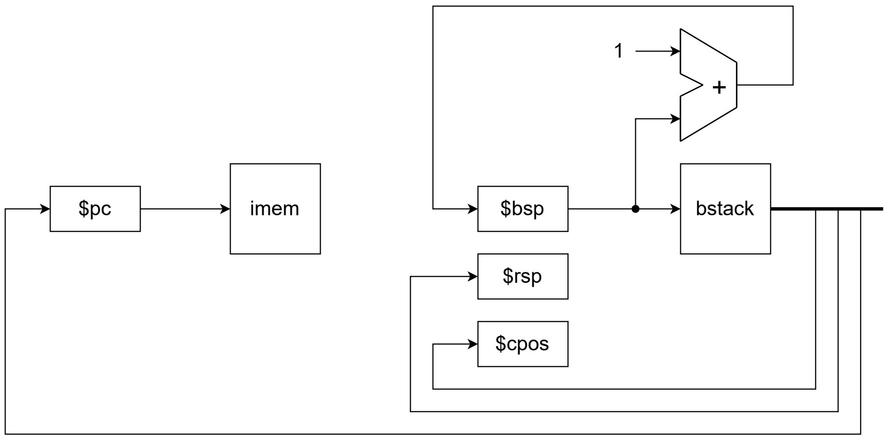


Figure 3.7: Logic diagram for the `s_bktr` instruction.

3.4.3.6 `s_rmv_bktr`

Listing 3.18 shows the microcode of the `s_rmv_bktr` instruction, which adds only the first micro-operation with respect to the microcode definition of the `s_bktr` instruction. This is apparent from the name of the `s_rmv_bktr` instruction, which combines the removal of the top backtrack stack entry (line 1) with a succeeding backtrack operation (lines 2 to 6). Its use can be observed in lines 4 to 9 of microcode Listing 3.8, which defines the microcode for a not-predicate PEG expression. Once again, any surrounding microcode is independent of these micro-operations, which means that, by definition of Design Principle 5, the functionality of the `s_rmv_bktr` instruction is maximally packed.

```
1 $bsp = $bsp + 1
2 t0 = bs[$bsp]
3 $bsp = $bsp + 1
4 $cpos = t0[0]
```

```
$rsp = t0[2]
$pc = t0[1]
```

Listing 3.18: Microcode for the `s_rmv_bktr` instruction.

The difference in logic for the `s_rmv_bktr` and `s_bktr` instructions is reflected by the insertion of an additional adder between the backtrack stack pointer register and the backtrack stack component, as shown in Figure 3.8. This additional adder is used to directly address the second backtrack stack entry from the top.

Dismissal of the top backtrack stack entry can also clearly be observed in the parsing machine state change inductively defined in Equation (s_rmv_bktr.1) for the `s_rmv_bktr` instruction. Instead of attaining the values contained in the top backtrack stack entry $(cp_{b1}, pc_{b1}, rs_{b1})$, the state machine changes its state variables to the values contained in the second entry from the top.

The `s_rmv_bktr` has only the character position register `$cpos` as operand, meaning that it too is encoded according to the E-type encoding format (see Section 3.4.2.1). Its opcode consists of the following binary format and function values: `fmt = 00`, `func = 101`.

$$\frac{\langle pc\,,\ cp\,,\ rs\,,\ (cp_{b1}, pc_{b1}, rs_{b1}) : (cp_{b2}, pc_{b2}, rs_{b2}) : bs\,,\ t\rangle \quad \texttt{imem}[pc] = \texttt{s\_rmv\_bktr \$cpos}}{\langle pc_{b2}\,,\ cp_{b2}\,,\ rs_{b2}\,,\ bs\,,\ t\rangle} \tag{s\_rmv\_bktr.1}$$

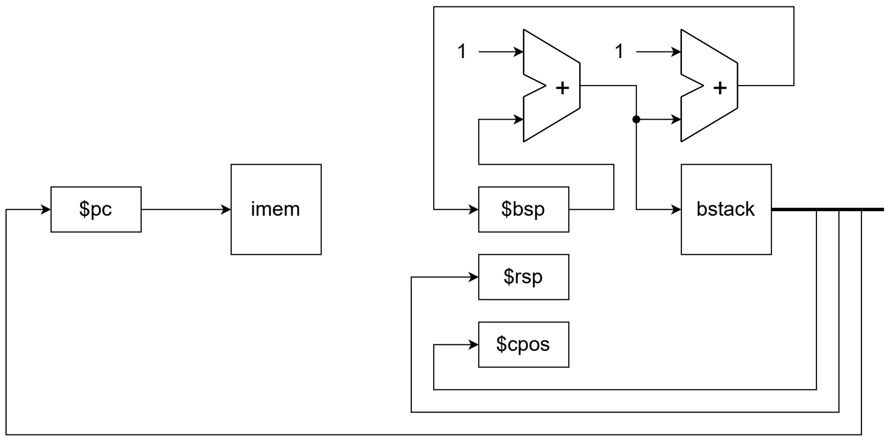


Figure 3.8: Logic diagram for the `s_rmv_bktr` instruction.

3.4.3.7 `char_set`

The microcode for parsing a single character is covered in Section 3.2.2.2 and microcode for the `char_set` instruction, as presented in Listing 3.19, is therefore almost a direct copy. The only difference is in line 9, which consists of two comparisons, effectively checking if the current character is between character range $[c_1\text{-}c_2]$. This type of comparison was already proposed in the microcode section for the character class PEG expression, Section 3.2.2.3. By using this comparison, the `char_set` instruction can be used to both parse a single character (i.e., $c_1 = c_2$) and a single character range (i.e., [c_1-c_2]).

The reason for substituting the simple comparison for equality into a range comparison is because it enables fast evaluation of literal strings and single range character classes alike, thereby packing more functionality in a single instruction (see Principle 5.

```
if ($cpos >= $fs) {
    t0 = bs[$bsp]
    $bsp = $bsp + 1
    $cpos = t0[0]
    $rsp = t0[2]
    $pc = t0[1]
}
else {
    if (dmem[$cpos] >= c1 && dmem[$cpos] <= c2) {
        $cpos = $cpos + 1
        $pc = $pc + 1
    }
    else {
        t1 = bs[$bsp]
        $bsp = $bsp + 1
        $cpos = t1[0]
        $rsp = t1[2]
        $pc = t1[1]
    }
}
```

Listing 3.19: Microcode for the `char_set` instruction.

Regardless of the complex microcode for a single instruction, the `char_set` instruction can still be executed in single cycle according to Principle 2. The hardware that achieves this feat is presented in Figure 3.9. On the left, the `$pc` and `$cpos` registers are used to fetch the current instruction and character respectively. At the same time, the top entry is fetched from the backtrack stack is fetched in case a backtrack operation is necessary. Next, the two opcodes c_1 and c_2 are compared against the active character and the `$cpos` register is compared against the file size register `$fs`. Finally, the two new possible values for each register are fed through multiplexers, which routes only one to their respective register based on the comparison results (output of the and-gate).

The microcode paths for instruction `char_set` are formally defined in Equation (char_set.1), Equation (char_set.2), and Equation (char_set.3). The first represents the only path to successful evaluation of a single character literal string, which results in a simple increment of the program counter and character position registers. The latter two equations define two different conditions with the same outcome, namely a backtrack operation. The cause is either that the active character is not in the specified range or that the character position is outside the input string bounds.

As stated earlier, the `char_set` instruction has at least two operands: the outer range character values c_1 and c_2. However, because this instruction also operates on the character position (albeit conditionally), the operand `($cpos)+` is used. This specific format is used to indicate that the register is used to index data memory, after which the register contents are automatically incremented.

The need to store two 8-bit immediate operands in a 32-bit instruction is satisfied by the immediate or I-type instruction encoding format as seen in Section 3.4.2.2. Note that the lower 8-bits are unused and that `immediate0` $= c_1$ and `immediate1` $= c_2$. Finally, the opcode associated with the `char_set` instruction consists of the following binary format and function values: `fmt = 01`, `func = 000`.

$$\frac{\begin{array}{l}\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{char\_set (\$cpos)+,}\ c_1\texttt{,}\ c_2 \\ cp < fs \\ c_1 \leq \texttt{dmem}[cp] \leq c_2\end{array}}{\langle pc+1\,,\ cp+1\,,\ rs\,,\ bs\,,\ t\rangle} \qquad \text{(char\_set.1)}$$

$$\frac{\begin{array}{l}\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{char\_set (\$cpos)+,}\ c_1\texttt{,}\ c_2 \\ cp < fs \\ (\texttt{dmem}[cp] < c_1) \vee (\texttt{dmem}[cp] > c_2)\end{array}}{\langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle} \qquad \text{(char\_set.2)}$$

$$\frac{\begin{array}{l}\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{char\_set (\$cpos)+,}\ c_1\texttt{,}\ c_2 \\ cp \geq fs\end{array}}{\langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle} \qquad \text{(char\_set.3)}$$

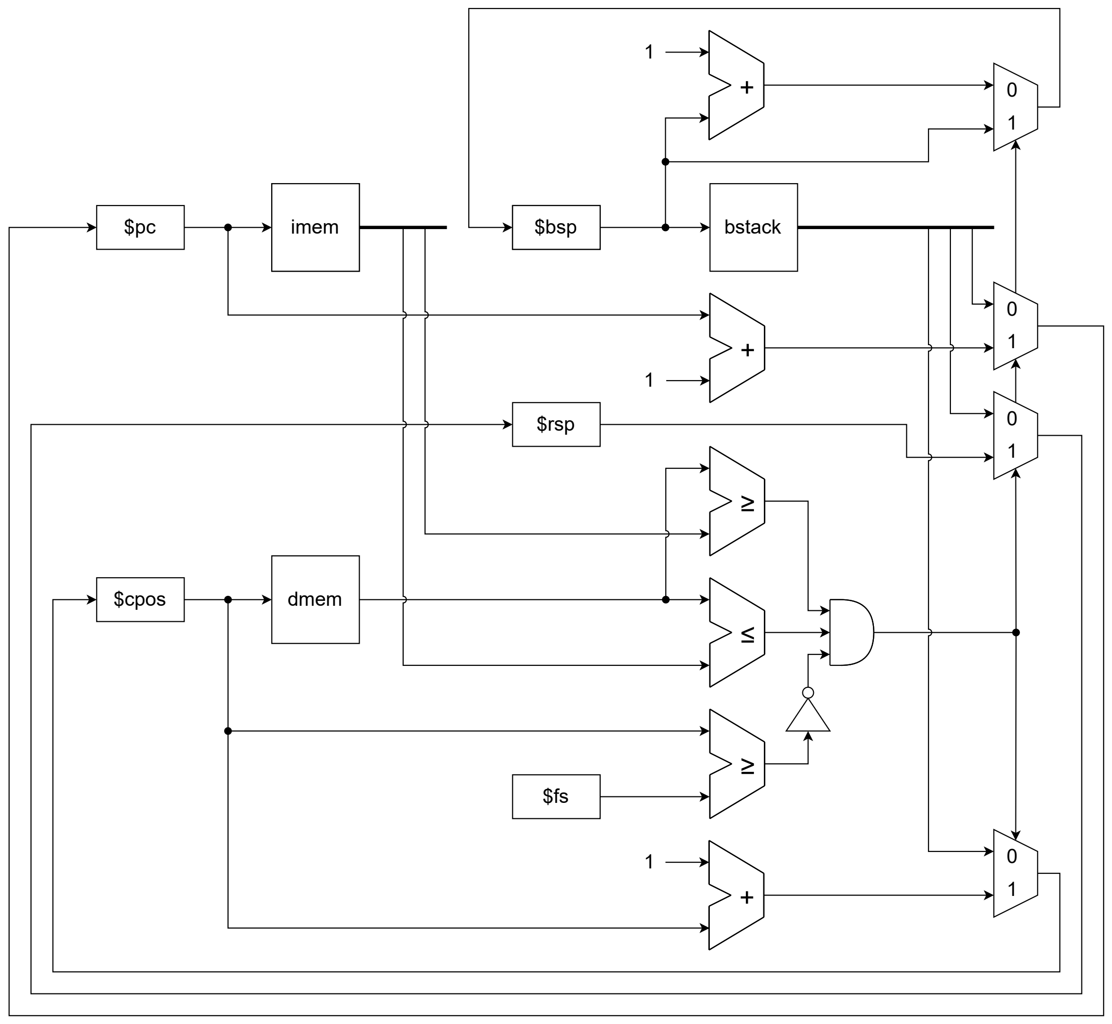


Figure 3.9: Logic diagram for the `char_set` instruction.

3.4.3.8 `char_set_beq`

Microcode for the `char_set_beq` instruction, as presented in Listing 3.20, is similar to the microcode definition of `char_set` (See Listing 3.19). The main difference is the code for the condition when the character position is inside the input string bounds but the active character is outside the specified character range [c_1-c_2]. Instead of initiating a backtrack operation, the program counter is simply incremented by one, which can be observed in line 14. The other difference is in line 11, which shows that the program counter is increased by an offset o if the active character is within the character range [c_1-c_2]. This is the reason for the suffix **`-beq`**, meaning branch or jump on equality.

```
1 if ($cpos >= $fs) {
2     t0 = bs[$bsp]
3     $bsp = $bsp + 1
4     $cpos = t0[0]
```

```
    $rsp = t0[2]
    $pc = t0[1]
}
else {
    if (dmem[$cpos] >= c1 && dmem[$cpos] <= c2) {
        $cpos = $cpos + 1
        $pc = $pc + o
    }
    else {
        $pc = $pc + 1
    }
}
```

Listing 3.20: Microcode for the char_set_beq instruction.

This instruction is specifically designed for evaluation of character class PEG expressions. Listing 3.21 shows an example of how the alphanumeric character class expression [0-9A-Za-z] can be evaluated using a combination of char_set_beq and s_bktr instructions. The written instruction format is similar to that of the char_set instruction, only adding an additional fourth operand representing an instruction address offset. The code can be read as follows. Assuming execution of line 1, if the active character is not between '0' and '9', the program counter is incremented by one; otherwise the program counter is increased by 4, thereby jumping to the instruction after line 4. Any time the active character does not fit within the specified range, this process is repeated for any char_set_beq instructions thereafter. Finally, if the active character does not match with any of the three ranges compared in lines 1 to 3, the parsing machine ends up in line 4, where the s_bktr instruction initiates a backtrack operation (see Section 3.4.3.5) in order to fail evaluation of the complete character class expression. A more detailed account of the character class expression translation is provided in Section 3.5.2.2.

```
1 char_set_beq ($cpos)+, '0', '9', 4
2 char_set_beq ($cpos)+, 'A', 'Z', 3
3 char_set_beq ($cpos)+, 'a', 'z', 2
4 s_bktr $cpos
```

Listing 3.21: PPEG program for alphanumeric character class PEG expression.

Note that there is no direct translation from the microcode for the character class PEG expression in Listing 3.2 and the microcode definition of the char_set_beq instruction. Although Principle 5 states that as much functionality must be packed in any instruction, an instruction that could parse any character class expression requires many more operands than can be stored in a 32-bit instruction. This could possibly be circumvented by creating a multi-cycle instruction, but this would go against both Principle 1 and Principle 2. For this reason, the character class PEG expression is split up into one or more character ranges that can each be parsed with only a single instruction.

The microcode definition of the char_set_beq instruction translates to the digital logic domain as presented in Figure 3.10. Compared to the logic diagram for the char_set instruction, only a few additions have been made. As can be observed in microcode Listing 3.20, there are three distinct actions for each of the three conditions. The significant change compared to the logic in Figure 3.9 is the additional possible values for the character position and program counter, as the former could now remain unchanged and the latter could be incremented by an arbitrary 8-bit offset stored in the 32-bit instruction word. These optional values are routed through two additional multiplexers.

The formal inductive definitions of the `char_set_beq` instruction can be observed in Equation (char_set_beq.1), Equation (char_set_beq.2), and Equation (char_set_beq.3). Compared to the inductive definition of the `char_set` instruction, only the outcome of the conditions presented in Equation (char_set_beq.1) and Equation (char_set_beq.2) are different.

The written format of the `char_set_beq` instruction that can be seen in the inductive definitions is also similar, except for the additional 8-bit offset as fourth operand. The I-type instruction encoding format (see Section 3.4.2.2) is used to store the two 8-bit character values and the 8-bit offset in the 32-bit instruction word. The 5-bit instruction opcode consists of the following binary format and function values: `fmt = 01`, `func = 001`.

$$\frac{\begin{array}{l}\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{char\_set\_beq (\$cpos)+,}\ c_1\,,\ c_2\,,\ o \\ cp < fs \\ c_1 \le \texttt{dmem}[cp] \le c_2\end{array}}{\langle pc + o\,,\ cp + 1\,,\ rs\,,\ bs\,,\ t\rangle} \qquad \text{(char\_set\_beq.1)}$$

$$\frac{\begin{array}{l}\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{char\_set\_beq (\$cpos)+,}\ c_1\,,\ c_2\,,\ o \\ cp < fs \\ (\texttt{dmem}[cp] < c_1) \lor (\texttt{dmem}[cp] > c_2)\end{array}}{\langle pc + 1\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle} \qquad \text{(char\_set\_beq.2)}$$

$$\frac{\begin{array}{l}\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{char\_set\_beq (\$cpos)+,}\ c_1\,,\ c_2\,,\ o \\ cp \ge fs\end{array}}{\langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle} \qquad \text{(char\_set\_beq.3)}$$

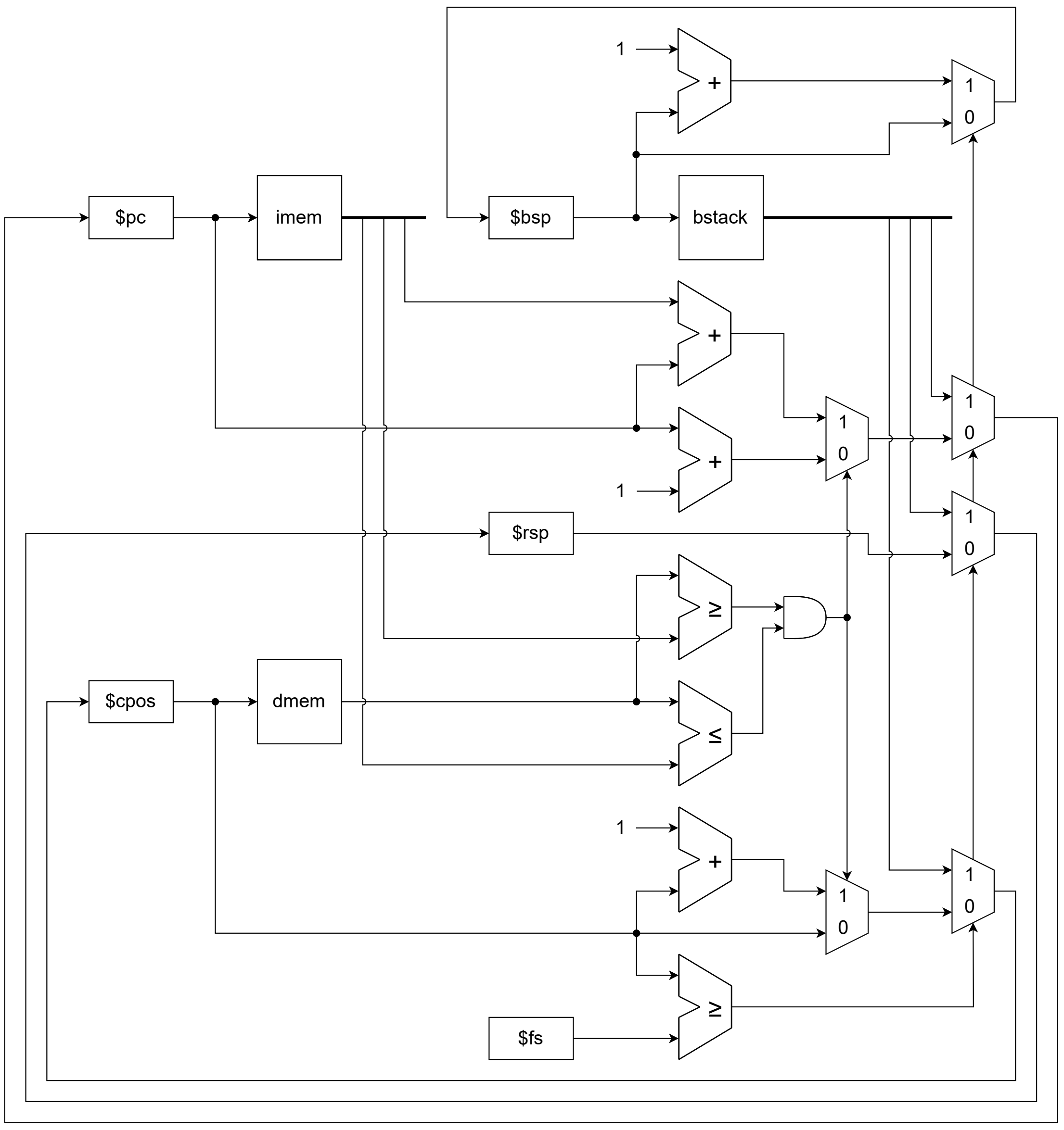


Figure 3.10: Logic diagram for the `char_set_beq` instruction.

3.4.3.9 `call`

The `call` instruction is used to initiate a non-terminal call and is therefore part of the non-terminal PEG expression. The `call` microcode in Listing 3.22 is a grouping (following principles 4 and 5) of the first three lines of the non-terminal expression microcode presented in Listing 3.11. Lines 1 and 2 push the return address (address of the next instruction) onto the return stack, after which the program counter is set to instruction address pc_j in line 3. Note that the address pc_j is intended to be only the first instruction of a non-terminal subroutine, but can in practice be any instruction address.

```
$rsp = $rsp - 1
rs[$rsp] = pc + 1
$pc = pc_j
```

Listing 3.22: Microcode for the `call` instruction.

Figure 3.11 presents the digital logic associated with the microcode of the `call` instruction. Note the direct connection from instruction memory `imem` to the program counter register `$pc`, which represents the 24-bit address that is directly written to the program counter register. The other part of the logic is the data transfer of `$pc + 1` to the return stack at address `$rsp - 1`, the new top stack entry address.

The translation from logic diagram behavior to the formal definition of the `cal` instruction is shown in Equation (call.1). Here, the conclusive parsing machine state has the program counter value set to the jump address operand pc_j. Moreover, the original return stack sequence rs is concatenated with the incremented program counter value, representing the return address.

The `call` instruction is the first instruction in this section to use the J-type instruction encoding, as introduced in Section 3.4.2.3. This instruction format type most prominently features the single 24-bit address operand, which stores the first instruction of the called non-terminal subroutine, or in other words the jump address pc_j. The binary format and function values of the `call` instruction's opcode field are as follows: `fmt = 10`, `func = 000`.

$$\frac{\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \qquad \texttt{imem}[pc] = \texttt{call}\ pc_j}{\langle pc_j\,,\ cp\,,\ (pc+1) : rs\,,\ bs\,,\ t\rangle} \tag{call.1}$$

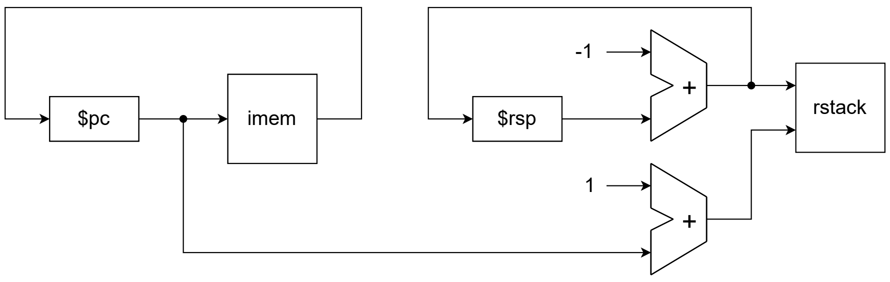


Figure 3.11: Logic diagram for the `call` instruction.

3.4.3.10 `s_push`

As the name implies and most similar to the `call` instruction, the `s_push` instruction pushes a new stack entry onto the backtrack stack. Listing 3.23 presents the microcode associated with this instruction, which contains only three micro-operations: two to push the new entry, consisting of the current character position and return stack pointer values and an absolute backtrack instruction address `pc_b`; the last operation simply increments the program counter by one.

```
$bsp = $bsp - 1
bs[$bsp] = {$cpos, pc_b, $rsp}
$pc = $pc + 1
```

Listing 3.23: Microcode for the `s_push` instruction.

The grouping of these particular micro-operations is no coincident, as they can be found at the start of the microcode for the optional, zero-or-more, and-predicate, not-predicate, and sequence PEG expressions (see Section 3.3). Because of principles 4 and 5, only these micro-operations have been grouped.

The digital logic required for the execution of the `s_push` instruction can be seen in Figure 3.12. Note that the new backtrack stack entry is created from values coming out of the character position register, return stack register, and the 32-bit instruction word.

The inductive definition of the `s_push` instruction can be found in Equation (s_push.1). The main change in parsing machine state is the concatenation of the new backtrack stack entry.

The only information besides the opcode that the `s_push` instruction needs to provide is the backtrack instruction address pc_b, or in other words the address that the parsing machine jumps to in case of a backtrack operation. However, because the instruction also stores the current character position in the backtrack stack, the character position register is the other operand of the `s_push` instruction. In conclusion, this instruction uses the J-type encoding format (see Section 3.4.2.3), whose opcode consists of the following binary format and function values: `fmt = 10`, `func = 100`.

$$\frac{\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \qquad \texttt{imem}[pc] = \texttt{s\_push \$cpos,}\ pc_b}{\langle pc+1\,,\ cp\,,\ rs\,,\ (cp, pc_b, rs) : bs\,,\ t\rangle} \tag{s_push.1}$$

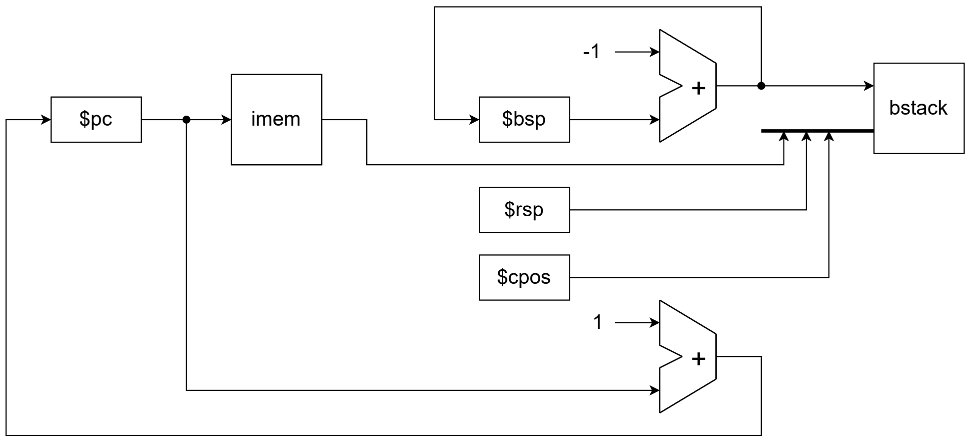


Figure 3.12: Logic diagram for the `s_push` instruction.

3.4.3.11 `s_upd`

The `s_upd` instruction, whose microcode is presented in Listing 3.24, is a contraction of the `s_rmv` and `s_push` instructions. Though the result is the same, instead of explicitly removing the top backtrack stack entry and pushing a new one, the `s_push` fetches the top stack entry, updates the character position and return stack pointer fields to the current values, and writes the updated entry to the same backtrack stack address. This process can be observed in lines 1 and 2 of Listing 3.24. Lastly, line 3 lets the parsing machine jump to an absolute instruction address pc_j.

The microcode definition of this instruction is the result of grouping lines 4 to 6 of microcode Listing 3.5 for the one-or-more PEG expression. The reason for grouping these lines are Principle 4 and Principle 5, as surrounding microcode is independent of the this grouping. Note that the reading from and writing to the same stack entry is allowed according to Principle 3.

```
1 t0 = bs[$bsp]
2 bs[$bsp] = {$cpos, t0[1], $rsp}
3 $pc = pc_j
```

Listing 3.24: Microcode for the `s_upd` instruction.

Figure 3.13 shows the digital logic required in order to execute the `s_upd` instruction. The backtrack stack component is addressed by the unchanging backtrack stack pointer register, such that the read and write addresses are the same. Note also that only the instruction address field of the top backtrack stack entry remains unchanged when writing the new entry.

The `s_upd` instruction behavior is formalized in Equation (s_upd.1). Here, the original top backtrack entry is changed into (cp, pc_b, rs). Moreover, the program counter value is set to the last instruction operand pc_j.

Besides the `$cpos` operand, the `s_upd` has an absolute instruction address pc_j as operand which requires the instruction to be encoded with the J-type instruction encoding format (see Section 3.4.2.3. The opcode field of the same instruction consists of the following binary format and function values: `fmt = 10`, `func = 101`.

$$\frac{\begin{array}{c}\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{s\_upd \$cpos,}\ pc_j\end{array}}{\langle pc_j\,,\ cp\,,\ rs\,,\ (cp, pc_b, rs) : bs\,,\ t\rangle} \tag{s\_upd.1}$$

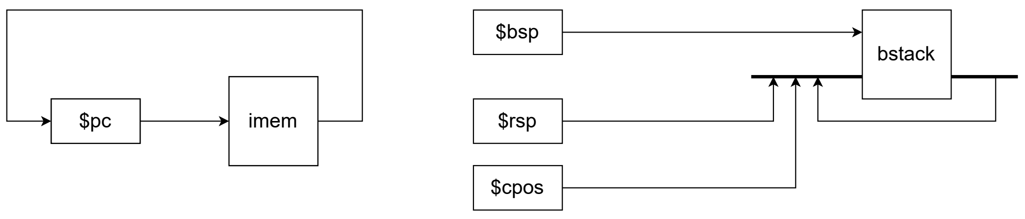


Figure 3.13: Logic diagram for the `s_upd` instruction.

3.4.3.12 `s_res`

The `s_res` instruction, unlike the `s_bktr` instruction, does not initiate a complete backtrack operation, but only uses the top backtrack stack entry to restore the character position register to some saved value. This is achieved by the microcode implementation shown in Listing 3.25. Like the `s_bktr` instruction, lines 1 and 2 fetch the top backtrack stack entry, transfer the character position therein to the character position register, and increment the backtrack stack pointer. However, the return stack pointer and program counter are not restored to the backtrack stack values. Instead, the program counter is set to an arbitrary absolute instruction address pc_j.

The use of the `s_res` instruction is in the execution of the and-predicate PEG expression, whose microcode is presented in Listing 3.7. Here, the `s_res` microcode can be found in lines 4 to 7, which are grouped according to principles 4 and 5. Recall from Section 3.2.2.8 that after successful evaluation of expression e in and-predicate expression $e?$, the character position needs to be restored to its value before evaluation of e. The restore operation is thus achieved by the `s_res` instruction.

```
1 t0 = bs[$bsp]
2 $cpos = t0[0]
3 $bsp = $bsp + 1
4 $pc = pc_j
```

Listing 3.25: Microcode for the `s_res` instruction.

The microcode behavior of `s_res` directly translates to the digital logic shown in Figure 3.14. The main observation is that the top entry is read from the backtrack stack, after which only the character position field is written to its respective register `$cpos`.

The equivalent inductive definition is presented in Equation (s_res.1), where it can be observed that only the character position and program counter state variables are changed to the value in the top backtrack stack entry and the absolute address operand respectively.

The unconditional jump to instruction address pc_j is facilitated by means of storing the address in the 24-bit operand field in the J-type instruction encoding format (see Section 3.4.2.3). The opcode of the `s_res` instruction consists of the following binary format and function values: `fmt = 10`, `func = 110`.

$$\frac{\begin{array}{l}\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{s\_res \$cpos,}\ pc_j\end{array}}{\langle pc_j\,,\ cp_b\,,\ rs\,,\ bs\,,\ t\rangle} \tag{s\_res.1}$$

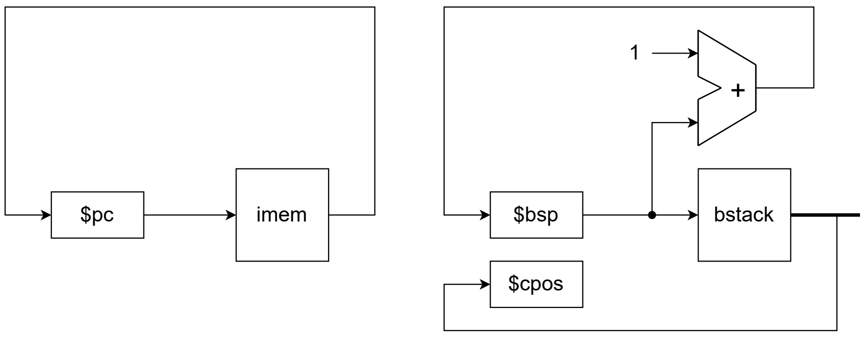


Figure 3.14: Logic diagram for the `s_res` instruction.

3.4.3.13 `s_rmv`

The `s_rmv` instruction is used to remove the top backtrack stack entry without restoring the parsing machine with the values within, which is achieved with the microcode shown in Listing 3.26. Line 1 increments the backtrack stack pointer, thereby discarding the top backtrack stack entry. Line 2 sets the program counter register to an absolute instruction address pc_j. The origin of the two micro-operations is the result from combining behavior in line 4 of the optional expression microcode (see Listing 3.4) and lines 4 and 5 of the sequence expression microcode (see Listing 3.9). These micro-operations are independent from surrounding micro-operations, which, combined with principles 4 and 5 is the reason for grouping the `s_rmv` microcode.

```
1 $bsp = $bsp + 1
2 $pc = pc_j
```

Listing 3.26: Microcode for the `s_rmv` instruction.

Figure 3.15 shows the rather simple logic diagram in order to support execution of the `s_rmv` instruction and Equation (s_rmv.1) presents its equally simple inductive definition. Note that, though the top backtrack entry is removed in the conclusive parsing machine state, none of its values are used said conclusive state.

Because the `s_rmv` instruction does not modify the character position register `$cpos`, its only operand is the absolute instruction address pc_j that the parsing machine unconditionally jumps to on execution. As before, this operand is stored based on the J-type instruction encoding format (see Section 3.4.2.3) with the following binary format and function values: `fmt = 10`, `func = 111`.

$$\frac{\begin{array}{l}\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \\ \texttt{imem}[pc] = \texttt{s\_rmv}\ \ pc_j\end{array}}{\langle pc_j\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle} \tag{s\_rmv.1}$$

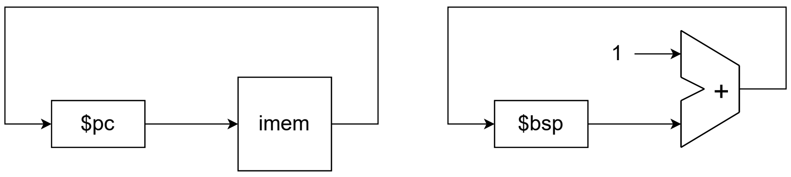


Figure 3.15: Logic diagram for the `s_rmv` instruction.

3.4.4 PPEG Architecture

The presented logic diagrams for the PPEG instructions can be combined to form the complete PPEG architecture logic diagram. This can be observed in Appendix A. Note the addition of the control unit, which contains all digital logic required to control the multiplexers, registers, and memory components. It does this based on the instruction opcode and three comparator outputs (2 character comparators + 1 string length comparator). For readability, the control signals for the multiplexers (select signals), registers (reset and write enable signals), and memory components (read enable and write enable) are not drawn in this schematic.

3.5 Translating PEG to PPEG Code

In the previous section the PPEG instruction set architecture was defined based on the microcode implementations of PEG expression as discussed in Section 3.3. The next step is to formalize the translation from PEG expression to PPEG code, which is the aim of this section. Finally, a proof is given that shows that the translation is coherent with the PEG recognizer formalization by means of the `match` function introduced in Section 3.2.

The section is divided into two parts. First an introduction and declaration of the PEG-PPEG translation formalism is provided in Section 3.5.1. Then in Section 3.5.2 the translation function is defined and proven for each PEG expression.

3.5.1 PEG-PPEG Translation Declaration

The translation of PEG to PPEG assembly code is defined as a function Π, whose declaration is provided in Equation (3.14). In order, the domain of Π consists of the set of PEG grammars $\mathcal{G}$; the set of natural numbers $\mathbb{N}$, which represents the initial program counter or instruction memory address at which to store the resulting PPEG code; and the set of PEG expressions $\mathcal{E}$. The latter has been previously defined in Section 3.2. The set $\mathcal{C}^*$ after the arrow is the codomain of Π, where $\mathcal{C}$ represents the set of valid PPEG instructions.

$$\Pi : \ \mathcal{G} \times \mathbb{N} \times \mathcal{E} \to \mathcal{C}^* \tag{3.14}$$

Invocation of the PEG-PPEG translation function is of the form $\Pi(G, pc, e)$. The resulting PPEG assembly code consists of zero or more instructions which are delimited by line feeds. Moreover, each instruction corresponds to a single program counter value pc. Assuming a function call $\Pi(G, pc, e)$, the first instruction is corresponds to address pc, the second to $pc + 1$, the third to $pc + 2$, etc.

3.5.2 PEG-PPEG Translation Definition and Proof

Where the previous section formalized a declaration of the PEG-PPEG translation function Π, this section aims to formalize its definition. Similar to the `match` function defined in Section 3.2.2, Π is inductively defined for each fundamental PEG expression. Therefore, the same notation is used with premises above a horizontal line and conclusion below. Note that the resulting PPEG code translations closely follow the microcode listings for each PEG expression as presented in Section 3.3.

After the definition of Π for a fundamental PEG expression, a proof is provided that the parsing machine behavior is in accordance with the PEG `match` recognizer behavior for that expression. Specifically, assuming the `match` definition of a PEG expression with an expected increase in character position, it is proven that the parsing machine state (see Section 3.4.3) is transformed to attain the same character position increase. These transformations are dictated by the PPEG instructions, as defined in the PPEG ISA, that are associated with the PEG expression under analysis or, in other words, by the PPEG instructions resulting from application of Π for the given PEG expression.

The following subsections define and explain the PEG-PPEG translation and proofs for all fundamental PEG expressions.

3.5.2.1 Literal String

The translation of the literal string PEG expression to a PPEG program is defined in Equation (3.15). As before, here a single-character literal string 'c' is assumed. A multi-character literal string is then a sequence PEG expression consisting of one or more of these single-character literal strings. Moreover, as discussed in Section 3.4.3.7, only a single `char_set` instruction is needed to evaluate a PEG expression 'c'. The handling of literal strings by the PPEG architecture

$$\frac{e = \text{‘}c\text{’}}{\Pi(G, pc, e) = \texttt{char\_set (\$cpos)+, } c\texttt{, } c} \tag{3.15}$$

Equation (3.16) shows a proof for correct parsing machine behavior in the case of a successful literal string expression evaluation. According to the operational semantics in Equation (lit.1), the expected final parsing machine state shows an increase in character position by 1 character, which is the same condition presented in the first line of Equation (3.16). Indeed, the second line shows that increase as $cp + 1$ in the parsing machine state transformation when executing the `char_set` instruction. Note that the notation $\xrightarrow{\texttt{char\_set}}$ represents a shorthand notation for the use of the inductive definition in Equation (char_set_beq.1), which assumes the associated conditions.

Equation (3.17) shows the proof for parsing behavior in the case of unsuccessful evaluation of a literal string. Recall that the null symbol $\varnothing$ represents an unsuccessful expression evaluation in terms of operational semantics (see Section 3.2). The equivalent parsing machine behavior is the initiation of a backtrack operation. Indeed, a backtrack operation is initiated by executing a `char_set` instruction when $s[i] \neq$ 'c' or the character position exceeds the string length. The result in state behavior is that the top backtrack stack entry (cp_b, pc_b, rs_b) is popped and its fields are used to restore the parsing state.

$$\begin{gathered}\texttt{if match(}G, \text{'}c\text{'}, s, i\texttt{)} = i+1 \texttt{ then}\\ \langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\texttt{char\_set}} \langle pc+1\,,\ cp+1\,,\ rs\,,\ bs\,,\ t\rangle\end{gathered} \tag{3.16}$$

$$\begin{gathered}\texttt{if match(}G, \text{'}c\text{'}, s, i\texttt{)} = \varnothing \texttt{ then}\\ \langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \xrightarrow{\texttt{char\_set}} \langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle\end{gathered} \tag{3.17}$$

3.5.2.2 Character Class

The translation of character class PEG expressions have already been discussed in some detail in Section 3.4.3.8. This explained how a combination of `char_set_beq` and `s_bktr` instructions can be used to evaluate this type of expression. This translation is formalized in Equation (3.18). Recall that the notation for the character class expression $[m]$ and its components C_x were introduced in Equation (3.11).

The PPEG code in the conclusion consists of n `char_set_beq` instructions, which correspond to individual character ranges (i.e., $c_{x,1}$-$c_{x,2}$) and elements (i.e., c_x). If the active character is within one of these ranges, the offset $n-(k-1)+1$ is used to jump past the `s_bktr` operation at the end, where k is the position of the `char_set_beq` instruction in the list of n other such instructions.

$$\frac{\begin{array}{l} e = [m] \\ m = C_1 C_2 \cdots C_n \\ n > 1 \\ (C_x = c_x = c_{x,1} = c_{x,2}) \vee (C_x = c_{x,1}\text{-}c_{x,2}) \end{array}}{\begin{array}{rl}\Pi(G, pc, e) = & \texttt{char\_set\_beq (\$cpos)+,}\ c_{1,1}\texttt{,}\ c_{1,2}\texttt{,}\ n+1 \\ & \vdots \\ & \texttt{char\_set\_beq (\$cpos)+,}\ c_{k,1}\texttt{,}\ c_{k,2}\texttt{,}\ n-(k-1)+1 \\ & \vdots \\ & \texttt{char\_set\_beq (\$cpos)+,}\ c_{n,1}\texttt{,}\ c_{n,2}\texttt{,}\ 2 \\ & \texttt{s\_bktr \$cpos}\end{array}} \tag{3.18}$$

Equation (3.19) proves that the parsing machine state behavior for a successful character class expression evaluation matches that of the expression's operational semantics defined in Equation (cls.1). The notation $\xrightarrow{\texttt{char\_set\_beq+}}$ represents the execution of one or more `char_set_beq` instructions, until one initiates the jump past `s_bktr`. Note in the final parsing machine state the increase in program counter value to $pc{+}n{+}1$, which rightfully addresses the instruction after `s_bktr`.

Equation (3.20) proves the same match in parsing machine state behavior and operational semantics for an unsuccessful evaluation of character class expressions. The parsing machine ends up executing the `s_bktr` instruction either after executing all n `char_set_beq` instructions or after executing only the first one, and finding the character position value to be outside the allowable range dictated by `$fs` (see Section 3.4.3.8).

$$\begin{aligned}&\texttt{if match}(G,[m],s,i) = i+1 \texttt{ then}\\ &\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\texttt{char\_set\_beq+}} \langle pc+n+1\,,\ cp+1\,,\ rs\,,\ bs\,,\ t\rangle\end{aligned} \tag{3.19}$$

$$\begin{aligned}&\texttt{if match}(G,[m],s,i) = \varnothing \texttt{ then}\\ &\langle pc\,,\ cp\,,\ rs\,,\ (cp_b,pc_b,rs_b):bs\,,\ t\rangle \xrightarrow{\texttt{char\_set\_beq+}} \langle pc+n\,,\ cp\,,\ rs\,,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (cp_b,pc_b,rs_b):bs\,,\ t\rangle\\ &\qquad\qquad\qquad\qquad\qquad\qquad \xrightarrow{\texttt{s\_bktr}} \langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle\end{aligned} \tag{3.20}$$

It must be noted that the formal translation definition in Equation (3.18) intentionally contains the premise $n > 1$, thereby indicating that this translation is only valid for character classes with more than one element. The reason for this is that, in the case of a character class consisting only of a single element (i.e., either $[c]$ or $[c_1\text{-}c_2]$), the PPEG code can be simplified to a single `char_set` instruction. This is formalized in Equation (3.21).

$$\frac{\begin{array}{c}e = [C_1]\\ (C_1 = c_1 = c_{1,1} = c_{1,2}) \vee (C_1 = c_{1,1}\text{-}c_{1,2})\end{array}}{\Pi(G,pc,e) = \texttt{char\_set (\$cpos)+, } c_{1,1}\texttt{, } c_{1,2}} \tag{3.21}$$

The use of the `char_set` instruction for a single range character class PEG expression is proven to be correct for successful evaluation in Equation (3.22) and for unsuccessful evaluation in Equation (3.23). Note that, in comparison with the translation in Equation (3.18), using this implementation only saves a single clock cycle in the case of an unsuccessful evaluation of the single-element character class expression.

$$\begin{aligned}&\texttt{if match}(G,[C_1],s,i) = i+1 \texttt{ then}\\ &\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\texttt{char\_set}} \langle pc+1\,,\ cp+1\,,\ rs\,,\ bs\,,\ t\rangle\end{aligned} \tag{3.22}$$

$$\begin{array}{l}\texttt{if match(}G, [C_1], s, i\texttt{)} = \varnothing \texttt{ then} \\ \langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \xrightarrow{\texttt{char\_set}} \langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle\end{array} \tag{3.23}$$

3.5.2.3 Any Character

As previously discussed in the any instruction specification (see Section 3.4.3.4), only the this instruction is used to evaluate an any PEG expression, which is indeed what the translation definition in Equation (3.24) shows.

$$\frac{e = \text{`.'}}{\Pi(G, pc, e) = \texttt{any \$cpos}} \tag{3.24}$$

Equation (3.25) and Equation (3.26) prove that the evaluation behavior of an any PEG expression is matched by execution of the any instruction. That is, on successful evaluation, Equation (3.25) shows that the character position in the parsing machine state is incremented by one character, following the any inductive definition in Equation (any.1). In contrast, if evaluation fails, a backtrack operation is performed according to the state transformation in Equation (any.2), which matches with the expected PEG behavior as observed in Equation (3.26).

$$\begin{array}{l}\texttt{if match(}G, \text{`.'}, s, i\texttt{)} = i + 1 \texttt{ then} \\ \langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\texttt{any}} \langle pc + 1\,,\ cp + 1\,,\ rs\,,\ bs\,,\ t\rangle\end{array} \tag{3.25}$$

$$\begin{array}{l}\texttt{if match(}G, \text{`.'}, s, i\texttt{)} = \varnothing \texttt{ then} \\ \langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \xrightarrow{\texttt{any}} \langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle\end{array} \tag{3.26}$$

3.5.2.4 Optional

Equation (3.27) presents the translation of an optional PEG expression, which follows the microcode behavior seen in Listing 3.4. First an s_push instruction is issued to store the current parsing machine state if evaluation of expression e' fails. Note that the absolute backtrack instruction address is $pc + |\Pi(G, x, e')| + 2$. The notation $|\Pi(G, x, e)|$ represents the number of instructions resulting from the translation of expression e, regardless of initial program counter value as indicated by x. This particular backtrack address therefore points to the instruction after s_rmv. Next, the translation $\Pi(G, pc, e?)$ evidently involves the translation of expression e itself, which is represented by the second line. Finally, only if e' is successfully evaluated, s_rmv is needed to remove the previously pushed backtrack stack entry. Note that its jump operand is the same absolute instruction address: simply the address of the next instruction.

$$\frac{e = e'?}{\begin{aligned}\Pi(G, pc, e) = {} & \texttt{s\_push \$cpos,}\ pc + \left|\Pi(G, x, e')\right| + 2 \\ & \Pi(G, pc+1, e') \\ & \texttt{s\_rmv}\ \ pc + \left|\Pi(G, x, e')\right| + 2\end{aligned}} \tag{3.27}$$

As stated in the previous paragraph, the PEG `match` behavior of an optional expression has two possible outcomes depending on evaluation of (see also Section 3.2.2.5). Using the PPEG code translation defined in Equation (3.27), proof of correct parsing machine behavior for a successful evaluation is presented in Equation (3.28). Similar to the operational semantics in Equation (opt.1), it assumes execution of code generated by $\Pi(G, pc + 1, e)$ results in a character position increase of j.

Unsuccessful evaluation of $e?$ defined in Equation (opt.2) is also correctly mirrored by the parsing machine model as made evident by Equation (3.29). Here it is assumed that a backtrack operation is initiated somewhere during execution of PPEG code generated by $\Pi(G, pc + 1, e)$, which indeed happens when evaluation of e fails.

$$\begin{aligned}
&\texttt{if match(}G, e?, s, i\texttt{)} = i + j\ \texttt{then} \\
&\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\texttt{s\_push}} \langle pc + 1\,,\ cp\,,\ rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G, x, e)| + 2, rs) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\Pi(G,pc+1,e)} \langle pc + |\Pi(G, x, e)| + 1\,,\ cp + j\,,\ rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G, x, e)| + 2, rs) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\texttt{s\_rmv}} \langle pc + |\Pi(G, x, e)| + 2\,,\ cp + j\,,\ rs\,,\ bs\,,\ t\rangle
\end{aligned} \tag{3.28}$$

$$\begin{aligned}
&\texttt{if match(}G, e?, s, i\texttt{)} = i\ \texttt{then} \\
&\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\texttt{s\_push}} \langle pc + 1\,,\ cp\,,\ rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G, x, e)| + 2, rs) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\Pi(G,pc+1,e)} \langle pc + |\Pi(G, x, e)| + 2\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle
\end{aligned} \tag{3.29}$$

3.5.2.5 Zero-Or-More

Equation (3.30) defines the translation from zero-or-more PEG expression to PPEG code according to the microcode behavior seen in Listing 3.5. Much the same way as the optional PEG expression translation, the PPEG code starts out pushing the current parsing machine state, with a backtrack address pointing to the instruction after `s_upd`. Next, given expression $e'*$, the PPEG code of expression e' represented as $\Pi(G, pc + 1, e')$ is executed in order to evaluate said expression. Only if it succeeds, will the final instruction `s_upd` be executed, which updates the character position in the previously pushed backtrack stack entry and jumps back to the first PPEG instruction resulting from $\Pi(G, pc + 1, e')$, as indicated by absolute address $pc + 1$.

$$\frac{e = e'*}{\begin{aligned}\Pi(G, pc, e) = {} & \texttt{s\_push \$cpos, } pc + \left|\Pi(G, x, e')\right| + 2 \\ & \Pi(G, pc + 1, e') \\ & \texttt{s\_upd \$cpos, } pc + 1\end{aligned}} \tag{3.30}$$

A zero-or-more expression $e*$ always evaluates successfully, but has two distinct evaluation behaviors depending on the evaluation outcome of expression e, which can clearly be observed in Section 3.2.2.6. On successful evaluation of e, the PEG operational semantics in Equation (zom.1) states that $e*$ must then be evaluated at the new increased character position. Equation (3.31) proves that the parsing machine adheres to this behavior. After the initial successful evaluation of e by executing $\Pi(G, pc + 1, e)$, which increased the character position by j characters, execution of `s_upd` causes the backtrack stack entry to be updated with the new character position value and a jump is initiated back to $pc + 1$. Next, the notation $\overset{*}{\rightarrow}$ represents the indefinite execution of the former two lines, until at last execution terminates due to a backtrack operation, which lets the machine jump to $pc + |\Pi(G, x, e)| + 2$ (instruction after `s_upd`). The result in a final character position increment of $j + k$, matching the expected behavior by $\texttt{match(}G, e*, s, i\texttt{)}$.

The other $e*$ expression evaluation behavior is proven to be correct in Equation (3.32) and matches the expected operational semantics defined in Equation (zom.2). That is, due to an unsuccessful evaluation of e by executing $\Pi(G, pc+1, e)$, the previously pushed backtrack stack entry is restored, which lets the parsing machine jump to the instruction after `s_upd`.

$$\begin{aligned}
&\texttt{if match(}G, e*, s, i\texttt{)} = i + j + k \texttt{ then} \\
&\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\texttt{s\_push}} \langle pc + 1\,,\ cp\,,\ rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G, x, e)| + 2, rs) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\Pi(G, pc+1, e)} \langle pc + |\Pi(G, x, e)| + 1\,,\ cp + j\,,\ rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G, x, e)| + 2, rs) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\texttt{s\_upd}} \langle pc + 1\,,\ cp + j\,,\ rs\,, \\
&\qquad\qquad (cp + j, pc + |\Pi(G, x, e)| + 2, rs) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{*} \langle pc + |\Pi(G, x, e)| + 2\,,\ cp + j + k\,,\ rs\,,\ bs\,,\ t\rangle
\end{aligned} \tag{3.31}$$

$$\begin{aligned}
&\texttt{if match(}G, e*, s, i\texttt{)} = i \texttt{ then} \\
&\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\texttt{s\_push}} \langle pc + 1\,,\ cp\,,\ rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G, x, e)| + 2, rs) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\Pi(G, pc+1, e)} \langle pc + |\Pi(G, x, e)| + 2\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle
\end{aligned} \tag{3.32}$$

3.5.2.6 One-Or-More

Translation of the one-or-more PEG expression $e'+$ shown in Equation (3.33) is straightforward, as this involves the concatenation of a translation of e' and of $e'*$, the latter of which has been handled in the previous section. The same substitution can also be observed in the microcode for the one-or-more expression in Listing 3.6.

$$\frac{e = e'+}{\begin{aligned}\Pi(G, pc, e) = {}& \Pi(G, pc, e') \\ & \Pi(G, pc + \left|\Pi(G, x, e')\right|, e'*)\end{aligned}} \tag{3.33}$$

Similar to the zero-or-more PEG expression, there are two distinct evaluation behaviors for a given one-or-more expression $e+$, namely one successful and the other unsuccessful. Equation (3.34) presents the case for successful evaluation and shows that the resulting increase in character position is exactly as expected by the operational semantics defined in Equation (oom.1). It can be seen that the proof uses the execution behavior of the zero-or-more expression as observed in Equation (3.31).

The proof for correct behavior when expression $e+$ is unsuccessfully evaluated is presented in Equation (3.35). Note that the only process by which a one-or-more PEG expression may fail is when the initial evaluation of expression e fails, as defined in Equation (oom.2). This too can be seen in the proof in the form of a backtrack operation during execution of code produced by $\Pi(G, pc, e)$.

$$\begin{aligned}
&\texttt{if match}(G, e+, s, i) = i + j + k \texttt{ then} \\
&\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\Pi(G,pc,e)} \langle pc + |\Pi(G, pc, e)|\ ,\ cp + j\ ,\ rs\ ,\ bs\ ,\ t\rangle \\
&\qquad\qquad\qquad\qquad \xrightarrow{\Pi(G,pc,e*)} \langle pc + |\Pi(G, pc, e)| + |\Pi(G, pc, e*)|\ , \\
&\qquad\qquad\qquad\qquad\qquad\qquad cp + j + k\ ,\ rs\ ,\ bs\ ,\ t\rangle
\end{aligned} \tag{3.34}$$

$$\begin{aligned}
&\texttt{if match}(G, e+, s, i) = \varnothing \texttt{ then} \\
&\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \xrightarrow{\Pi(G,pc,e)} \langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle
\end{aligned} \tag{3.35}$$

3.5.2.7 And-Predicate

The translation of the and-predicate PEG expression presented in Equation (3.36) follows directly from its microcode behavior shown in Listing 3.7 and the PPEG ISA specification discussed in Section 3.4.3. Evaluation of and-predicate expression $\&e'$ is not-unlike evaluating e' itself, except for the fact that it may not increase the character position. To this end, an `s_push` instruction is issued to store the initial character position value. Thereafter, e' itself is evaluated by executing the PPEG code $\Pi(G, pc + 1, e')$. On successful evaluation, the `s_res` instruction restores the character position as stored by the `s_push` instruction and jumps past `s_bktr`. On failed evaluation of e', the machine state pushed by the initial `s_push` instruction is used to jump to `s_bktr`, which initiates a backtrack operation, thereby failing the entire evaluation of $\&e'$

$$\frac{e = \&e'}{\begin{aligned}\Pi(G, pc, e) = {} & \texttt{s\_push \$cpos,}\ pc + \left|\Pi(G, x, e')\right| + 2 \\ & \Pi(G, pc+1, e') \\ & \texttt{s\_res \$cpos,}\ pc + \left|\Pi(G, x, e')\right| + 3 \\ & \texttt{s\_bktr \$cpos}\end{aligned}} \tag{3.36}$$

The proof for correctness of the described parsing machine behavior can be observed for both successful evaluation and unsuccessful evaluation in Equation (3.37) and Equation (3.38) respectively. Indeed, Equation (3.37) shows that on successful evaluation of expression e and thus by extension $\&e$, the character position state variable of the parsing machine is restored to its original value cp, matching the operational semantics defined in Equation (and.1). On the other hand, Equation (3.38) shows that a backtrack operation is initiated on failed evaluation of e when executing $\Pi(G, pc + 1, e)$. This in turn results in the occurrence of a backtrack operation as initiated by `s_bktr`, thereby failing the evaluation of $\&e$ as a whole, which is exactly as indicated by the operational semantics defined in Equation (and.2).

$$\begin{aligned}
&\texttt{if match(}G, \&e, s, i\texttt{)} = i \texttt{ then} \\
&\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\texttt{s\_push}} \langle pc+1\,,\ cp\,,\ rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G,x,e)| + 2, rs) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\Pi(G,pc+1,e)} \langle pc + |\Pi(G,x,e)| + 1\,,\ cp + j\,,\ rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G,x,e)| + 2, rs) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\texttt{s\_res}} \langle pc + |\Pi(G,x,e)| + 3\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle
\end{aligned} \tag{3.37}$$

$$\begin{aligned}
&\texttt{if match(}G, \&e, s, i\texttt{)} = \varnothing \texttt{ then} \\
&\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \xrightarrow{\texttt{s\_push}} \langle pc+1\,,\ cp\,,\ rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G,x,e)| + 2, rs) : \\
&\qquad\qquad (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\Pi(G,pc+1,e)} \langle pc + |\Pi(G,x,e)| + 2\,,\ cp\,,\ rs\,, \\
&\qquad\qquad (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\texttt{s\_bktr}} \langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle
\end{aligned} \tag{3.38}$$

3.5.2.8 Not-Predicate

The not-predicate PEG expression $!e$ is opposite to the and-predicate expression &! in that it fails on successful evaluation of e and succeeds otherwise, but similar in that the character position may not be increased on successful evaluation of $!e$. This has previously been described in terms of microcode in Listing 3.8 and, together with the PPEG ISA specification in Section 3.4.3, forms the basis of the translation presented in Equation (3.39).

The not-predicate expression $!e'$ translation starts with storing the current parsing machine state by issuing a `s_push` instruction. Thereafter, the PPEG code for expression e' is executed. If evaluation fails, the stored state is used to recover the character position and to jump past `s_rmv_bktr`. Otherwise, the second to top backtrack stack entry is used by `s_rmv_bktr` to initiate a backtrack operation, indicating an unsuccessful evaluation of $!e'$.

$$\frac{e = !e'}{\begin{aligned}\Pi(G, pc, e) = {} & \texttt{s\_push \$cpos,}\ pc + \left|\Pi(G, x, e')\right| + 2 \\ & \Pi(G, pc+1, e') \\ & \texttt{s\_rmv\_bktr \$cpos}\end{aligned}} \tag{3.39}$$

The correctness of the description in the previous paragraph is proven for both successful evaluation and unsuccessful evaluation of the not-predicate expression in Equation (3.40) and Equation (3.41) respectively. Similar to the parsing machine behavior of and-predicate evaluation, the proof for successful evaluation of $!e$ shows that the initial character position is retained by the parsing machine, which matches the operational semantics in Equation (not.1). The operational semantics for unsuccessful evaluation of $!e$ in Equation (not.2) is also matched by the parsing machine behavior, as seen by the backtrack operation at the end.

$$\begin{aligned}
&\texttt{if match}(G, !e, s, i) = i\ \texttt{then} \\
&\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\texttt{s\_push}} \langle pc+1\,,\ cp\,,\ rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G,x,e)| + 2, rs) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\Pi(G,pc+1,e)} \langle pc + |\Pi(G,x,e)| + 2\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle
\end{aligned} \tag{3.40}$$

$$\begin{aligned}
&\texttt{if match}(G, !e, s, i) = \varnothing\ \texttt{then} \\
&\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \xrightarrow{\texttt{s\_push}} \langle pc+1\,,\ cp\,,\ rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G,x,e)| + 2, rs) : \\
&\qquad\qquad (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\Pi(G,pc+1,e)} \langle pc + |\Pi(G,x,e)| + 1\,,\ cp\,,\ rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G,x,e)| + 2, rs) : \\
&\qquad\qquad (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \\
&\qquad \xrightarrow{\texttt{s\_rmv\_bktr}} \langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle
\end{aligned} \tag{3.41}$$

3.5.2.9 Sequence

The translation of a sequence PEG expression e_1e_2 to PPEG code is presented in Equation (3.42) and is simply the concatenation of code from translating e_1 and e_2.

$$\frac{e = e_1e_2}{\begin{aligned}\Pi(G, pc, e) = \Pi(G, pc, e_1)& \\ \Pi(G, pc + \lvert\Pi(G, x, e_1)\rvert\,, e_2)&\end{aligned}} \tag{3.42}$$

The behavior of the parsing machine when executing the code in the conclusion of Equation (3.42) depends on the evaluation results of both e_1 and e_2. For this reason there are three distinct execution paths, which mirror the three results discussed in the operational semantics of the sequence expression (see Section 3.2.2.10). First, Equation (3.43) proves that if both expressions e_1 and e_2 evaluate successfully, the parsing machine ends up in the same state as by the `match` function, namely an increase of $j + k$ in character position. Similarly, if either e_1 succeeds and e_2 fails or e_1 fails evaluation directly, then Equation (3.44) and Equation (3.45 respectively prove that the parsing machine behavior is in accordance with the operational semantics with the same conditions, i.e., Equation (seq.2) and Equation (seq.3).

$$\begin{aligned}&\texttt{if match(}G, e_1e_2, s, i\texttt{)} = i + j + k \texttt{ then}\\ &\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\Pi(G,pc,e_1)} \langle pc + \lvert\Pi(G, x, e_1)\rvert\ ,\ cp + j\ ,\ rs\,,\ bs\,,\ t\rangle\\ &\qquad\xrightarrow{\Pi(G,pc+\lvert\Pi(G,x,e_1)\rvert,e_2)} \langle pc + \lvert\Pi(G, x, e_1)\rvert + \lvert\Pi(G, x, e_2)\rvert\ ,\\ &\qquad\qquad\qquad\qquad cp + j + k\,,\ rs\,,\ bs\,,\ t\rangle\end{aligned} \tag{3.43}$$

$$\begin{aligned}&\texttt{if match(}G, e_1e_2, s, i\texttt{)} = \varnothing \texttt{ then}\\ &\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \xrightarrow{\Pi(G,pc,e_1)} \langle pc + \lvert\Pi(G, x, e_1)\rvert\ ,\ cp\,,\ rs\,,\\ &\qquad\qquad\qquad\qquad\qquad\qquad (cp_b, pc_b, rs_b) : bs\,,\ t\rangle\\ &\qquad\qquad\qquad\qquad\xrightarrow{\Pi(G,pc+\lvert\Pi(G,x,e_1)\rvert,e_2)} \langle pc_b\,,\ cp_b\,,\ rs\,,\ bs\,,\ t\rangle\end{aligned} \tag{3.44}$$

$$\begin{aligned}&\texttt{if match(}G, e_1e_2, s, i\texttt{)} = \varnothing \texttt{ then}\\ &\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \xrightarrow{\Pi(G,pc,e_1)} \langle pc_b\,,\ cp_b\,,\ rs\,,\ bs\,,\ t\rangle\end{aligned} \tag{3.45}$$

3.5.2.10 Prioritized Choice

The prioritized choice PEG expression e_1 / e_2 also involves the possible evaluation of two sub-expressions e_1 and e_2. However, it differs from the previously discussed sequence expression in that only one of these expressions has to be successfully evaluated for the entire expression to be successfully evaluated. This behavior is translated to PPEG code as defined in Equation (3.46).

At the start, a `s_push` instruction is issued to be able to possibly reset the parsing machine state to the current one before trying the evaluate e_2 if e_1 failed evaluation, which is why the backtrack address corresponds to the address of the first instruction of $\Pi(G, x, e_2)$. Second, the code corresponding to expression e_1 is executed. Here, the execution path may split. If e_1 is successfully evaluated, `s_rmv` is executed, which removes the initially pushed backtrack entry and jumps past the code for expression e_2. Otherwise, the same backtrack entry allows the parsing machine to jump to the code for expression e_2.

$$\frac{e = e_1 \;/\; e_2}{\begin{aligned}\Pi(G, pc, e) = {} & \texttt{s\_push \$cpos,}\;\; pc + |\Pi(G, x, e_1)| + 2 \\ & \Pi(G, pc+1, e_1) \\ & \texttt{s\_rmv}\;\; pc + |\Pi(G, x, e_1)| + |\Pi(G, x, e_2)| + 2 \\ & \Pi(G, pc + |\Pi(G, x, e_1)| + 2, e_2)\end{aligned}} \tag{3.46}$$

As with the sequence PEG expression, three execution paths can followed corresponding to the three operational semantics definitions defined in Section 3.2.2.11. The proof in Equation (3.47) shows the parsing machine behavior when expression e_1 is successfully evaluated, which matches the expected result defined in Equation (prc.1). The proof for correct behavior when evaluation of e_1 fails but evaluation of e_2 succeeds is presented in Equation (3.48) and also matches the expected result from the operational semantics defined in Equation (prc.2). Finally, if both expressions are unsuccessfully evaluated, the parsing machine behavior still correctly follows the expected behavior presented in Equation (prc.3), as is proven in Equation (3.49). That is, the parsing machine is restored to some previous stored state by means of a backtrack operation.

$$\begin{aligned}
&\texttt{if match(}G, e_1 \;/\; e_2, s, i\texttt{)} = i + j \texttt{ then} \\
&\langle pc\,,\; cp\,,\; rs\,,\; bs\,,\; t\rangle \xrightarrow{\texttt{s\_push}} \langle pc+1\,,\; cp\,,\; rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G, x, e_1)| + 2, rs) : bs\,,\; t\rangle \\
&\qquad \xrightarrow{\Pi(G,pc+1,e_1)} \langle pc + |\Pi(G, x, e_1)| + 1\,,\; cp + j\,,\; rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G, x, e_1)| + 2, rs) : bs\,,\; t\rangle \\
&\qquad \xrightarrow{\texttt{s\_rmv}} \langle pc + |\Pi(G, x, e_1)| + |\Pi(G, x, e_2)| + 2\,, \\
&\qquad\qquad cp + j\,,\; rs\,,\; bs\,,\; t\rangle
\end{aligned} \tag{3.47}$$

$$\begin{aligned}
&\texttt{if match(}G, e_1 \;/\; e_2, s, i\texttt{)} = i + k \texttt{ then} \\
&\langle pc\,,\; cp\,,\; rs\,,\; bs\,,\; t\rangle \xrightarrow{\texttt{s\_push}} \langle pc+1\,,\; cp\,,\; rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G, x, e_1)| + 2, rs) : bs\,,\; t\rangle \\
&\qquad \xrightarrow{\Pi(G,pc+1,e_1)} \langle pc + |\Pi(G, x, e_1)| + 2\,,\; cp\,,\; rs\,,\; bs\,,\; t\rangle \\
&\qquad \xrightarrow{\Pi(G,pc+|\Pi(G,x,e_1)|+2,e_2)} \langle pc + |\Pi(G, x, e_1)| + |\Pi(G, x, e_2)| + 2\,, \\
&\qquad\qquad cp + k\,,\; rs\,,\; bs\,,\; t\rangle
\end{aligned} \tag{3.48}$$

$$\begin{aligned}
&\texttt{if match}(G, e_1 \;/\; e_2, s, i) = \varnothing \;\texttt{then} \\
&\langle pc\,,\; cp\,,\; rs\,,\; (cp_b, pc_b, rs_b) : bs\,,\; t\rangle \xrightarrow{\texttt{s\_push}} \langle pc + 1\,,\; cp\,,\; rs\,, \\
&\qquad\qquad (cp, pc + |\Pi(G, x, e_1)| + 2, rs) : \\
&\qquad\qquad (cp_b, pc_b, rs_b) : bs\,,\; t\rangle \\
&\xrightarrow{\Pi(G, pc+1, e_1)} \langle pc + |\Pi(G, x, e_1)| + 2\,,\; cp\,,\; rs\,, \\
&\qquad\qquad (cp_b, pc_b, rs_b) : bs\,,\; t\rangle \\
&\xrightarrow{\Pi(G, pc+|\Pi(G,x,e_1)|+2, e_2)} \langle pc_b\,,\; cp_b\,,\; rs_b\,,\; bs\,,\; t\rangle
\end{aligned} \tag{3.49}$$

3.5.2.11 Non-Terminal

In Section 3.9 the operational semantics of the non-terminal PEG expression A_k was discussed, from which it followed that its evaluation is directly related to the evaluation of the expression e' associated with the production rule $A_k \leftarrow e'$. Following the microcode behavior presented Listing 3.11, the non-terminal expression translates to a single call to the PPEG subroutine associated with non-terminal as shown in Equation (3.50). Note that the absolute address of this subroutine is represented by evaluation of a function $o(G, A_k)$. The exact meaning of this function is discussed later in Section 3.5.3.

$$\frac{\begin{array}{l} e = A_k \\ G = (\Sigma, N, S, P) \\ A_k \in N \end{array}}{\Pi(G, pc, e) = \texttt{call}\;\; o(G, A_k)} \tag{3.50}$$

Based on the formalized translation of the non-terminal expression, Equation (3.51) and Equation (3.52) provide proofs for correct parsing machine behavior when evaluating such expressions. Both proofs shows that after the `call` instruction, the parsing machine jumps to absolute address $o(G, A_k)$. Here, the parsing machine starts execution of PPEG code associated with e, where $e = \texttt{ntmap}(G, A_k)$ (see Equation (3.13)). In Equation (3.51) it can be seen that, assuming successful evaluation of e, execution ends up at the instruction after the last instruction belonging to e. That is, execution resumes at instruction address $o(G, A_k) + |\Pi(G, x, \texttt{ntmap}(G, A_k))|$, where the latter term represents the number of PPEG instructions with the translation of expression e. There, a `ret` instruction is executed (see Section 3.5.3), which takes the machine back to the instruction after the initial `call` instruction by means of the top return stack entry.

In contrast, if evaluation of e fails, by extension evaluation of non-terminal expression A_k fails too. For this reason, a backtrack operation is initiated, which restores the state variables to values stored in the top backtrack stack entry as presented in Equation (3.52). The result of such an operation is that the return stack entry pushed by the initial `call` instruction is discarded. Moreover, the parsing machine program execution jumps out of the currently executing PPEG subroutine associated with non-terminal A_k. This is in accordance with the operational semantics of the non-terminal PEG expression presented in Equation (nte.2).

$$\begin{aligned}
&\texttt{if match(}G, A_k, s, i\texttt{)} = i + j \texttt{ then}\\
&\langle pc\,,\ cp\,,\ rs\,,\ bs\,,\ t\rangle \xrightarrow{\texttt{call}} \langle o(G, A_k)\,,\ cp\,,\ (pc+1) : rs\,,\ bs\,,\ t\rangle\\
&\xrightarrow{\Pi(G, o(G,A_k), \texttt{ntmap(}G,A_k\texttt{)})} \langle o(G, A_k) + |\Pi(G, x, \texttt{ntmap(}G, A_k\texttt{)})|\,,\\
&\qquad cp + j\,,\ (pc+1) : rs\,,\ bs\,,\ t\rangle\\
&\xrightarrow{\texttt{ret}} \langle pc + 1\,,\ cp + j\,,\ rs\,,\ bs\,,\ t\rangle
\end{aligned} \tag{3.51}$$

$$\begin{aligned}
&\texttt{if match(}G, A_k, s, i\texttt{)} = \varnothing \texttt{ then}\\
&\langle pc\,,\ cp\,,\ rs\,,\ (cp_b, pc_b, rs_b) : bs\,,\ t\rangle \xrightarrow{\texttt{call}} \langle o(G, A_k)\,,\ cp\,,\ (pc+1) : rs\,,\\
&\qquad (cp_b, pc_b, rs_b) : bs\,,\ t\rangle\\
&\xrightarrow{\Pi(G, o(G,A_k), \texttt{ntmap(}G,A_k\texttt{)})} \langle pc_b\,,\ cp_b\,,\ rs_b\,,\ bs\,,\ t\rangle
\end{aligned} \tag{3.52}$$

3.5.3 PEG Grammar Translation

The translation of fundamental PEG expressions discussed in Section 3.5.2 pave the way for a formalization of the translation of entire PEG grammars. The parsing machine behavior for evaluation of a PEG grammar has already been discussed to some extent in Section 3.3.2.13, but was limited to the microcode level. Here, the parsing machine behavior is defined in terms of PPEG code by introducing a PEG grammar translation function $\Pi'(G)$.

$$\Pi' :\ \mathcal{G} \to \mathcal{C}^* \tag{3.53}$$

Equation (3.53) presents the declaration of the grammar translation function. The domain of Π', which is simply a subset of Π as declared in Equation (3.14), solely consists of the set of PEG grammars $\mathcal{G}$. Its codomain $\mathcal{C}^*$ represents PPEG code consisting of an arbitrary number of PPEG instructions.

Finally, the definition of the partial PEG grammar translation function Π' is presented in Equation (3.54). The premises of its inductive definition set the conditions for a valid translation, which is to say those PEG grammars with at least one non-terminal and its production rule. The grammars that adhere to these conditions are translated to the PPEG code seen in the conclusion.

At the start of any PEG grammar evaluation, a backtrack stack entry is pushed in order to have a defined execution path for when the evaluation of the grammar fails. Specifically, the `s_push` instruction stores the absolute address of the `set_fail` instruction, which sets a status flag indicating that grammar evaluation failed. Next, a call to the start non-terminal is invoked, whose code is always stored starting at address 4. Recall from Section 3.5.2.11 that if evaluation of a non-terminal succeeds, program execution returns to the instruction following its call. Therefore, if evaluation of the grammar succeeds, the `set_success` instruction is executed next, which sets a status flag indicating that grammar evaluation succeeded.

An important observation from the translated PPEG code is how the code for the grammar's non-terminals, i.e., the code for $e = \texttt{ntmap(}G, A_k\texttt{)}$ (see Section 3.2.2.12), is stored. First, it is expected that the code for non-terminals is stored in the order that they were defined, which ensures that the start non-terminal $S = A_1$ is always stored starting at address 4. Second, code for non-terminals are always followed by a `ret` instruction. As was explained in Section 3.5.2.11, the result is that program execution returns to just after a non-terminal was initially called, provided evaluation of said non-terminal succeeds.

$$\frac{\begin{array}{l} G = (\Sigma, N, S, P) \\ S = A_1 \\ N = \{A_1, A_2, \ldots, A_n\} \\ n > 0 \end{array}}{\begin{array}{ll} \Pi'(G) = & \texttt{s\_push \$cpos, 3} \\ & \texttt{call 4} \\ & \texttt{set\_success} \\ & \texttt{set\_fail} \\ & \Pi(G, 4, \texttt{ntmap(}G, A_1\texttt{)}) \\ & \texttt{ret} \\ & \vdots \\ & \Pi(G, 4 + \Sigma_{j=1}^{k-1}\{|\Pi(G, x, \texttt{ntmap(}G, A_j\texttt{)})| + 1\}, \texttt{ntmap(}G, A_k\texttt{)}) \\ & \texttt{ret} \\ & \vdots \\ & \Pi(G, 4 + \Sigma_{j=1}^{n-1}\{|\Pi(G, x, \texttt{ntmap(}G, A_j\texttt{)})| + 1\}, \texttt{ntmap(}G, A_n\texttt{)}) \\ & \texttt{ret} \end{array}} \quad (3.54)$$

Finally, observe the notation used to indicate the start address of each non-terminal translation. In general given the grammar G, the start address for translation of the expression associated with non-terminal A_k is computed with a function $o(G, A_k)$ defined in Equation (3.55). The first non-terminal is located at address 4, which is the reason for that constant in the definition of o. Thereafter, the location of an arbitrary non-terminal translation depends only on the sum of the number of instruction for each non-terminal translation before it plus one. The "plus one" originates from the fact that each non-terminal translation is extended with a single `ret` instruction.

$$\frac{\begin{array}{l} G = (\Sigma, N, S, P) \\ N = \{A_1, A_2, \ldots, A_n\} \\ 1 \leq k \leq n \end{array}}{o(G, A_k) = 4 + \Sigma_{j=1}^{k-1}\{|\Pi(G, x, \texttt{ntmap(}G, A_j\texttt{)})| + 1\}} \quad (3.55)$$

At this point it may be proven that any PEG grammar G that can be translated to PPEG code by $\Pi'(G)$ can be used to parse any input string as is expected by the PEG operational semantics defined in Section 3.2. Execution of a grammar's equivalent PPEG code must follow the operational semantics according to the result of $\texttt{match(}G, A_1, s, 0\texttt{)} = i$. Note that the initial parsing machine state assumes a total reset of the parsing machine before execution starts. to this end, the program counter points to address 0; the character position is set to 0; both the return stack and backtrack stack are empty, as indicated by the null-symbol ∅; and the parsing machine status is set to 0, indicating an "active parse" status.

The proofs for successful and unsuccessful evaluation of a grammar G and input string s are provided in Equation (3.56) and Equation (3.57) respectively. Both start with the initial backtrack entry that is stored and a call to the start non-terminal. Next, Equation (3.56) assumes that evaluation of this non-terminal, and thus the grammar as a whole, evaluates to success, which in has increased the character position by i characters. By executing the `ret` at the end of the non-terminal code it gracefully returns to execute the `set_success` instruction. Note the final parsing machine state, which consists of the character position set to i as expected by the operational semantics. Moreover, in contrast to the return stack, the backtrack stack still contains a single entry, namely the one pushed at the start. This entry is only used when the complete grammar fails evaluation as can be observed in Equation (3.57). Lastly, the `set_success` instruction has set the parsing machine status flag to 1, indicating a successful parse.

If evaluation of the start non-terminal, and thus the complete grammar, fails evaluation, the parsing machine behaves as presented in the proof shown Equation (3.57). The initial backtrack stack entry is used to restore the parsing machine state to an empty return stack; the character position value set to 0; and, most importantly, the program counter to point to the `set_fail` instruction. After executing this instruction, the parsing machine status flag is set to -1, thereby indicating a failed parse.

$$
\begin{aligned}
&\texttt{if match(}G, A_1, s, 0\texttt{)} = i \texttt{ then}\\
&\langle pc = 0, cp = 0, rs = \varnothing, bs = \varnothing, t = 0\rangle \xrightarrow{\texttt{s\_push}} \langle 1\,,\ 0\,,\ \varnothing\,,\ (0,3,\varnothing)\,,\ 0\rangle\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \xrightarrow{\texttt{call}} \langle 4\,,\ 0\,,\ (2)\,,\ (0,3,\varnothing)\,,\ 0\rangle\\
&\qquad\qquad\qquad \xrightarrow{\Pi(G,4,\texttt{ntmap(}G,A_1\texttt{))}} \langle 4 + |\Pi(G, x, \texttt{ntmap(}G, A_1\texttt{))}|\ ,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad i\,,\ (2)\,,\ (0,3,\varnothing)\,,\ 0\rangle\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \xrightarrow{\texttt{ret}} \langle 2\,,\ i\,,\ \varnothing\,,\ (0,3,\varnothing)\,,\ 0\rangle\\
&\qquad\qquad\qquad\qquad\qquad \xrightarrow{\texttt{set\_success}} \langle 2\,,\ i\,,\ \varnothing\,,\ (0,3,\varnothing)\,,\ 1\rangle
\end{aligned}
\tag{3.56}
$$

$$\begin{aligned}
&\texttt{if match}(G, A_1, s, 0) = \varnothing \texttt{ then}\\
&\langle pc = 0, cp = 0, rs = \varnothing, bs = \varnothing, t = 0\rangle \xrightarrow{\texttt{s\_push}} \langle 1\,,\ 0\,,\ \varnothing\,,\ (0,3,\varnothing)\,,\ 0\rangle\\
&\qquad\qquad\qquad\qquad\qquad\qquad\xrightarrow{\texttt{call}} \langle 4\,,\ 0\,,\ (2)\,,\ (0,3,\varnothing)\,,\ 0\rangle\\
&\qquad\qquad\qquad\xrightarrow{\Pi(G,4,\texttt{ntmap}(G,A_1))} \langle 3\,,\ 0\,,\ \varnothing\,,\ \varnothing\,,\ 0\rangle\\
&\qquad\qquad\qquad\qquad\qquad\qquad\xrightarrow{\texttt{set\_fail}} \langle 3\,,\ 0\,,\ \varnothing\,,\ \varnothing\,,\ -1\rangle
\end{aligned} \tag{3.57}$$

3.6 Conclusion

In order to design a suitable parsing machine that operates according to a PEG, the most basic components and concepts required for its implementation were explored, such as how limited backtracking and non-terminal calls are handled (see Section 3.1). These concepts were employed for an inductive formalization of the top-down parsing algorithm inherent to PEG (see Section 3.2).

The design of the parsing machine itself begun with a description of its micro-architecture, which resulted from translating the formalized PEG expressions into microcode describing the flow of data between its fundamental components (see Section 3.3). The microcode was then split and grouped in such a way, that a minimal instruction set could be defined. The microcode definition of these instructions then translated to a mathematical formalization, which defines a parsing machine state transition for each PPEG instruction (see Section 3.4).

Based on the PPEG instruction set and the original microcode, the translation from PEG expression to PPEG assembly code is formalized. Finally, the PEG-PPEG translation formalization combined with the PPEG instruction set formalization were used to verify the parsing machine's operation with respect to the earlier defined inductive formalization of PEG expressions (see Section 3.5).

Software Tools 4

Due to the similarities of the PPEG parsing machine presented in Chapter 3 with conventional computer architectures, similar tools can be developed to analyze its function and to enhance user experience. This chapter discusses three of such tools developed for the PPEG parsing machine architecture.

Section 4.1 introduces a software-based implementation of the PPEG architecture, which is developed to analyze the internal processes when parsing a given text. Section 4.2 explains PEG compiler and PPEG assembler tools to transform a PEG into an equivalent binary program that can be run on the PPEG architecture.

4.1 PPEG Virtual Parsing Machine

Before attempting to implement the PPEG architecture directly into (reconfigurable) hardware, a proof-of-concept was developed in the Python programming language. The term virtual parsing machine (VPM) is used to refer to this PPEG architecture implementation. Its implementation in Python is explained in short in Section 4.1.1. Besides working as a proof-of-concept, the VPM is useful as a profiling tool, which is discussed in Section 4.1.2. Finally, its implementation in Python allows for rapid development of additional features, the most interesting of which is explored in Section 4.1.3.

4.1.1 VPM Implementation

The Python implementation of the PPEG parsing machine closely follows the design outlined in Chapter 3. For this reason, no complex Python constructs and libraries are employed, but only simple imperative code is used in its implementation.

The VPM is implemented as a single class consisting of only five methods, namely:

- *Class Constructor*: creates and initializes class properties that represents the parsing machine state. As can be observed in the formal parsing machine state definition in Section 3.4.3, this contained five state variables (i.e., $\langle pc, cp, rs, bs, t \rangle$). Therefore, the constructor defines program counter and character position variables; two lists of parameterized lengths representing the return stack and backtrack stack, along with two variables for the return stack pointer and backtrack stack pointer; and one state variable that represents parse state having one of three values: failed parse, successful parse, and active parse. Finally, two binary arrays of parameterized lengths are initialized, one with 8-bit elements and the other with 32-bit elements, which represent data memory and instruction memory respectively.

- `reset`: resets the parsing machine state variables. This entails setting the program counter and character position variables to zero; setting the return and backtrack stack pointers to their respective stack sizes minus one; and setting the parse state variable to "active parse".
- `load_bytecode`: reads the raw binary contents of a file representing PPEG bytecode, or binary machine code, of a compiled PEG grammar. The binary contents are read as 32-bit words, each representing a single PPEG instruction (see Section 3.4.2). The binary words are written directly to the binary array variable representing the instruction memory starting at index 0.
- `load_data`: reads the raw binary contents of a file representing the input string that will be parsed by the VPM. The binary contents are read as 8-bit words (bytes), each representing a single (ASCII) character (see Section 3.1.1.2). Similar to the `load_bytecode` method, these bytes are written directly to the binary array variable representing the data memory starting at index 0.
- `advance_clock_cycle`: executes the PPEG instruction pointed to by the current program counter value according to its definition in Section 3.4.3. The instruction decoding process is performed based on the instruction encoding format discussed in 3.4.2, such that program state can be transformed based the opcode and operand fields. For the purpose of analysis, a boolean value is returned indicating if the execution of the current "clock cycle" resulted in a backtrack operation.

The first four methods of the VPM are used to set everything up before parsing a file with a compiled PEG grammar. The final method can then be called in a loop, thereby continuously executing instructions, until the parse state variable is no longer set to "active parse". The parse result can finally be read from the parse state variable, which is either "successful parse" or "failed parse".

4.1.2 VPM as Profiling Tool

The fact that the PPEG parsing machine is implemented in software makes it much easier to dissect ongoing internal processes during a parse. For this reason, the VPM codebase is interspersed with counter and flag variables to keep track of a number of possibly important events.

For example, the number of passed clock cycles are tracked as well as the total execution time. These can be used for comparison with the hardware-based PPEG parsing machine implementation. The final clock cycle count must naturally be equivalent between both implementations, but the execution time should in theory be higher for the hardware-based implementation.

Other parse results cannot so easily be compared between software and hardware implementations. However, if both follow the expected behavior defined in the PPEG ISA specification, these are transferable to the hardware-based implementation. These parse results are the number of backtracks, number of non-terminal calls, the maximum used backtrack stack entries and return stack entries, and the farthest jump in character position value caused by a backtrack operation.

Of special attention are the maximum used backtrack stack entries and return stack entries. The reason is that the size of the backtrack stack and return stack is not easily determined without reference. By parsing files that are common for a specific use case, these values can be used make a good estimate for the required sizes of the stacks.

During the parsing process, the VPM also keeps track of successfully evaluated non-terminals, thereby creating the parse tree. On successful parses, the VPM can then report the parse tree for post processing and visualization of the parsed files (see Listing 4.2). Furthermore, the VPM keeps track of the longest path. On successful parses, this is obviously equal to the length of the parsed input file. However, on failed parses, this provides the user with insights into where the parser got stuck and therefore where the parsed file is possibly invalid with regard to the compiled PEG.

4.1.3 Memoization Unit

As mentioned earlier, the implementation of the PPEG parsing machine in the Python programming language allows for rapid prototyping. For this reasons, extensions to the base PPEG architecture can be quickly realized and analyzed for potential improvements in parsing. One such extension is the memoization unit.

Memoization is the process of storing non-terminal evaluation results in a lookup table to save later re-evaluation after a backtrack operation, and was introduced in Section 2.2.2.4. At worst, all non-terminals can have a successful or unsuccessful parse at any character position in the input string, thereby requiring a $m \times n$ lookup table, where m is the number of non-terminals and n is the length of the input string. The memory required to implement this lookup table is too much for any reasonable string lengths in a hardware-based implementation.

To solve this problem, the lookup table is implemented as a p-way set-associative cache with m sets and least-recently used (LRU) replacement policy. Assuming $p = 1$, each non-terminal in the PEG has only one cell in the lookup table to store the most recent evaluation result. Consider the following PEG production rule:

$$A \leftarrow B\ C\ /\ B\ D$$

A successful evaluation of non-terminal B stores the resulting new character position at the single cache set for non-terminal B. If evaluation of non-terminal C fails, a new call to non-terminal B is prevented by checking the cache set for B and checking if the current character position matches with the set's character position (cache tag field). In this case, the positions match and non-terminal B therefore does not need to be re-evaluated. Note the importance to check the cache set's character position with the current character position, as otherwise the following PEG production rule would wrongly result in a cache hit:

$$A \leftarrow B\ C\ /\ a\ B\ D$$

For the previous PEG example, a 1-way set-associative cache would suffice. However, consider this time the the following PEG production rule:

$$A \leftarrow B\ B\ C\ /\ B\ B\ D$$

In this case, the second call to non-terminal B would overwrite the result of the first call in cache for the 1-way cache. Therefore, on failure to evaluate non-terminal C, non-terminal B needs to be re-evaluated twice before reaching non-terminal D. This can be solved by extending the cache to a 2-way set-associative cache configuration, such that two evaluation results of non-terminal B can be stored at a given time.

Storing indications of failed non-terminal evaluations in memoization cache can also readily speed up the parsing process. Take the following PEG production rule as an example:

$$A \leftarrow B\ C\ /\ B\ D\ /\ E$$

If evaluation of non-terminal B at the current character position fails, an indication of this failed evaluation is written to cache at the cache set for non-terminal B. Thereafter, the second alternate rule is evaluated. Here, instead of re-evaluating non-terminal B a second time at the same character position, a cache read finds the failed evaluation result for that non-terminal and position and initiates an immediate backtrack operation. Finally, the last alternate rule is evaluated, namely non-terminal E.

Note that caching failed non-terminal calls is not as straightforward as caching successful non-terminal calls. For example, take the following PEG production rules:

$$A \leftarrow a\ B$$
$$B \leftarrow b\ C$$

Evaluation of non-terminal A leads to evaluation of non-terminal B, which in turn leads to evaluation of non-terminal C. Assuming evaluation of non-terminal C fails, evaluation of non-terminals B and A fail by association. Following the previous example, this requires three separate writes to cache: an indication of failed evaluation for each of the three non-terminals.

In general, on a backtrack operation, a cache write is required for each non-terminal associated with the top return addresses in the return stack up to (but excluding) the return address in the top backtrack stack entry. This might be difficult to implement in hardware as the number of memory writes per clock cycle is limited. This could be solved by stalling the parsing machine until all cache writes have been performed, but such pauses in parsing may inhibit the parsing more than the cache is able to speed it up.

Finally, it is important to observe that for any given PEG with m non-terminals and input string with length n, the worst case scenario requires an n-way set-associative cache with m sets. However, in practice due to the principle of locality, even a set associativity of 1 of the memoization cache can result in significant speedups [26], although the speedup differs per PEG and input string. An optimal cache configuration with respect to speedup and hardware requirements can be obtained by analyzing the results for varying cache configurations in the VPM. indicates that a

4.2 Compilation and Assembly Tools

The process of translating a PEG grammar definition into a binary executable file that can be run on the PPEG architecture consists of two subprocesses: compilation and assembly. The first is the process of translating a PPEG grammar definition into a PPEG assembly program and the latter is the translation of this assembly program into a binary executable file. The compilation process is discussed in Section 4.2.1 and the assembly process is discussed in Section 4.2.2.

4.2.1 PEG Compiler

Construction of the PEG compiler is not much different from conventional compilers. A PEG grammar can be viewed as a programming language and PPEG assembly is not much different than general-purpose computer assembly. For this reason, the compilation approach is similar as presented in the ubiquitous Dragon Book [45]. A diagram outlining this process is shown in Figure 4.1, which shows the components in the compiler pipeline (boxes) and the intermediate representations (arrows).

In the following subsections, the compiler components and intermediate representations are discussed in more detail. The explanations are supported by means of an example of PEG grammar compilation.

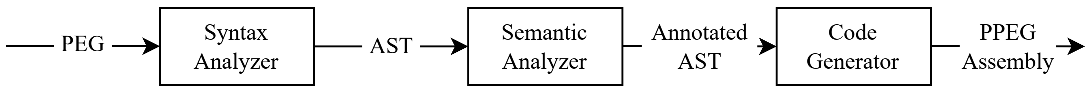


Figure 4.1: PEG compilation steps and intermediate representations.

4.2.1.1 Syntax Analyzer

The first stage in the PEG compiler is the parsing of a PEG grammar and constructing a corresponding parse tree or abstract syntax tree (AST) (see Section 2.2.1), which is the job of the syntax analyzer. In the PEG compiler, the parser that is used by the syntax analyzer is the virtual PEG parsing machine discussed in Section 4.1. The PEG grammar that is used to parse a PEG grammar is the PEG grammar listed in Appendix B, which is the PEG grammar that describes the PEG syntax in the original paper about PEG [3].

Note that, because this syntax analyzer is based on PEG, which can inherently describe lexical syntax, there is no need for a separate lexical analyzer component before the syntax analyzer in the compiler pipeline diagram shown in Figure 4.1.

```
Grammar
0
20
|
+-------------------------------------------+
|                                           |
Definition                                  Definition
0                                           8
8                                           20
|                                           |
+--------------------+----------+           +--------------------+----------+
|                    |          |           |                    |          |
Identifier           LEFTARROW  Expression  Identifier           LEFTARROW  Expression
0                    2          5           8                    10         13
2                    5          8           10                   13         20
|                    |          |           |                    |          |
+-----------+        +          +           +-----------+        +          +
|           |        |          |           |           |        |          |
IdentStart  Spacing  Spacing    Sequence    IdentStart  Spacing  Spacing    Sequence
0           1        4          5           8           9        12         13
1           2        5          8           9           10       13         20
            |        |          |                       |        |          |
            +        +          +                       +        +          +
            |        |          |                       |        |          |
            Space    Space      Prefix                  Space    Space      Prefix
            1        4          5                       9        12         13
            2        5          8                       10       13         20
                                |                                           |
                                +                                           +
                                |                                           |
                                Suffix                                      Suffix
                                5                                           13
                                8                                           20
                                |                                           |
                                +                                           +
                                |                                           |
                                Primary                                     Primary
                                5                                           13
                                8                                           20
                                |                                           |
                                +                                           +
                                |                                           |
                                Identifier                                  Class
                                5                                           13
                                8                                           20
                                |                                           |
                                +-----------+                               +-----------+
                                |           |                               |           |
                                IdentStart  Spacing                         Range       Spacing
                                5           6                               14          18
                                6           8                               17          20
                                            |                               |           |
                                            +                               +-----+     +
                                            |                               |     |     |
                                            Space                           Char  Char  Space
                                            6                               14    16    18
                                            8                               15    17    20
                                            |                                           |
                                            +                                           +
                                            |                                           |
                                            EndOfLine                                   EndOfLine
                                            6                                           18
                                            8                                           20
```

Listing 4.1: Abstract syntax tree of PEG grammar from Listing 4.2.

For an example of the compilation process, the PEG grammar presented in Listing 4.2 is used. The start non-terminal `A` simply defines a non-terminal PEG expression `B`. In turn, non-terminal `B` defines a single character class PEG expression consisting only of the lowercase ASCII characters. When this grammar is parsed by the virtual PEG parsing machine, the AST shown in Listing 4.1 is generated. In top-to-bottom order, the AST nodes consist of: non-terminal identifier; position of the first character of the parsed substring with regard to the complete input string; position of the last character plus one of the parsed substring. The printed character positions are in accordance with the PEG example grammar as character stream shown in Listing 4.3, where the '`\r`' and '`\n`' symbols represent the ASCII escape codes for carriage return and line feed respectively.

```
A <- B
B <- [a-z]
```

Listing 4.2: PEG grammar example.

```
position: 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
string:    A     <  -     B \r \n  B     <  -     [  a  -  z  ] \r \n
```

Listing 4.3: PEG grammar from Listing 4.2 as character stream.

Note that the AST in Listing 4.1 as generated by the syntax analyzer only provides information regarding the syntactic and hierarchical structure of the parsed PEG grammar in Listing 4.2. For example, it specifies that the expression belonging to non-terminal `B` spans character positions 13 up to, but not including, 20. The `Expression` node itself presents a path to a `Range` node, containing the two characters at position 14 and 16, which match to 'a' and 'z' respectively according to Listing 4.3.

4.2.1.2 Semantic Analyzer

The second stage in the PEG compiler is the semantic analyzer component, which takes an AST as input and produces an annotated AST as output. This achieved by applying a depth-first search through the abstract syntax tree. During this traversal, the nodes and child nodes are matched with predefined patterns that translate to partial annotated abstract syntax trees representing PEG expressions. The location of these matches are used to construct a new annotated abstract syntax tree that thus contains semantic information regarding the parsed PEG grammar.

A subset of AST match patterns is shown in Listings 4.4 to 4.6. The partial AST on the left represents the AST pattern which, when matched, translates to the partial annotated AST on the right.

```
|                                              |
Sequence                                       Sequence
|                                              |
+-------+-------+-----           -->           +---+---+--
|       |       |                              |   |   |
Prefix  Prefix  Prefix
|       |       |
```

Listing 4.4: Sequence PEG expression AST translation.

```
|                                              |
Prefix                           -->           AndPredicate
|                                              |
+----+
|    |
AND  Suffix
     |
```

Listing 4.5: And-predicate PEG expression AST translation.

```
|                                              |
Suffix                           -->           Optional
|                                              |
+--------+
|        |
Primary  QUESTION
|
```

Listing 4.6: Optional PEG expression AST translation.

```
|                                              |
Primary                          -->           LiteralString
|                                              {literal = "c1 c2 c3 ..."}
+
|
Literal
|
+-----+-----+---
|     |     |
Char  Char  Char
```

Listing 4.7: Literal string PEG expression AST translation.

Some AST patterns are translated to an annotated AST with additional semantic information. An example of this can be seen in the annotated AST associated with a literal string PEG expression presented in Listing 4.7. The `LiteralString` node in the annotated AST has a `literal` property that contains the string of characters associated with the `Char` nodes in the AST. Other annotated AST nodes that include an additional property are the non-terminal definition and expression, which both store the non-terminal identifier, and the character class expression, which stores the character class itself.

Continuing with the example of compiling the PEG grammar in Listing 4.2, the semantic analyzer takes the AST in Listing 4.1 produced by the syntax analyzer as input. By using the process described earlier, the annotated AST shown in Listing 4.8 is produced.

```
Grammar
|
+------------+
|            |
Definition   Definition
{id = "A"}   {id = "B"}
|            |
NonTerminal  CharacterClass
{id = "B"}   {class = [[97, 122]]}
```

Listing 4.8: Annotated abstract syntax tree of PEG grammar from Listing 4.2.

4.2.1.3 Code Generator

The last stage in the PEG compiler is the PPEG assembly code generation, which is the job of the code generator. This component applies a depth-first search through an annotated AST and uses the recursive PEG expression to PPEG code translation function $\Pi(G, pc, e)$, which was introduced in Section 3.5, in order to generate the associated PPEG code. The depth-first search traversal causes the code to be generated linearly with respect to the line numbers. In short, the code generator component is a practical implementation of the formal grammar translation function $\Pi'(G)$.

For the example PEG grammar in Listing 4.2, the code generator takes the annotated AST in Listing 4.8 produced by the semantic analyzer and produces the PPEG assembly program shown in Listing 4.9. Here, the translation from PEG grammar to PPEG instructions can be readily observed. For example, the first 4 instructions are as defined by $\Pi'(G)$ in Section 3.5.3, which precede the PPEG code associated with non-terminals starting in lines 6 and 10. Similarly, the the `NonTerminal` and `CharacterClass` annotated AST nodes translate to line 6 by means of $\Pi(G, x, \texttt{B})$ and to line 9 by means of $\Pi(G, x, \texttt{[a-z]})$ respectively.

```
 1      s_push $cpos, L0
 2      call A
 3      set_success
 4  L0: set_fail
 5
 6  A:  call B
 7      ret
 8
 9  B:  char_set ($cpos)+, 97, 122
10      ret
```

Listing 4.9: PPEG assembly program of PEG grammar from Listing 4.2.

Note that, though the Π' and Π translation functions are defined such as to generate PPEG assembly code with absolute instruction addresses, the assembly code generated by the code generator substitutes these with labels. The reason is simply the increase in legibility for the user.

4.2.2 PPEG Assembler

The final process in the path to generating a binary executable file that can be run on a PPEG parsing machine is the job of the PPEG assembler. Its input is any valid PPEG assembly program.

The generation of the binary machine code is achieved by two linear passes over the PPEG assembly code. The first pass is used to resolve any label references into their corresponding absolute or relative instruction memory address. The second pass translates each line from PPEG assembly word to its 32-bit binary equivalent according to the PPEG ISA specification in Section 3.4.

```
addr  |  31          PPEG binary          0  |  PPEG assembly
------+--------------------------------------+------------------------------
0     |  10100000 00000000 00000000 00000011  |  s_push $cpos, 3
1     |  10000000 00000000 00000000 00000100  |  call 4
2     |  00010000 00000000 00000000 00000000  |  set_success
3     |  00001000 00000000 00000000 00000000  |  set_fail
4     |  10000000 00000000 00000000 00000110  |  call 6
5     |  00011000 00000000 00000000 00000000  |  ret
6     |  01000000 01111010 01100001 00000000  |  char_set ($cpos)+, 97, 122
7     |  00011000 00000000 00000000 00000000  |  ret
```

Listing 4.10: PPEG binary program of PEG grammar from Listing 4.2.

The assembly process is clarified with a continuation on the example presented in Section 4.2.1. Consider the PPEG assembly program shown in Listing 4.9. After the first pass that resolves the labels `A`, `B`, and `L0`, the assembly program in the right-most column of Listing 4.10 is obtained. Thereafter, using the PPEG ISA specification during the second pass, the assembler translates the assembly program on the right to the PPEG binary program shown in the middle column of Listing 4.10. The left-most column shows the instruction memory address associated with the binary and assembly instruction words on the right.

4.3 Conclusion

This chapter explained three main software tools developed for use with the PPEG architecture. The first is a VPM or virtual parsing machine. That is, a software emulator of the PPEG parsing machine architecture was built based on the microcode definitions of the PPEG ISA. The emulator is extended with tools to analyze internal processes and ongoing events. Moreover, parse tree visualization and parse error reporting are included in the VPM. Finally, a memoization extension was added in order to analyze its effect on parses while accounting for the required hardware complexity (see Section 4.1).

The other two tools that were developed are the PEG compiler and PPEG assembler (see Section 4.2). The first takes an arbitrary PEG and compiles it into an equivalent PPEG assembly program. This is achieved by a number of transformations by means of a PEG syntax analyzer, PEG semantic analyzer, and finally PPEG code generator which translates an annotated AST into a PPEG assembly program according to the PEG translation model discussed in Section 3.5. Finally, the PPEG assembly program is transformed to a binary file by means of the PPEG assembler tool, which is based on the PPEG ISA specification discussed in Section 3.4.

5 Results and Analysis

Based on the PPEG architecture design presented in Chapter 3, this chapter aims to measure and analyze the performance and parse behavior of the PPEG parsing machine. This is achieved by running benchmarks on the virtual parsing machine profiling tool discussed in Section 4.1.2 and on an FPGA-based implementation of the PPEG architecture.

Section 5.1 discusses the the hardware setup for both the virtual parsing machine and FPGA-based implementation, the benchmarks, and the measurement procedures. Thereafter, the benchmark measurement results are analyzed in Section 5.2, which includes analysis of parsing throughput, stack size, and clock cycle distribution. Finally, the efficacy of a memoization cache in the virtual parsing machine is measured and discussed in Section 5.3.

5.1 Test Setup

In order to analyze the behavior and performance of the PPEG parsing machine designed in Chapter 3, a number of important elements need to be specified first, such as the hardware that is used to run the benchmarks on, the construction of the benchmarks, and the measurement procedure. This section aims to provide a complete overview of these elements.

5.1.1 Hardware Setup

Virtual Parsing Machine
For the purpose of benchmark measurements, the PPEG virtual parsing machine runs on a Dell Latitude 5580 laptop with an Intel Core i7-7600U CPU at 2.80GHz and 16GB of RAM. The operating system is Windows 10 version 21H1. Finally, the virtual parsing machine is implemented in Python and runs on Python version 3.10.1.

FPGA-based PPEG Parsing Machine
The PPEG parsing machine architecture, as specified in Chapter 3, is implemented on an FPGA (field-programmable gate array) for the purpose of running benchmark measurements. The specific FPGA that was used is the Intel Arria 10 (product code: 10AS027E4F29E3SG) on the Mercury+ AA1 SoC module from Enclustra.

PPEG architecture was implemented on the FPGA with the following configuration:

- $2^8 = 256$ return stack entries.

- $2^6 = 64$ backtrack stack entries.
- 12-bit instruction memory address width; equivalent to $2^{12} = 4096$ instruction words or $4096 \times 4 = 16$KB.
- 18-bit data memory address width; equivalent to $2^{18} = 262{,}144$ bytes or, equivalently, 262,144 characters.

The return and backtrack stack size were selected based on VPM benchmark results. The instruction memory size was selected based on the binary code size of the largest benchmark grammar. The data memory size was set as large as possible, while keeping the target system clock frequency at 100MHz.

Table 5.1: Usage of FPGA resources for implementation of PPEG core.

Resource	**Usage**
ALMs	2040
Combinational ALUTs	1394
Dedicated Logic Registers	2589
DSP Blocks	0
Block Memory Bits	5504

Using this PPEG configuration, synthesis and implementation for the Intel Arria 10 FPGA results in the resource usage presented as reference in Table 5.1. Note that this only accounts for the PPEG core, which excludes instruction memory and data memory. Because of this, the number of block memory bits represents the combined bit-size of the return stack and backtrack stack. A single return stack entry requires 12 bits (instruction memory address). Therefore, the complete return stack requires 2^8 entries $\times$ 12 bits $=$ 3072 bits. Similarly, a single backtrack entry requires (return stack address + instruction memory address + data memory address) $=$ 8 bits + 12 bits + 18 bits $=$ 38 bits. Therefore, the complete backtrack stack requires 2^6 entries$\times$38 bits $=$ 2432 bits. The total required block memory bits is therefore indeed 3072 bits + 2432 bits $=$ 5504 bits.

Besides the PPEG core and instruction and data memory, the FPGA-based implementation also includes a so-called system controller, which provides an interface between the PPEG system and the outside. Communication between the outside is based on read and write commands via the UART serial protocol. The PPEG status register, file size register, enable pin, and restart pin are all memory mapped, together with the instruction and data memory. This way, to for example set the enable pin, a simple write command to its corresponding memory address is needed. Likewise, reading the PPEG core status simply requires a memory read command to its memory address. The system controller provides an easy and intuitive method for communication between the PPEG system and an external controller. The complete FPGA-based PPEG system can be seen in Figure 5.1.

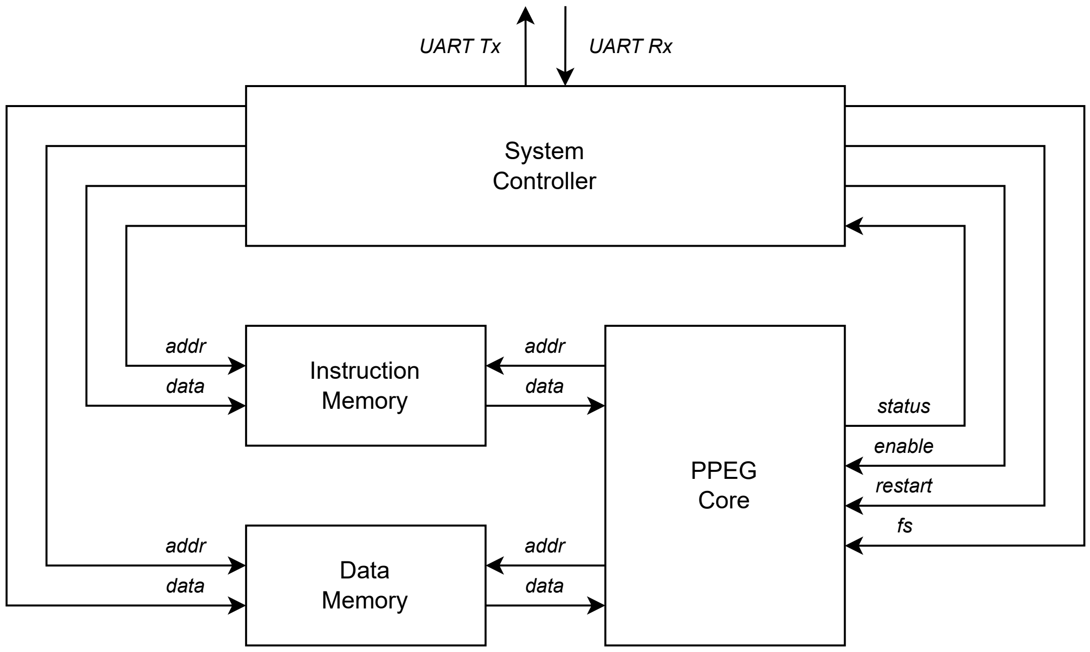


Figure 5.1: FPGA-based system overview.

5.1.2 Reference Implementations

In order to suitably analyze the performance and operation of the PPEG parsing machine, the ability to compare to similar existing implementations is key. Although no parsing machine has been designed for and implemented on hardware, solutions that come close are parsing machines implemented as virtual machines. Of these, LPEG, MiniNez, and GPEG are the most notable ones, as discussed in 2.5.3.

Unfortunately, LPEG is not a parsing machine based on compiling PEG into binary machine code [38]. Instead, grammars are constructed as Lua code with the LPEG library. Moreover, the LPEG virtual machine is intended as a pattern matching device and not an actual parser. For these reasons, only MiniNez and GPEG implementations are considered when comparing benchmark results, as both these implementations are compiled and intended as recognizers or parsers.

5.1.3 Benchmark Files

In order to obtain useful data about the performance and operation of the PPEG parsing machine, a set of grammars and files for parsing are needed. Preferably, these grammars and files are used by the reference implementations in order to more easily compare benchmark measurement results.

Regarding the file sets used by the benchmarks, the MiniNez paper performs tests on a CSV data set and JSON data set obtained from an unspecified repository at the USGS (United States Geological Survey) website. Furthermore, its synthetic XML file set was generated by the `xmlgen` data generator tool, originally created for the XMark benchmark project [46]. The MiniNez paper also uses a syslog data set and email data set, but the data sets for these file types could not be reproduced. The GPEG paper only performs tests on an unspecified synthetic JSON data set and on a Java dataset obtained from the OpenJDK7 source code. Because the source code for OpenJDK7 could not be found, instead a Java dataset was constructed from the Oracle JDK5 source code [2].

Regarding the grammars used by the benchmarks, the MiniNez paper specifies PEGs for CSV, XML, JSON. The open source code for the GPEG project contains a PEG for Java 1.7, which was originally written by Redziejowski [47].

In the end, 6 separate data sets were created with 4 unique file types: CSV, XML, JSON, and Java. The The single CSV data set was obtained from the USGS Earthquake Hazards Program, which keeps a catalog of past earthquakes from which data can be requested in various file formats. Similar to the CSV data set, one XML data set and one JSON data set were also obtained from the USGS Earthquake Hazards Program. Another XML data set was obtained using the `xmlgen` data generator tool. This XML data set was then converted to JSON data using a Python script to produce a second JSON data set. Finally, as mentioned earlier, a Java data set was constructed from the Oracle JDK5 source code. Each data set consists of 60 files with file sizes ranging between 1KiB and 256KiB, the maximum that fits in instruction memory as specified in Section 5.1.1.

The 6 data sets that are used for the benchmarks provide a reasonable mix of smaller and larger PEGs. Moreover, the XML and JSON files generated by the `xmlgen` tool provide synthetic data sets, whereas the CSV, XML, and JSON files obtained from the USGS Earthquake Hazards Program provide good data sets with real-world data.

5.1.4 Measurement Procedure

The virtual parsing machine and FPGA-based parsing machine have two completely different ways to measure performance and other metrics, which are discussed below.

Virtual Parsing Machine
As previously discussed in Section 4.1.2, the VPM can be used as a profiling tool in order to gather as much data regarding parsing flow, memory utilization, throughput, and more. After loading the binary machine code and data file into separate binary arrays in Python, the following metrics are measured during each parse and used for the benchmarks:

- file size
- number of clock cycles
- number of backtrack operations
- number of non-terminal calls
- number of redundant non-terminal calls
- maximum return stack size

- maximum backtrack stack size
- parse time

Of these, only the measurement of the parse time requires a more detailed account for the purpose reproducibility. Because the VPM is implemented in Python, the `time` module is used from the Python Standard Library. More specifically, the function `time.process_time()` is called right before and after the parse of the file in question. This function returns the sum of the system and user CPU time of the current process in fractional seconds. By subtracting the return value at the end of the parse with that at the beginning of the parse, the total parse time is computed.

Note that he VPM parse time is not very accurate, in part because it includes the small measurement time of other benchmark metrics, but also because the measured parse time seems to be influenced by the CPU's utilization at the moment of running the benchmarks. However, the VPM parse time can still be used for comparisons in e.g. orders of magnitude.

FPGA-based PPEG Parsing Machine
Before running the parsing machine on the FPGA, the binary machine code and data file for the grammar and file in question are written to memory using the UART command interface as discussed in 5.1.1. Thereafter, the parsing machine is restarted and enabled, which starts its execution at instruction memory address zero. Because most benchmark metrics can be measured by means of the VPM as mentioned before, the only metric that is measured in order to obtain the parsing machine's performance is number of executed clock cycles. An internal counter is included in the PPEG system, which only counts when the PPEG core is enabled and until the status changes from "busy" to some other status. At the end of the parse the total number of executed clock cycles can be obtained by means of a read command to the corresponding address.

Because the clock speed is fixed to 100MHz, the parse time can be easily determined by the following formula:

$$Parse\ Time = Clock\ Cycles/Clock\ Frequency$$

Note that, because the PPEG parsing machine is completely deterministic, the number of clock cycles spent on parsing a particular file is always the same. Therefore, each file only needs to be parsed once.

5.2 Benchmark Measurements

Using the virtual parsing machine analytics as discussed in Section 4.1.2, a number of interesting benchmark metrics come to light, such as parsing time, stack sizes, clock cycle distribution, and more. Moreover, this section discusses the performance of the FPGA-based PPEG implementation, as described in Section 5.1, and compares it with the virtual implementation as well as other parsing machine implementations.

5.2.1 Parse Time

As is to be expected, the discrepancy in parse time when running the benchmark files on the FPGA-based implementation and VPM is substantial. Across all benchmarks, a 2 - 2.5 magnitude difference is found between these implementations.

Figure 5.2a shows the parse time distribution for the 60 JSON benchmark files sourced from the USGS Earthquake Hazards Program. The dashed gray lines are linear projections obtained by applying least-squares regression to the measurement data. From this it can be observed that the parse time appears to be linear with increasing input file size. Appendix C contains the parse time versus input size plots for the other five benchmarks.

5.2.2 Stack Size

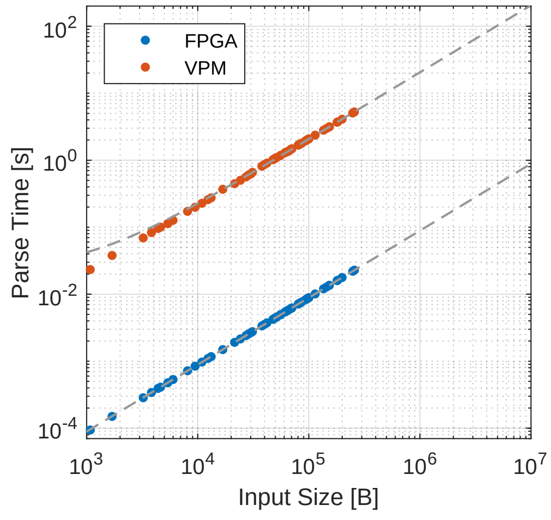
When running the benchmarks on the VPM, it keeps track of the maximum number of return and backtrack stack entries that were stored per file. The resulting maximum stack size distribution for the JSON benchmark files sourced from the USGS Earthquake Hazards Program is presented in Figure 5.2c. It can be observed that the required stack size is almost constant for all input file sizes and from which it can be concluded that stack size does not directly depend on input file size. Similar stack size versus input file size plots are presented for the other five benchmarks in Appendix C.

Table 5.2 reports the maximum return and backtrack stack values across all files per benchmark. As might be expected, smaller less complex grammars like the PEG for CSV requires fewer stack entries than a large and complex grammar like the PEG for Java. Note too that, though required stack size does not depend on input file size, files with the same file type but different structure can have wildly different stack requirements. For example, though both the USGS data repository and `xmlgen` tool provide a XML data sets, the `xmlgen` XML files require almost twice the stack size than the USGS XML files. The same is true for JSON files.

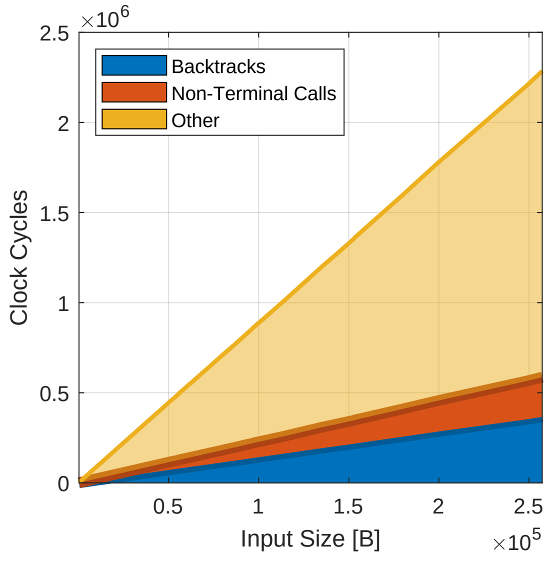
Another interesting result from the stack size measurements is their distribution in case of heterogeneous file sets. The CSV, XML, and JSON data sets are rather homogeneous, which can be observed from the fact that their stack sizes do not deviate much from the mean. This is unlike the heterogeneous Java file set selected from the JDK5 project. The table reports a maximum of 134 return stack entries and 45 backtrack stack entries. However, the mean stack sizes when parsing this file set are only 68 and 27 entries for the return and backtrack stack respectively, or about half of the maximum reported sizes.

From these results it can be concluded that the required return and backtrack stack sizes depend mostly on the size and complexity of the grammar at hand. The file that is parsed determines how much of the complexity is needed for the parse, and therefore the maximum required stack size. In the case of the PPEG parsing machine, if the use case and grammar are known, analysis with the VPM can determine a good estimate for initial stack sizes.

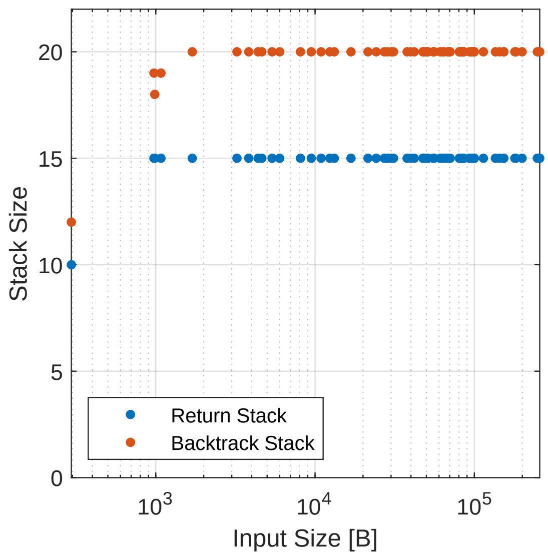
On a final note, it is important to consider that the stack size is given in number of stack



(a) Parse times of benchmark files for virtual parsing machine (VPM) and FPGA implementations.



(b) Clock cycle distribution when parsing benchmark files.



(c) Maximum stack sizes required when parsing benchmark files.

Figure 5.2: Benchmark results for JSON files sourced from USGS Earthquake Hazards Program [1].

entries and not in bytes. In the case of the test setup listed in Section 5.1, a single return stack entry consists of 12 bits, whereas a single backtrack stack entry consists of 38 bits. The size of the stack entries depends on maximum input file size, grammar size, and, in the case of the backtrack stack, on the size of the return stack.

Table 5.2: Maximum return stack and backtrack stack sizes required when parsing the benchmark files.

Benchmark	**USGS**			**xmlgen**		**JDK5**
File Type	**CSV**	**XML**	**JSON**	**XML**	**JSON**	**Java**
Return Stack	4	16	15	28	40	134
Backtrack Stack	6	15	20	27	47	45

5.2.3 Clock Cycle Distribution

Non-terminal calls and backtrack operations heavily influence the execution flow of the PPEG parsing machine. This influence is clearly visible in the clock cycle distribution shown in Figure 5.2b, which presents the data for the JSON benchmark files sourced from the USGS Earthquake Hazard Program. It can be observed that about 10% of the clock cycles are spent on non-terminal calls, which are simply jumps to some other part of instruction memory. Additionally, about 16% of the clock cycles are spent on backtrack operations, each being a combination of a jump in instruction memory and data memory. In total, about 26% of all clock cycles, and therefore parse time, is spent on jumps.

Appendix C presents similar plots showing the clock cycle distribution of the other five benchmarks. Table 5.3 presents the same data as percentage of clock cycles spent on non-terminal calls and backtrack operations for all six benchmarks. Similar to the stack size data presented in Table 5.2, this data varies for different grammars, but also for different file structures with the same file type. Furthermore, the benchmark with the largest fraction of clock cycles spent on executing jumps is the Java JDK5 benchmark with a total of 40.4%.

Table 5.3: Average percentage of clock cycles spent on backtracks and non-terminal calls when parsing the benchmark files.

Benchmark	**USGS**			**xmlgen**		**JDK5**
File Type	**CSV**	**XML**	**JSON**	**XML**	**JSON**	**Java**
Bactracks	15.7	16.3	16.1	19.5	15.0	29.2
Calls	2.1	4.0	9.6	3.8	4.3	11.2

The importance of these statistics lies in the parsing machine memory interface. As explained shortly in Section 5.1.1, the current system uses a limited amount of instruction and data memory in order to fit the entire design, including memory, on a single FPGA. No cache is used to enable faster access to instructions and data as this design already supports single-cycle memory reads and writes at 100 MHz system clock frequency. However, if future designs want to allow parsing larger files with possibly larger grammars, external memory might be needed. This leads to an increase in cost of jumps in instruction memory and data memory to above one clock cycle. It is therefore important to consider the clock cycle distribution when designing new grammars and for future extensions to the parsing machine.

5.2.4 Code Size

Though reducing the binary code size of the compiled PEGs is not necessarily an objective of this project, a comparison between the PPEG binary code and that of the two other parsing machine implementations that are considered, namely MiniNez and GPEG, is still provided in Table 5.4. The MiniNez values were obtained from [4], the GPEG values were generated by running its open source compiler [5] on the five indicated grammars, and the PPEG values were generated by running the PPEG compiler and assembler (see Section 4.2) on the same five grammars. Unfortunately, the MiniNez paper did not provide values for PEG and Java grammars and no compiler was found in its open source repository.

Table 5.4: Code size in bytes of various PEG specifications compiled for three parsing machine implementations.

Grammar	**MiniNez**	**GPEG**	**PPEG**
CSV	84	26	96
XML	366	228	692
JSON	368	478	728
PEG	-	2178	840
Java	-	25740	10980

It can be observed that the PPEG binaries are much larger for the smaller CSV, XML, and JSON grammars, whereas the binaries for the larger PEG and Java grammars are less than half that of the GPEG binaries. This can be attributed to the effect of separately stored character set tables. Both MiniNez and GPEG parsing machines store character sets as 256-bit words, where each bit represents a single 8-bit character. This allows for a single 256-bit comparison when a character must be compared against a character class. However, the code sizes in the table do not include the memory allocation for these character sets.

Taking the JSON grammar as an example, the GPEG compiler reports the allocation for 6 character sets in the character set table. However, GPEG optimizes character sets consisting solely of (7-bit) ASCII characters by storing only the lower 128 bits. Because 5 of the 6 character sets consist of ASCII characters only, the character set table for the JSON grammar requires the allocation of $5 \times 128 + 1 \times 256 = 896$ bits. The character set table in addition to the binary size of the GPEG-compiled JSON grammar requires a total of $896 + 478 = 1374$ bits of memory. Likewise, the GPEG-compiled CSV grammar requires an additional 256-bit word for its single character set, thereby requiring $256 + 26 = 282$ bits of memory. MiniNez requires similar allocation sizes for its own character sets.

Another effect that adds to the binary sizes is padding required for misaligned instruction words. This does not pose a problem in the case of MiniNez and PPEG, because both implementations use a fixed instruction word length (16-bit and 32-bit respectively). However, GPEG instructions have many different lengths, ranging from 8 to 48 bits. By design, these instructions need to be 16-bit aligned, which sometimes requires additional 8-bit padding, which is not included in Table 5.4.

5.2.5 Parsing Throughput

Finally, the parsing throughput is the metric that provides good insight of parser performance, irrespective of input file size, assuming parsing speed increases linearly with increasing file size as seen in Section 5.2.1. Table 5.5 reports the parsing throughput in mebibytes per second [MiB/s] for 4 different file types and 4 implementations, which include the VPM-based and FPGA-based PPEG implementation. The MiniNez [4] and GPEG [5] throughput values were obtained from their respective papers, which unfortunately do not report figures for all file types. The PPEG throughput values for XML and JSON are the average of the data sets obtained from the USGS repository and `xmlgen` tool.

Though discussed before in Section 5.2.1, note the substantial difference in throughput between VPM-based and FPGA-based PPEG implementations. Moreover, the the FPGA-based PPEG implementation reports an approximately 3× higher throughput than the MiniNez implementation for the CSV, XML, and JSON grammars. Only the GPEG implementation outperforms the FPGA-based PPEG architecture, namely for the JSON file with a 5× higher throughput.

Table 5.5: Parsing throughput in MiB/s of various benchmarks for four parsing machine implementations.

Grammar	**MiniNez**	**GPEG**	**PPEG (VPM)**	**PPEG (FPGA)**
CSV	6.18	-	0.06	14.88
XML	5.88	-	0.06	15.82
JSON	2.99	54.50	0.05	11.44
Java	-	7.34	0.04	9.50

Unfortunately, because the MiniNez and GPEG parsing machine implementations are based on virtual machines, the hardware used to run these virtual machines on determines much of the throughput. For example, the throughput values for the MiniNez implementation were obtained by running its virtual machine on a Raspberry Pi 2 Model B (4-core Cortex-A7 running at 900MHz), whereas the GPEG virtual machine ran on a AMD Ryzen 5 1600 (6-core CPU running at 3.2GHz). Assuming the virtual machines ran uninterrupted on a single core of their respective processors at the indicated base frequency, Table 5.6 reports the benchmark throughput in bytes per clock cycle. These values were computed based on the values in Table 5.5 using the following formula:

$$\textit{Throughput}\ [\text{B/clock cycle}] = \textit{Throughput}\ [\text{MiB/s}] \times 2^{20}\ [\text{B/MiB}]/\textit{Clock Frequency}\ [\text{Hz}]$$

From this table it can be observed that throughput by the PPEG implementation is almost a magnitude larger than that of GPEG and 1.5 magnitude larger than that of MiniNez. However, note that these throughput values for the MiniNez and GPEG implementations depend on some naive assumptions and furthermore still depend on the target hardware (ISA, instructions per clock cycle, etc.).

Table 5.6: Parsing throughput in bytes per clock cycle [B/clock cycle] of various benchmarks for three parsing machine implementations.

Grammar	**MiniNez**	**GPEG**	**PPEG**
CSV	0.0072	-	0.156
XML	0.0069	-	0.166
JSON	0.0035	0.0179	0.120
Java	-	0.0024	0.100

In general, it is difficult to accurately compare the throughput of the PPEG parsing machine with the two virtual machine implementations, because of their dependence on existing hardware. Only if they were to be implemented in hardware similar to the PPEG parsing machine would such a comparison be possible. However, the MiniNez and GPEG architectures were both designed as a virtual machine, and their design is therefore not optimized for an implementation in hardware.

5.3 Memoization Measurements

Section 4.1.3 discussed a possible performance optimization methodology called memoization that involves caching non-terminal cache results. Using the benchmark files that were introduced in Section 5.1.3, the effectiveness of memoization may be measured for a number of PEG grammars and a variety of cache configurations.

5.3.1 Memoization Metrics

To provide meaningful insight into the effectiveness of memoization agnostic of PEG parser implementation, the number of redundant non-terminals calls is measured during benchmark execution. A non-terminal call is redundant if that same non-terminal has been called at the same character position at least once before. Therefore, with complete memoization – i.e., all non-terminal call results at every character position are stored – there would be no redundant calls. The percentage of redundant calls is thus a good measure for determining how much the parser wastes on re-evaluating past results.

Though the percentage of redundant calls gives a good implementation-agnostic insight into wasted parse time, the number of clock cycles spent by non-terminal calls can vary wildly. Therefore, a more accurate metric is the exact number of clock cycles that were wasted by this specific parsing machine implementation. For this reason, the total number clock cycles spent on parsing is also measured and provides an implementation-specific measure of parsing speedup.

The measurements for memoization effectiveness are performed for increasing cache associativity. The baseline implementation is represented by a cache associativity of zero. The measurements for each file type also include the results for a cache with maximum set associativity. Assuming an input string length of n characters, an n-way set-associative cache is thus used when a maximum cache associativity is mentioned.

Finally, as indicated in Section 4.1.3, caching failed non-terminal evaluation results due to caching might prove to be difficult to implement in hardware. For this reason, a memoization unit with backtrack caching and without backtrack caching was implemented in the virtual parsing machine and both are explored in the measurements.

5.3.2 Benchmark Results

Table 5.7 shows the memoization results for increasing cache associativity values when parsing XML benchmark files from the USGS repository [1]. Here, a cache associativity of zero means that no memoization was used, whereas *max* indicates that an n-way set-associative cache with m sets was used, where n is the input string length and m is the number of non-terminals used by the XML PEG grammar, namely $m = 12$ (refer to Section B.2). Furthermore, the number of clock cycles reported in the table represents the total number of clock cycles when parsing all 60 XML benchmark files. Regarding memoization without caching backtrack results, it can be observed that the redundant calls and number of clock cycles do not decrease at all for any cache associativity configuration. Apparently, the redundant calls in the XML grammar are only caused by re-evaluating previously failed non-terminals. Memoizing these failed non-terminals with only a direct-mapped cache (1-way set associative) already results in the removal of all redundant non-terminal calls, which simply requires a small cache with only 12 entries, one for each non-terminal. However, the total performance gain is barely measurable in both percentage of redundant calls (a 0.1% drop) and number of clocks cycles (a drop of 300 cycles).

Table 5.7: Percentage of redundant calls for various memoization configurations when parsing XML benchmark files [1].

Cache Associativity	**No Backtrack Caching**		**Backtrack Caching**	
	Redundant Calls [%]	*Clock Cycles ($\times 10^6$)*	*Redundant Calls [%]*	*Clock Cycles ($\times 10^6$)*
0	0.1	31.7	0.1	31.7
1	0.1	31.7	0	31.7
max	0.1	31.7	0	31.7

Table 5.8 presents the memoization results for the JSON benchmark files from the USGS repository [1]. In contrast to the memoization results for XML, caching failed non-terminal evaluations is not needed, as a direct-mapped cache without backtrack caching already removes all redundant non-terminal calls. Moreover, a more significant speedup is gained by memoizing successful non-terminal evaluations as the original 6.1% redundant non-terminal calls are removed, resulting in a total decrease of 1 million clock cycles. In total, a direct-mapped cache with only 17 sets is enough for a 2.9% decrease in clock cycle executions when parsing JSON files with the JSON PEG found in Section B.3.

Table 5.8: Percentage of redundant calls for various memoization configurations when parsing JSON benchmark files [1].

Cache Associativity	**No Backtrack Caching**		**Backtrack Caching**	
	Redundant Calls [%]	*Clock Cycles ($\times 10^6$)*	*Redundant Calls [%]*	*Clock Cycles ($\times 10^6$)*
0	6.1	34.5	6.1	34.5
1	0	33.5	0	33.5
max	0	33.5	0	33.5

Lastly, Table 5.9 shows the memoization results for the Java benchmark files from the JDK [2]. Both caching and not caching failed non-terminal evaluations result in reductions in redundant calls and clock cycles. However, if only successful evaluations are cached, the number of redundant calls is at best limited to 11.5%, which is only a 2.3% reduction. For a direct-mapped cache, this is only a 2.0% reduction in redundant calls and 3.9% reduction in clock cycles. On the other hand, if failed evaluations are also cached, the percentage of redundant calls drops to 1.4% for even a direct-mapped cache, giving a 12.5% reduction. Similarly, the number of clock cycles drops by 8.6%. Note that a higher cache associativity does decrease the number of redundant calls and clock cycles, but only by an insignificant amount. In conclusion, the Java PEG found in [47] a direct-mapped cache is optimal considering the trade-off between cache size and performance gain, preferably with caching of failed evaluations enabled. Do consider, however, that 225 sets are needed in order implement a direct-mapped cache for this Java PEG: one for each non-terminal definition.

Only the CSV file type remains to be discussed with regard to memoization. However, it was found that no redundant non-terminal calls occur when parsing CSV benchmark files from the USGS repository [1] according to the CSV PEG defined in Section B.4, even without memoization enabled.

5.3.3 Discussion

From the results presented in this section, it can be concluded that an increase in parsing performance by means of a memoization unit highly depends on the grammar. Analysis of the results for the four grammars used in this section indicates that a larger and

Table 5.9: Percentage of redundant calls for various memoization configurations when parsing Java benchmark files [2].

Cache Associativity	**No Backtrack Caching**		**Backtrack Caching**	
	Redundant Calls [%]	*Clock Cycles ($\times 10^6$)*	*Redundant Calls [%]*	*Clock Cycles ($\times 10^6$)*
0	13.8	30.4	13.8	30.4
1	11.8	29.2	1.4	27.8
2	11.8	29.2	1.4	27.8
4	11.8	29.2	1.3	27.8
max	11.5	28.8	0	27.0

more complex grammar benefits from this optimization more than smaller grammars do. This makes sense intuitively, as the conditions for redundant calls (see Section 4.1.3) are probably more likely to appear in large complex grammars. Moreover, if a memoization unit is to be used by the parsing machine, increasing its set associativity leads to sharp diminishing returns in terms of performance gain. Therefore, a simple direct-mapped cache is often enough for sufficient speedup.

In regard to caching failed non-terminal evaluation results, its benefit is clearly visible when parsing Java benchmark files, while parsing JSON benchmark files did not perform any different with or without this feature. The use of backtrack caching therefore also highly depends on the grammar at hand.

The use of a memoization unit depends entirely on the use case, as does the use of the backtrack caching feature. To this end, the virtual parsing machine can be used to determine their efficacy for specific use cases.

5.4 Conclusion

To determine the operational behavior and performance of the PPEG parsing machine, a test setup was constructed which consists of an implementation of the PPEG architecture in software (virtual parsing machine) and hardware (FPGA). Benchmarks were created consisting of PEGs for CSV, XML, JSON, and Java files, thereby creating a diverse set of PPEG programs with varying size and complexity. These four grammars cover 6 different data sets, of which 2 were synthetically generated, 3 were obtained from real-world data sets, and 1 was selected from an open-source code repository (see Section 5.1).

By running these benchmarks, it was found that the maximum return and backtrack sizes are mostly independent of input file size, but dependent on the size and complexity of the grammar at hand (see Section 5.2.2). Another observation was that a significant number of clock cycles is spent on backtrack operations and non-terminal calls, ranging anywhere from a total of 18% up to 40% percent for these benchmarks. The consequent large number of jumps in instruction and data memory must be kept into account if a more complex memory hierarchy is ever considered (see Section 5.2.3). By comparing the binary code size of PPEG-compiled grammars with other existing implementations, it was observed that the PPEG binaries are generally smaller than those of other implementations if the size of their character set table is taken into account (see Section 5.2.4). Finally, though an attempt was made to compare the performance of the PPEG implementation in hardware with other existing implementations, this proved difficult because their sole implementation in software is dependent on the hardware on which they run. However, it appears that because the PPEG architecture is optimized to run on hardware, it reaches comparable if not better parsing throughput when running the benchmarks than the other reference parsing machine implementations running on much faster hardware (see Section 5.2.5).

The benchmarks were additionally run on the virtual parsing machine with various memoization cache configurations. From the results it can be concluded that large complex grammars benefit more from memoization than smaller grammars, because the conditions for redundant calls appear more for increasing grammar sizes. Moreover, due to diminishing speedup for increasing memoization cache associativity values, a direct-mapped cache is often enough for sufficient speedup (see Section 5.3).

6 Conclusion

6.1 Summary

Chapter 2 explained the information required to get a good understanding of the fundamentals on which the rest of this report builds forth. It walked through key elements of formal grammars, which are used to define the syntax of a language with a varying level of constraints and expressiveness (see Section 2.1). Context-free grammars especially strike a good balance between the imposed constraints and its expressiveness, thereby being often used by parser generators (see Section 2.2).
Unfortunately, the expressiveness of a context-free grammar still poses a problem when two or more correct parses exist for any sentence in its associated language. For this reason, analytic grammars might prove a better type of grammar, as it cannot define ambiguous grammars (see Section 2.3). The most prominent analytic grammar is PEG (parsing expression grammar), which additionally has the ability to define lexical syntax and, unlike context-free grammars, is based on a limited backtracking top-down parsing algorithm (see Section 2.4).
Though much research has been carried out for parsing techniques that are implemented in software, few have been studied for implementation in hardware, namely: pattern matching engines, tabular parsers, and virtual machines. However, the first does not perform exact syntax validation and the second requires memory sizes proportional to the input string length. Only virtual machines, which model a conventional computer architecture, has the potential to satisfy the required goals (see Section 2.5).
By studying existing grammars, parsing techniques, and hardware-oriented implementations, a combination of grammar and parsing technique was selected based on a best fit with regard to the goals listed in Section 1.1. This lead to the decision to base the recognizer design on parsing machines that implement a limited backtracking top-down parsing approach based on parsing expression grammars (see Section 2.6).

In order to design a suitable parsing machine that operates according to a PEG, the most basic components and concepts required for its implementation were explored, such as how limited backtracking and non-terminal calls are handled (see Section 3.1). These concepts were employed for an inductive formalization of the top-down parsing algorithm inherent to PEG (see Section 3.2).
The design of the parsing machine itself begun with a description of its micro-architecture, which resulted from translating the formalized PEG expressions into microcode describing the flow of data between its fundamental components (see Section 3.3). The microcode was then split and grouped in such a way, that a minimal instruction set could be defined. The microcode definition of these instructions then translated

to a mathematical formalization, which defines a parsing machine state transition for each PPEG instruction (see Section 3.4).
Based on the PPEG instruction set and the original microcode, the translation from PEG expression to PPEG assembly code is formalized. Finally, the PEG-PPEG translation formalization combined with the PPEG instruction set formalization were used to verify the parsing machine's operation with respect to the earlier defined inductive formalization of PEG expressions (see Section 3.5).

Chapter 4 explained three main software tools developed for use with the PPEG architecture. The first is a VPM or virtual parsing machine. That is, a software emulator of the PPEG parsing machine architecture was built based on the microcode definitions of the PPEG ISA. The emulator is extended with tools to analyze internal processes and ongoing events. Moreover, parse tree visualization and parse error reporting are included in the VPM. Finally, a memoization extension was added in order to analyze its effect on parses while accounting for the required hardware complexity (see Section 4.1).
The other two tools that were developed are the PEG compiler and PPEG assembler (see Section 4.2). The first takes an arbitrary PEG and compiles it into an equivalent PPEG assembly program. This is achieved by a number of transformations by means of a PEG syntax analyzer, PEG semantic analyzer, and finally PPEG code generator which translates an annotated AST into a PPEG assembly program according to the PEG translation model discussed in Section 3.5. Finally, the PPEG assembly program is transformed to a binary file by means of the PPEG assembler tool, which is based on the PPEG ISA specification discussed in Section 3.4.

To determine the operational behavior and performance of the PPEG parsing machine, a test setup was constructed which consists of an implementation of the PPEG architecture in software (virtual parsing machine) and hardware (FPGA). Benchmarks were created consisting of PEGs for CSV, XML, JSON, and Java files, thereby creating a diverse set of PPEG programs with varying size and complexity. These four grammars cover 6 different data sets, of which 2 were synthetically generated, 3 were obtained from real-world data sets, and 1 was selected from an open-source code repository (see Section 5.1).
By running these benchmarks, it was found that the maximum return and backtrack sizes are mostly independent of input file size, but dependent on the size and complexity of the grammar at hand (see Section 5.2.2). Another observation was that a significant number of clock cycles is spent on backtrack operations and non-terminal calls, ranging anywhere from a total of 18% up to 40% percent for these benchmarks. The consequent large number of jumps in instruction and data memory must be kept into account if a more complex memory hierarchy is ever considered (see Section 5.2.3). By comparing the binary code size of PPEG-compiled grammars with other existing implementations, it was observed that the PPEG binaries are generally smaller than those of other implementations if the size of their character set table is taken into account (see Section 5.2.4). Finally, though an attempt was made to compare the performance of the PPEG implementation in hardware with other existing implementations, this proved difficult because their sole implementation in software is dependent on the hardware on which they run. However, it appears that because the PPEG architecture is optimized to run

on hardware, it reaches comparable if not better parsing throughput when running the benchmarks than the other reference parsing machine implementations running on much faster hardware (see Section 5.2.5).
The benchmarks were additionally run on the virtual parsing machine with various memoization cache configurations. From the results it can be concluded that large complex grammars benefit more from memoization than smaller grammars, because the conditions for redundant calls appear more for increasing grammar sizes. Moreover, due to diminishing speedup for increasing memoization cache associativity values, a direct-mapped cache is often enough for sufficient speedup (see Section 5.3).

6.2 Main Contributions

The aim of this section is to answer the research questions and elaborate on the goals that were introduced in Section 1.1, after which the main contributions are listed.

The main research question was as follows:

- How can a flexible text-based recognizer be built with digital electronics?

The answer to this question is found in a combination of Chapter 2 and Chapter 3. Text-based recognition is usually achieved by means of a parser. Next, a parsing machine approach was chosen, as that allows the creation of a fixed machine that is implemented in hardware, while still allowing to flexibly load different grammars to parse any file type. In order to avoid ambiguity and integrate the lexical syntax analysis part into one parsing machine, the machine is based on parsing expression grammar. Based on a general stack approach for non-terminal calls and backtrack operations, a micro-architecture was constructed, which in turn led to the design of the final instruction set architecture. The simplicity of the architecture allows for possible future extensions in order to enhance recognition behavior and performance.

- What are the design considerations for choosing a hardware-oriented recognition technique?

Section 2.6 details the various design considerations for choosing a hardware-oriented parsing machine approach. Other hardware-based solutions do exist, but are either not strict enough in their recognition approach (e.g., pattern matching engines), or have substantial memory requirements (e.g. tabular parsing approaches). Therefore, both strictness of the recognition approach and memory requirements are important considerations for hardware. Moreover, as the aim is to design a flexible recognizer, the process of loading new file templates or grammars into the recognizer must be as easy as possible, which is why the parsing machine uses a simple instruction and data memory design.

- How can performance be enhanced by extending an existing base design?

The main enhancement that was investigated during this project was the addition of a memoization unit, as discussed in Section 4.1.3. In short, intermediate parse results are stored in a cache in order to minimize the number of non-terminal re-evaluations after backtrack operations. In Section 5.3 it was found that the performance enhancement that such a memoization unit provides depends on the complexity of the grammar at hand. For the benchmarks used in this project, the decrease in parse time reached up to 8.6%

- How does an implementation in hardware compare to existing software implementations?

In Section 5.2, both the parsing throughput and binary code size were measured for the benchmarks used in this project and were compared to the results reported by the MiniNez and GPEG virtual machine implementations. It was observed that the binary code size of PPEG is generally smaller than that of the other implementations, assuming that the size of the character set table employed by those implementations are taken into account. The parsing throughput reaches comparable results to the reference virtual machine implementations running on faster hardware.

The main contributions of this thesis are as follows:

- Designed and implemented a hardware-accelerated PEG parsing machine.
- Developed a mathematical formalism for a PEG-based recognizer.
- Developed a mathematical formalism for the behavior of the PPEG architecture.
- Developed a mathematical formalism for the translation from PEG to PPEG code.
- Developed a proof that the PPEG architecture implements all fundamental PEG expressions.
- Created compiler and assembler that translates PEG to PPEG assembly and binary machine code respectively.
- Compared the performance of the PPEG parsing machine with existing similar implementations.
- Investigated the performance improvement when adding memoization to the parsing machine.

6.3 Future Work

The developments on the PPEG parsing machine architecture presented in this work have proven that hardware-accelerated recognizers are indeed possible, achieve good performance compared to existing software-based solutions, and can be optimized in performance by integrating caching techniques. There are, however, a multitude of additional developments that can improve performance or add more interesting recognizer behavior. The most interesting future work is summarized hereafter.

Error Detection and Handling
The current PPEG architecture design does not include any fail-safe mechanism for

handling undefined behavior. Such behavior might be cause by, but is not limited to, the following erroneous events:

- Return stack or backtrack stack push operation when these are completely filled.
- Return stack or backtrack stack pop operation when these are completely empty.
- Overflow of character position.
- Overflow of program counter.
- Character position out of bounds.
- Program counter out of bounds.

In the future, these, and other events, must be caught and handled gracefully instead of producing undefined parsing behavior. In the case of the above events, this entails stopping the current parse and setting the parsing machine status register to indicate the specific error that caused the interrupt.

Character Set Table and Character String Table
Similar to the MiniNez and GPEG virtual machines, the PPEG architecture may be extended to use a character set table to store complex character classes. When such a character class is then encountered during the parse, the table can be used to perform a character class comparison with a single instruction. Additionally, a character string table might be constructed to store the relatively long constant string literals specified by the grammar. When encountering this string during the parse, the table can be used to perform multiple character comparisons at a time by means of multiple 8-bit comparators in parallel.

It is recommended to first implement these changes in the virtual parsing machine in order to analyze the benefit of these additions for the intended use case, similar as was done for the memoization unit in Section 5.3.

Memory Hierarchy
As was discussed in hardware test setup presented in Section 5.1.1, the current FPGA-based implementation of the PPEG parsing machine only utilizes as much available on-chip memory, which is of course limited. In order to parse larger input files with larger grammars, this flat memory structure needs to be replaced by a better memory hierarchy, similar to those implemented in conventional computer architectures. For example, external work memory can be added to store these larger files and binaries. However, this can affect memory read and write times significantly. To this end, separate or combined instruction and data cache can be considered for speeding up these memory operations. The efficacy of these caches rely heavily on the jump behavior of the parsing machines as spatial and temporal locality of instructions and data are affected by this.

It is again recommended to implement and analyze the exact effect of any new memory hierarchy on the parsing behavior by means of the virtual parsing machine. This allows for rapid prototyping, reducing the long development time on the target hardware platform.

Semantic Analysis

One of the most interesting topics for future work on the PPEG architecture is the inclusion of semantic analysis. Because the PPEG parsing machine is based on PEG, lexical syntax and general syntax analysis are already included in the machine. However, because the goal is to build a flexible text-based recognizer, which can include the ability to recognize and validate specific data types by means of semantic analysis.

Although attempted in Appendix B, in general PEG cannot exactly specify the correct syntax for XML files. The following snippet presents the PEG expression for a simple XML element:

```
Element <- '<' Name '>' Content '</' Name '>')
```

XML requires the start tag and closing tag to have the same name. However, the above PEG does not and cannot set this constraint. If support for string type comparisons was added to the PPEG architecture, this PEG specification can be extended to set this necessary constraint as follows:

```
Element <- '<' {Name} '>' Content '</' {Name} '>')
    {{  assert x0.string() == x1.string()  }}
```

Here, the two `Name` non-terminals are surrounded by curly brackets to indicate that they are the subject of a semantic check. The part between double curly brackets contains these semantic checks, in this case a simple assert. `x0` refers to the first element in single curly brackets, `x1` to the second element in curly brackets, etc. The suffix `.string()` indicates that the parsed elements between brackets are operated on as strings. In this case, the `assert` statement means to verify that the opening and closing tags are the same string, which results in a backtrack operation if this is not the case.

Another example that shows the benefit of adding a semantic analysis component to the parsing machine is as follows. Assume that a 16-bit unsigned integer needs to be parsed, or in other words a number ranging anywhere from 0 to 65,535. Though this appears simple, an implementation in PEG is not trivial, as can be seen from the following 16-bit integer definition:

```
UINT16 <-                           [0-9] /
                              [1-9] [0-9] /
                        [1-9] [0-9] [0-9] /
                  [1-9] [0-9] [0-9] [0-9] /
            [1-5] [0-9] [0-9] [0-9] [0-9] /
            6     [0-4] [0-9] [0-9] [0-9] /
            6     5     [0-4] [0-9] [0-9] /
            6     5     5     [0-2] [0-9] /
            6     5     5     3     [0-5]
```

As can be seen, parsing a 16-bit integer can result in a significant number of backtrack operations. If the PPEG parsing machine supported integer conversion and semantic checks, the previous PEG can be reduced to the following simple grammar:

```
UINT16  <- {[0-9]+}
        {{  assert x0.uint() <= 65535  }}
```

Similar to the `.string()` suffix, the suffix `.uint()` indicates that the parsed element between curly brackets is operated on as an unsigned integer. During the syntactic parse of the `UINT16` non-terminal, the parsed characters need to be converted to integers and accumulated to compute the integer value that has been parsed. Thereafter, a simple integer comparison can be performed to check if the parsed value is equal to or lower than 65,535.

In conclusion, adding a semantic analysis component can extend the basic abilities of the PEG recognizer and possibly speed up the parsing process at the same time. The exploration of such a component is therefore an interesting next step in this project.

Bibliography

[1] USGS, “USGS Earthquake Catalog - API Documentation.” [Online]. Available: https://earthquake.usgs.gov

[2] Oracle, “JDK 5.0 Source Code & Licensing Overview.” [Online]. Available: https://www.oracle.com/java/technologies/javase/source-license.html

[3] B. Ford, “Parsing expression grammars,” *ACM SIGPLAN Notices*, vol. 39, no. 1, 1 2004. [Online]. Available: https://doi.org/10.1145/982962.964011

[4] S. Honda and K. Kuramitsu, “Implementing a Small Parsing Virtual Machine on Embedded Systems,” *CoRR*, vol. abs/1511.03406, 11 2015. [Online]. Available: https://arxiv.org/abs/1511.03406

[5] Z. Yedidia, *Incremental PEG Parsing (BSc thesis).* Cambridge: Harvard College, 3 2021. [Online]. Available: https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37368580

[6] G. Jäger and J. Rogers, “Formal language theory: refining the Chomsky hierarchy,” *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 367, no. 1598, 7 2012. [Online]. Available: https://doi.org/10.1098/rstb.2012.0077

[7] N. Chomsky, “Three models for the description of language,” *IEEE Transactions on Information Theory*, vol. 2, no. 3, 9 1956. [Online]. Available: https://doi.org/10.1109/TIT.1956.1056813

[8] D. Grune and C. J. Jacobs, *Parsing Techniques: A Practical Guide*, 2nd ed., D. Gries and F. B. Scheider, Eds. Amsterdam: Springer, 2008. [Online]. Available: https://doi.org/10.1007/978-0-387-68954-8

[9] J. E. Hopcroft and J. D. Ullman, *Formal Languages and Their Relation to Automata*, 1st ed., M. A. Harrison and R. S. Varga, Eds. Addison-Wesley Publishing Company, 1969. [Online]. Available: https://dl.acm.org/doi/10.5555/1096945

[10] B. Ford, *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking (MSc thesis).* Cambridge: Massachusetts Institute of Technology, 9 2002. [Online]. Available: https://doi.org/1721.1/87310

[11] J. L. Hein, “Regular Language Topics,” in *Discrete Structures, Logic, and Computability*, 4th ed. Jones & Bartlett Learning, 2015, ch. 11.4, pp. 54–59. [Online]. Available: https://dl.acm.org/doi/book/10.5555/515454

[12] A. V. Aho and J. D. Ullman, *The theory of parsing, translation, and compiling*, 1st ed., G. Forsythe, Ed. Englewood Cliffs: Prentice-Hall, Inc., 1 1972, vol. 1. [Online]. Available: https://dl.acm.org/doi/book/10.5555/578789

[13] D. E. Knuth, “Top-down syntax analysis,” *Acta Informatica*, vol. 1, no. 2, pp. 79–110, 1971. [Online]. Available: https://doi.org/10.1007/BF00289517

[14] A. Afroozeh and A. Izmaylova, “Practical General Top-Down Parsers,” Ph.D. dissertation, University of Amsterdam, Amsterdam, 9 2019. [Online]. Available: https://hdl.handle.net/11245.1/7f1bf220-8a2d-45ba-8402-a66f6c18a860

[15] D. Michie, ““Memo” Functions and Machine Learning,” *Nature*, vol. 218, no. 5136, pp. 19–22, 4 1968. [Online]. Available: https://doi.org/10.1038/218019A0

[16] D. H. Younger, “Recognition and parsing of context-free languages in time n3,” *Information and Control*, vol. 10, no. 2, pp. 189–208, 2 1967. [Online]. Available: https://doi.org/10.1016/S0019-9958(67)80007-X

[17] J. Earley, “An efficient context-free parsing algorithm,” *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, 2 1970. [Online]. Available: https://doi.org/10.1145/362007.362035

[18] P. Gilbert, “On the Syntax of Algorithmic Languages,” *Journal of the ACM*, vol. 13, no. 1, pp. 90–107, 1 1966. [Online]. Available: https://doi.org/10.1145/321312.321319

[19] S. C. Johnson, “Yacc: Yet Another Compiler-Compiler,” AT&T Bell Laboratories, Murray Hill, Tech. Rep., 1978.

[20] T. Parr, S. Harwell, and K. Fisher, “Adaptive LL(*) parsing,” *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 579–598, 12 2014. [Online]. Available: https://doi.org/10.1145/2714064.2660202

[21] L. C. Kats, E. Visser, and G. Wachsmuth, “Pure and declarative syntax definition,” in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA ’10.* New York, New York, USA: ACM Press, 2010, p. 918. [Online]. Available: https://doi.org/10.1145/1869459.1869535

[22] A. Birman and J. D. Ullman, “Parsing algorithms with backtrack,” in *11th Annual Symposium on Switching and Automata Theory (swat 1970).* IEEE, 10 1970, pp. 153–174. [Online]. Available: https://doi.org/10.1109/SWAT.1970.18

[23] R. M. McClure, “Programming languages for non-numeric processing—1,” in *Proceedings of the 1965 20th national conference on -.* New York, New York, USA: ACM Press, 1965, pp. 262–274. [Online]. Available: https://doi.org/10.1145/800197.806050

[24] D. V. Schorre, “META II a syntax-oriented compiler writing language,” in *Proceedings of the 1964 19th ACM national conference on -.* New York, New York, USA: ACM Press, 1964, pp. 301–41. [Online]. Available: https://doi.org/10.1145/800257.808896

[25] ISO, “Syntactic Metalanguage - Extended BNF - ISO/IEC 14977,” International Standards Organization, Tech. Rep., 1996. [Online]. Available: https://www.iso.org/standard/26153.html

[26] R. R. Redziejowski, "Some Aspects of Parsing Expression Grammar," *Fundamenta Informaticae*, vol. 85, pp. 441–451, 1 2008. [Online]. Available: https://dl.acm.org/doi/10.5555/2365896.2365924

[27] ——, "Applying Classical Concepts to Parsing Expression Grammar," *Fundamenta Informaticae*, vol. 93, pp. 325–336, 2009. [Online]. Available: https://dl.acm.org/doi/10.5555/1576070.1576093

[28] ——, "Parsing Expression Grammar as a Primitive Recursive-Descent Parser with Backtracking," *Fundamenta Informaticae*, vol. 79, pp. 513–524, 9 2007. [Online]. Available: http://www.romanredz.se/papers/FI2007.pdf

[29] Cho Y. H., J. Moscola, and J. Lockwood, "Context-Free-Grammar based Token Tagger in Reconfigurable Devices," in *22nd International Conference on Data Engineering Workshops (ICDEW'06).* IEEE, 2006, pp. 78–78. [Online]. Available: https://doi.org/10.1109/ICDEW.2006.42

[30] J. Moscola, J. W. Lockwood, and Y. H. Cho, "Reconfigurable content-based router using hardware-accelerated language parser," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 2, pp. 1–25, 4 2008. [Online]. Available: https://doi.org/10.1145/1344418.1344424

[31] J. V. Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson, "XML Accelerator Engine," in *International Workshop on High Performance XML Processing*, New York, 5 2004.

[32] C. Ciressan, E. Sanchez, M. Rajman, and J.-C. Chappelier, "An FPGA-based coprocessor for the parsing of context-free grammars," in *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00871).* IEEE Comput. Soc, 4 2000, pp. 236–245. [Online]. Available: https://doi.org/10.1109/FPGA.2000.903411

[33] ——, "An FPGA-Based Syntactic Parser for Real-Life Almost Unrestricted Context-Free Grammars," *Lecture Notes in Computer Science*, vol. 2147, pp. 590–594, 2001. [Online]. Available: https://doi.org/10.1007/3-540-44687-7_61

[34] H. Cheng and K. Fu, "Algorithm partition and parallel recognition of general context-free languages using fixed-size VLSI architecture," *Pattern Recognition*, vol. 19, no. 5, pp. 361–372, 1 1986. [Online]. Available: https://doi.org/10.1016/0031-3203(86)90003-8

[35] Y. T. Chiang and K. S. Fu, "Parallel Parsing Algorithms and VLSI Implementations for Syntactic Pattern Recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, no. 3, pp. 302–314, 5 1984. [Online]. Available: https://doi.org/10.1109/TPAMI.1984.4767522

[36] A. Šaikūnas, "Parsing with Earley Virtual Machines," in *Proceedings of the 2017 FedCSIS*, 9 2017, pp. 165–173. [Online]. Available: http://dx.doi.org/10.15439/2017F162

[37] R. Cox, "Regular Expression Matching: the Virtual Machine Approach," 12 2009. [Online]. Available: https://swtch.com/~rsc/regexp/regexp2.html

[38] S. Medeiros and R. Ierusalimschy, "A parsing machine for PEGs," in *Proceedings of the 2008 symposium on Dynamic languages - DLS '08*. New York, New York, USA: ACM Press, 7 2008. [Online]. Available: https://doi.org/10.1145/1408681.1408683

[39] R. Ierusalimschy, "A Text Pattern-Matching Tool based on Parsing Expression Grammars," *Software: Practice and Experience*, vol. 39, no. 3, pp. 221–258, 3 2009. [Online]. Available: https://doi.org/10.1002/spe.892

[40] J. von Neumann, "First draft of a report on the EDVAC," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993. [Online]. Available: https://doi.org/10.1109/85.238389

[41] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann Publishers, 5 2002. [Online]. Available: https://dl.acm.org/doi/10.5555/1999263

[42] R. Harper, *Practical Foundations for Programming Languages*, 2nd ed. Cambridge: Cambridge University Press, 2016. [Online]. Available: https://doi.org/10.1017/CBO9781316576892

[43] INCITS, "INCITS 4-1986[R2017]," InterNational Committee for Information Technology Standards, Tech. Rep., 2017. [Online]. Available: https://standards.incits.org/apps/group_public/project/details.php?project_id=1829

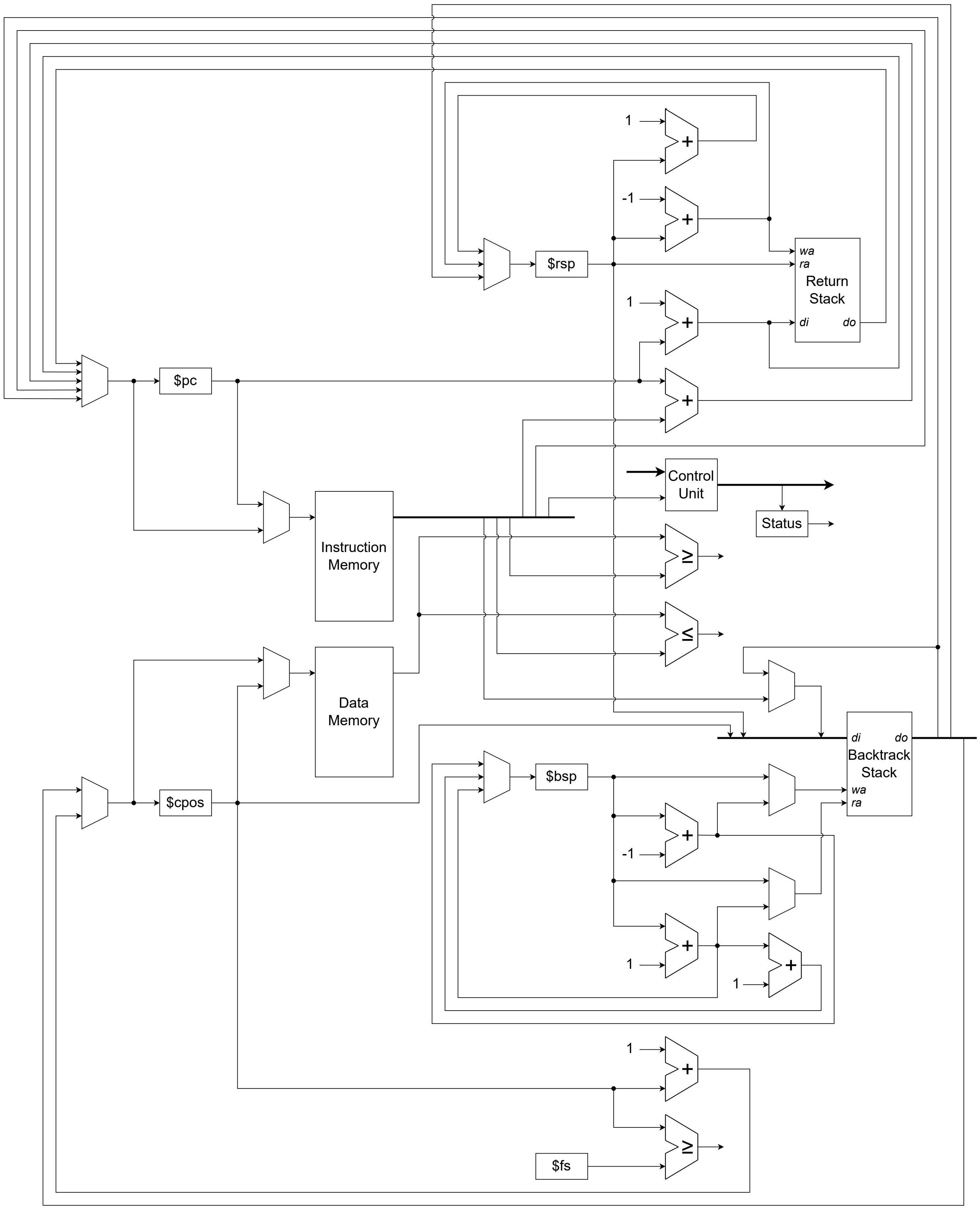
[44] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th ed. Morgan Kaufmann Publishers, 2014. [Online]. Available: https://dl-acm-org.tudelft.idm.oclc.org/doi/book/10.5555/2568134

[45] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed., M. Hirsch, Ed. Boston: Addison Wesley, 8 2006.

[46] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: A Benchmark for XML Data Management," in *Proceedings of the 28th VLDB Conference*, Hong Kong, 2002. [Online]. Available: https://doi.org/10.1016/B978-155860869-6/50096-2

[47] R. Redziejowski, "Parsing Expression Grammar for Java 1.7 for Mouse 1.1 - 1.5," 7 2011. [Online]. Available: https://github.com/zyedidia/gpeg/blob/master/grammars/java.peg

PPEG Architecture A



1
-1
1
$rsp
wa
ra
Return Stack
di
do
$pc
Control Unit
Status
Instruction Memory
Data Memory
$bsp
$cpos
di
do
Backtrack Stack
wa
ra
-1
1
1
1
$fs

B PEG Definitions

B.1 PEG Syntax Definition

```
# Hierarchical syntax
Grammar         <- Spacing Definition+ EndOfFile
Definition      <- Identifier LEFTARROW Expression
Expression      <- Sequence (SLASH Sequence)*
Sequence        <- Prefix*
Prefix          <- (AND / NOT)? Suffix
Suffix          <- Primary (QUESTION / STAR / PLUS)?
Primary         <- Identifier !LEFTARROW
                        / OPEN Expression CLOSE
                        / Literal / Class / DOT

# Lexical syntax
Identifier      <- IdentStart IdentCont* Spacing
IdentStart      <- [a-zA-Z_]
IdentCont       <- IdentStart / [0-9]
Literal         <- "'" (!"'" Char)* "'" Spacing
                        / '"' (!'"' Char)* '"' Spacing
Class           <- '[' (!']' Range)* ']' Spacing
Range           <- Char '-' Char / Char
Char            <- '\\' [nrt'"\-\[\]\\]
                        / '\\' [0-2][0-7][0-7]
                        / '\\' [0-7][0-7]?
                        / !'\\' .
LEFTARROW       <- '<-' Spacing
SLASH           <- '/' Spacing
AND             <- '&' Spacing
NOT             <- '!' Spacing
QUESTION        <- '?' Spacing
STAR            <- '*' Spacing
PLUS            <- '+' Spacing
OPEN            <- '(' Spacing
CLOSE           <- ')' Spacing
DOT             <- '.' Spacing
Spacing         <- (Space / Comment)*
Comment         <- '#' (!EndOfLine .)* EndOfLine
Space           <- ' ' / '\t' / EndOfLine
EndOfLine       <- '\r\n' / '\n' / '\r'
EndOfFile       <- !.
```

Listing B.1: PEG definition of the PEG syntax (obtained from [3]).

B.2 XML Syntax Definition

```
TopLevel        <- PROLOG? _* DTD? _* Element _*
PROLOG          <- '<?xml' (!'?>' .)* '?>'
DTD                     <- '<!' (!'>' .)* '>'
Element         <-  '<' Name (_+ Attribute)* ('/>' / '>' Content '</' Name
    '>') _*
Name            <- [A-Za-z:] ('-' / [A-Za-z0-9:._])*
Attribute       <- Name _* '=' _* String
String          <- '"' (!'"' .)* '"'
Content         <- (Element / CDataSec / CharData)*
CDataSec        <- '<![CDATA[' (!']]>' .)* ']]>' _*
COMMENT         <- '<!--' (!'-->' .)* '-->' _*
CharData        <- (!'<' .)+
_                       <- [ \t\r\n]
```

Listing B.2: PEG definition of the XML syntax (obtained from [4]).

B.3 JSON Syntax Definition

```
doc             <- JSON !.
JSON            <- S_ (Number / Object / Array / String / True / False / Null)
    S_
Object          <- '{' (String ':' JSON (',' String ':' JSON)* / S_) '}'
Array           <- '[' (JSON (',' JSON)* / S_) ']'
StringBody      <- Escape? ((!["\\\00-\37] .)+ Escape*)*
String          <- S_ '"' StringBody '"' S_
Escape          <- '\\' (["{|\\bfnrt] / UnicodeEscape)
UnicodeEscape   <- 'u' [0-9A-Fa-f] [0-9A-Fa-f] [0-9A-Fa-f] [0-9A-Fa-f]
Number          <- Minus? IntPart FractPart? ExpPart?
Minus           <- '-'
IntPart         <- '0' / [1-9] [0-9]*
FractPart       <- '.' [0-9]+
ExpPart         <- [eE] [+\-]? [0-9]+
True            <- 'true'
False           <- 'false'
Null            <- 'null'
S_              <- [\11-\15\40]*
```

Listing B.3: PEG definition of the JSON syntax (obtained from [5]).

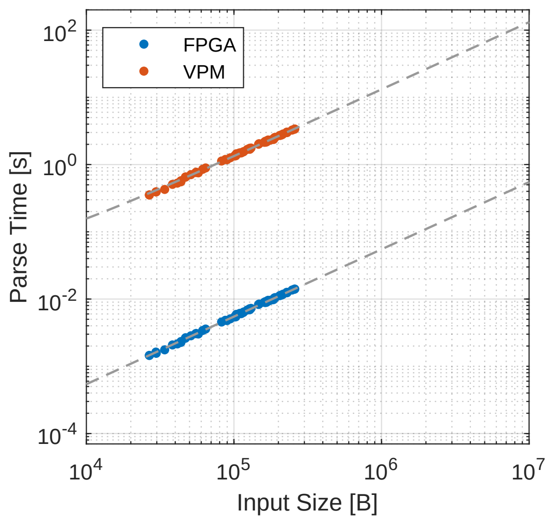
B.4 CSV Syntax Definition

```
File    <- CSV*
CSV     <- Value (',' Value)* '\n'
Value   <- (![,\n] .)*
```

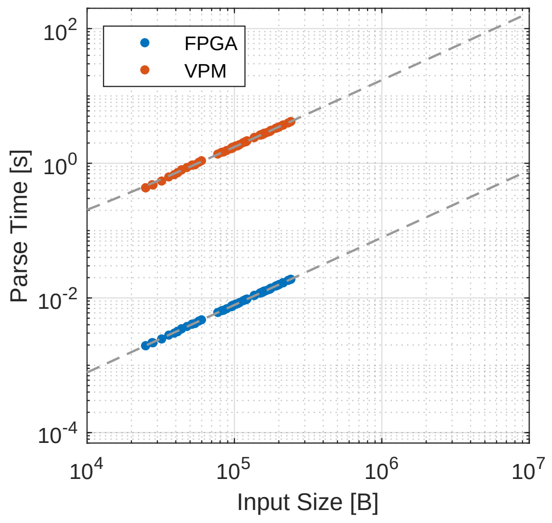
Listing B.4: PEG definition of the CSV syntax (obtained from [4]).

Benchmark Parse Measurements

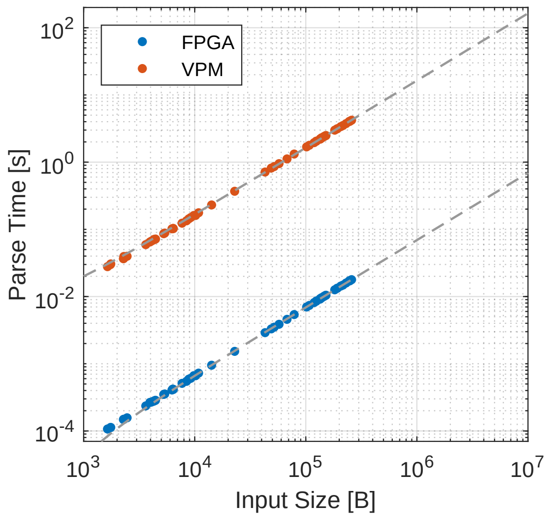
C



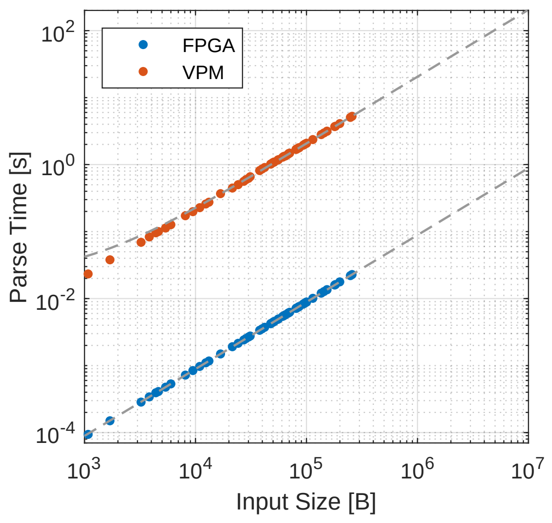
(a) XML files generated with `xmlgen` [46].



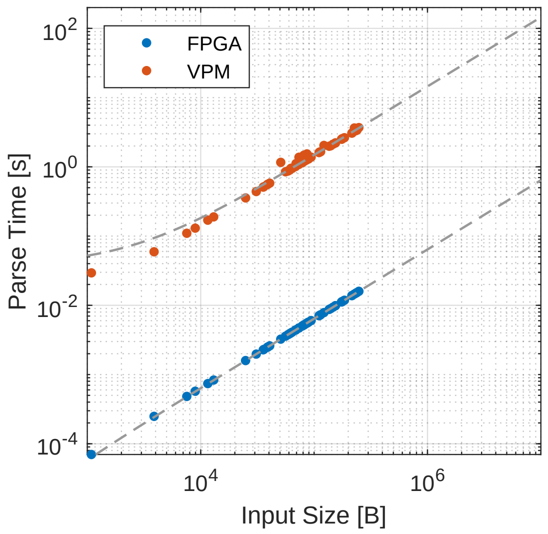
(b) JSON files generated with `xmlgen` [46].



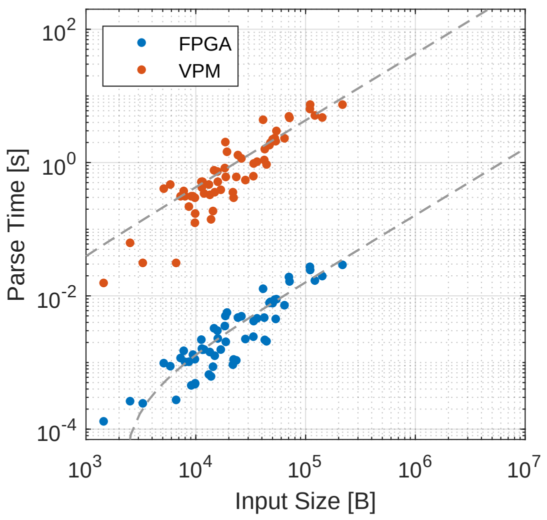
(c) XML files sourced from USGS Earthquake Hazards Program [1].



(d) JSON files sourced from USGS Earthquake Hazards Program [1].

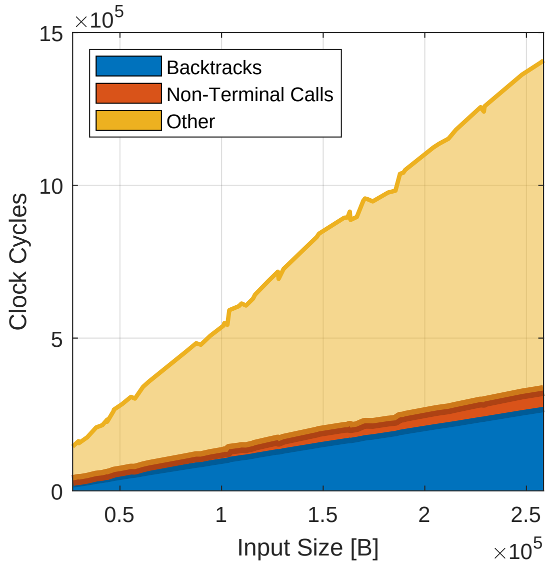


(e) CSV files sourced from USGS Earthquake Hazards Program [1].

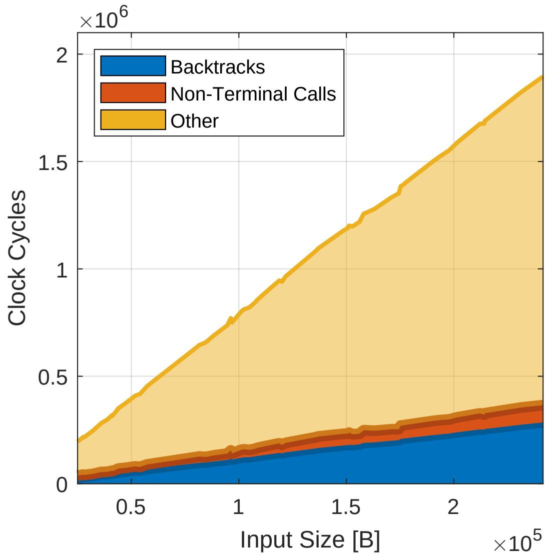


(f) Java files sourced from Oracle JDK 5.0 [2].

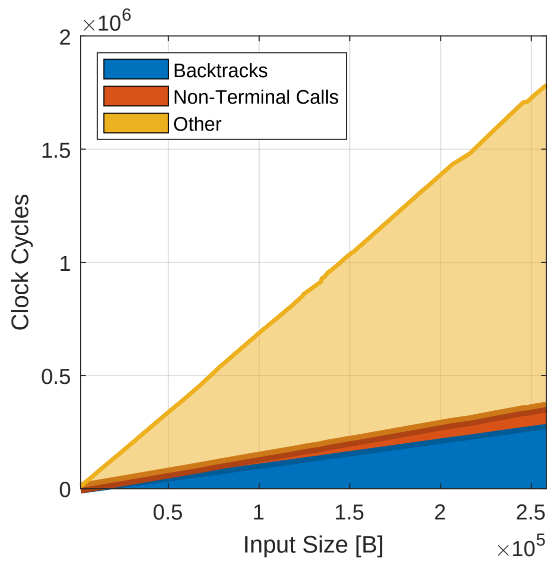
Figure C.1: Parse times by FPGA implementation and VPM for files from five benchmarks. The dashed gray lines are linear projections obtained by applying least-squares regression.



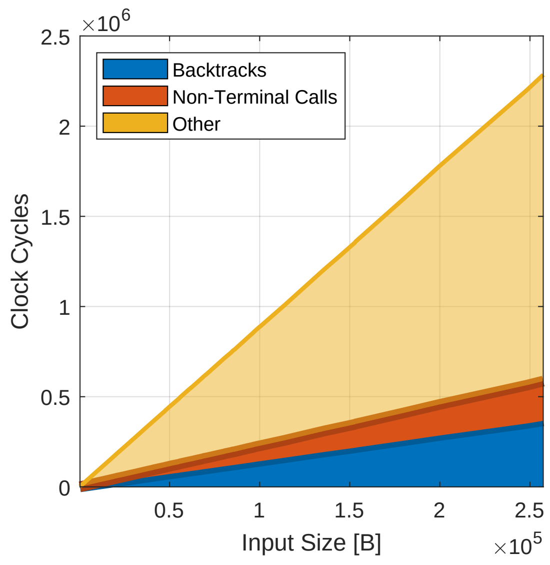
(a) XML files generated with `xmlgen` [46].



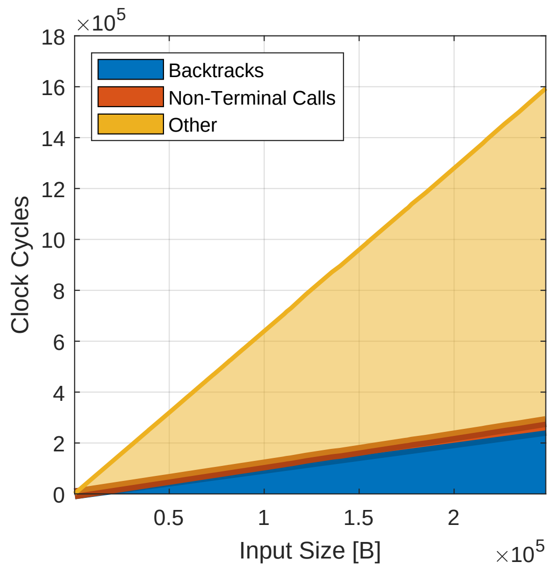
(b) JSON files generated with `xmlgen` [46].



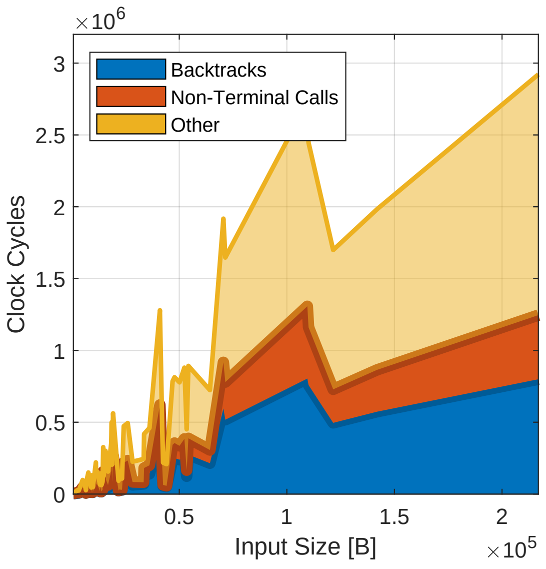
(c) XML files sourced from USGS Earthquake Hazards Program [1].



(d) JSON files sourced from USGS Earthquake Hazards Program [1].

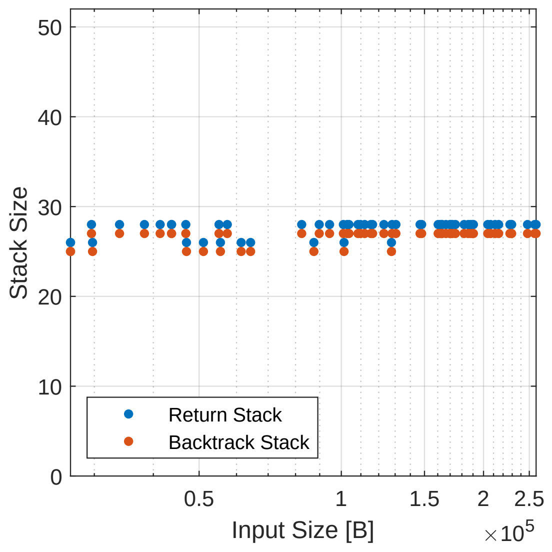


(e) CSV files sourced from USGS Earthquake Hazards Program [1].

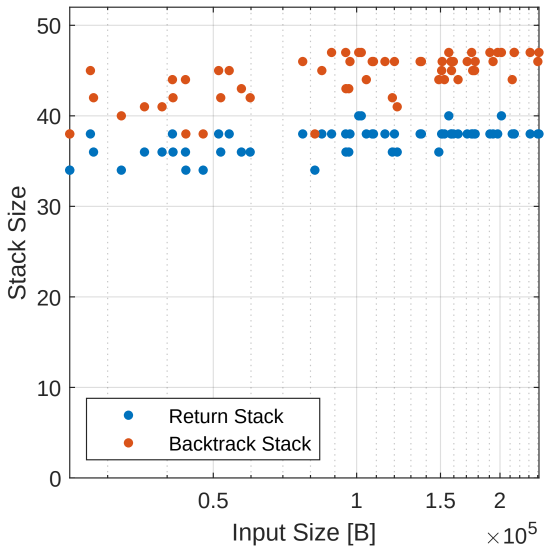


(f) Java files sourced from Oracle JDK 5.0 [2].

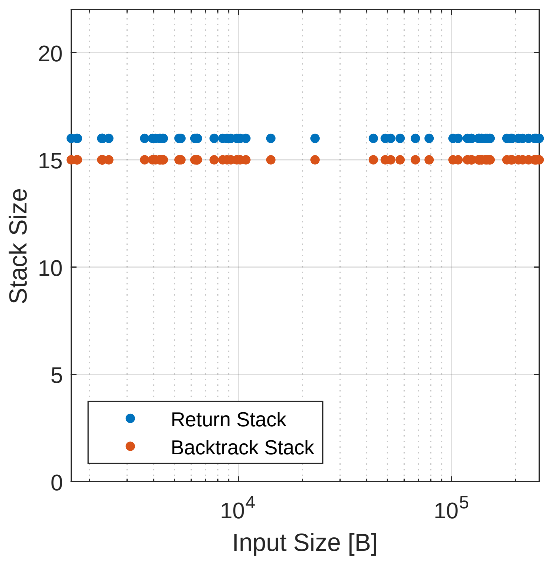
Figure C.2: Number of clock cycles spent on backtracks, non-terminal calls, and other activities when parsing files from five benchmarks.



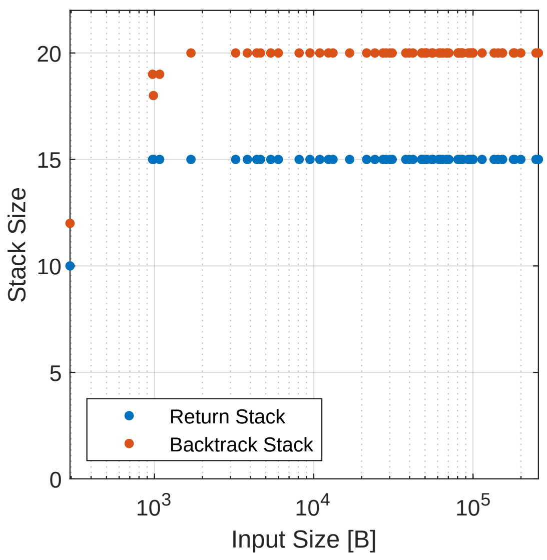
(a) XML files generated with `xmlgen` [46].



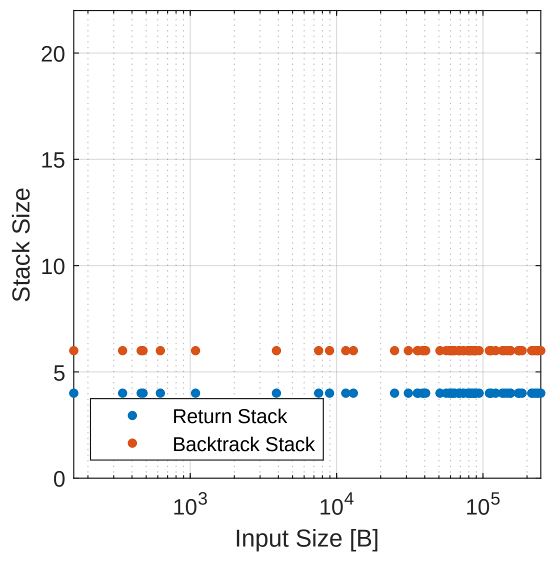
(b) JSON files generated with `xmlgen` [46].



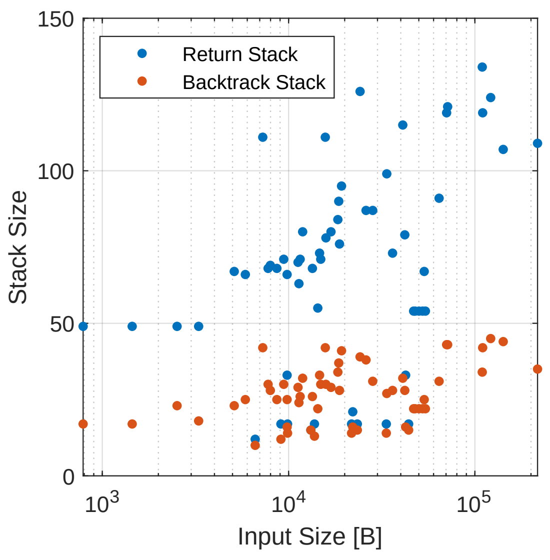
(c) XML files sourced from USGS Earthquake Hazards Program [1].



(d) JSON files sourced from USGS Earthquake Hazards Program [1].



(e) CSV files sourced from USGS Earthquake Hazards Program [1].



(f) Java files sourced from Oracle JDK 5.0 [2].

Figure C.3: Maximum return stack and backtrack stack entries required when parsing files from five benchmarks.