

BACHELOR THESIS
IN
COMPUTER SCIENCE

Radboud University



On Parsing Expression Grammars

A recognition-based system for deterministic languages

Author:

Démian Janssen

`wd.janssen@student.ru.nl`

First supervisor/assessor:

Herman Geuvers

`herman@cs.ru.nl`

Student number:

4182294

Second assessor:

Freek Wiedijk

`freek@cs.ru.nl`

January 11, 2016

Contents

1	Introduction	1
2	Formal languages	2
2.1	Operations on strings	3
2.2	Operations on languages	4
3	Regular expressions	5
3.1	Syntax	5
3.2	Semantics	5
3.3	Properties	6
3.4	Equivalence	7
4	Parsing expression grammars	9
4.1	Syntax	9
4.2	Semantics	10
4.3	Syntactic sugar	14
4.4	Properties	15
4.5	Well-formedness	16
5	Chomsky hierarchy	18
5.1	Transforming regular expressions to parsing expression grammars	18
5.2	Class of languages	20
6	Discussion	22

1 Introduction

Birman & Ullman [1] have shown that the class of deterministic context-free languages can be parsed in linear time complexity on a random access machine. In fact, the time complexity extends to an even larger class of languages. The limitations to computing power and memory in their time, however, did not allow for the algorithm to be put into practice. In formal language theory, we are used to *generative* grammars such as regular expressions and context-free grammars. However, generative grammars are often non-deterministic by nature and certain constructions may cause major headaches to software engineers who have to implement a parsing algorithm which correctly and efficiently parses the language. Extensive parsing libraries have been written to help mitigate these headaches. Ford [4] has used Birman & Ullman's paper [1] to create a *recognition-based* grammar he calls *parsing expression grammars*. The main difference between *generative* and *recognition-based* grammars is that there is not much of a gap between the theory of recognition-based grammars and its practical applications. Parsing expression grammars are similar to context-free grammars in terms of syntax. The biggest differences being that parsing expression grammars are deterministic, and they have some form of context-sensitivity through syntactic predicates, which allows them to even parse (some) context-sensitive languages.

In this thesis, we provide some basic concepts of formal language theory relevant to computer science and relate them to the parsing expression grammar formalism. We introduce the parsing expression grammar formalism by Ford [4] and discuss some of its properties. We introduce an algorithm by Medeiros, Mascarenhas & Ierusalimschy [6] (Medeiros et al.) which transforms regular expressions to parsing expression grammars.

2 Formal languages

In the field of computer science, a language is seen as a set of strings, as opposed to *natural languages* most people are familiar with. Certain restrictions on a language can define the language's class, as defined by Chomsky [2]. We describe formal languages with formal constructs such as *regular expressions*, *context-free grammars* and *set notation*. Throughout this thesis, we will mostly use *set notation* to describe languages as it is arguably the most readable way of describing a formal language. Before we start defining formal languages, we define the atomic building blocks.

Definition 2.1. *Symbol.*

A character without any particular meaning. Unless specified otherwise, arbitrary symbols will be denoted by a, b or a_i, b_i for some $i \in \mathbb{N}$.

Computer scientists like to put elements in sets. Therefore we formally define the alphabet as a set of symbols in definition 2.2.

Definition 2.2. *Alphabet.*

An alphabet is a *non-empty, finite* set of symbols. Arbitrary alphabets will be denoted by Σ or Σ_i for some $i \in \mathbb{N}$.

Example 2.3. *An alphabet.*

Any of the following sets is a valid *alphabet*:

1. $\{a\}$
2. $\{a, b\}$
3. $\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$
4. $\{a, \bar{a}, \theta, z, \delta, e, \ddot{e}, \mathcal{K}, \mathfrak{z}, u, \check{u}, \kappa, \mathcal{L}, \mathcal{M}, \mathcal{H}, o, n, p, c, m, y, \phi, x, \psi, \mathcal{U}, \mathcal{U}, \delta, \mathfrak{b}, \mathfrak{b}, \mathcal{N}, \mathcal{R}\}$
5. $\{n \in \mathbb{N} \mid l \leq n \leq u\}$ a set of numbers between some lower- and upper-bound, l and u respectively.

Even though we just declared symbols to be characters without any particular meaning, there is one exception which we will be using in our definition of strings. The Greek letter λ is used to denote an *empty sequence* of symbols. We formally define strings in definition 2.4.

Definition 2.4. *String.*

A string is a finite sequence of symbols. A string over an alphabet Σ is defined inductively, where $a \in \Sigma$ is an arbitrary symbol:

1. λ is an *empty string*
2. if s is a string, then as is also a string

Arbitrary strings will be denoted with s, s' or s_i for some $i \in \mathbb{N}$. Since λ is the empty string, we say the concatenation $\lambda\lambda = \lambda$.

Example 2.5. *A string.*

Using the alphabets as defined in example 2.3, we can define example strings over these alphabets.

1. λ, a, aa, aaa
2. a, b, ab
3. $a, b, \dots, y, z, \text{string}, \text{hello}$
4. $a, б, \dots, ю, я, \text{строка}, \text{привет}$
5. Suppose $l = 0$ and $u = 1$, then this language consists of binary strings such as $0, 1, 01$ and 10

It is important to note that the empty string λ is a string in any alphabet Σ^* . If the alphabet itself includes λ as a symbol, we would need to use a different symbol to denote the empty string.

Definition 2.6. *Language.*

A language over the alphabet Σ is defined as a set of strings using symbols from that alphabet. Languages are often bound by a set of constraints. Arbitrary languages will be denoted L or L_i for some $i \in \mathbb{N}$. For every language L over an alphabet Σ we have $L \subseteq \Sigma^*$, where Σ^* is as defined in definition 2.15.

2.1 Operations on strings

Using our definition of strings, we can define operations on strings. Interesting properties of a string, such as the length of a string or the number of occurrences of an arbitrary symbol a in a string, can be defined inductively.

Definition 2.7. The length of string s , given by $|s| : \Sigma^* \rightarrow \mathbb{N}$,
Given a string s over an alphabet Σ , we define the length of s inductively, where a is a symbol in Σ :

$$\begin{aligned} |\lambda| &= 0 \\ |as| &= 1 + |s| \end{aligned}$$

Definition 2.8. The number of occurrences of a in s , given by $|s|_a : \Sigma^* \rightarrow \mathbb{N}$.
Given a string s over an alphabet Σ , we define the number of occurrences of a in s inductively, where a and b are symbols in Σ such that $a \neq b$:

$$\begin{aligned} |\lambda|_a &= 0 \\ |bs|_a &= 0 + |s|_a \\ |as|_a &= 1 + |s|_a \end{aligned}$$

Apart from counting symbols in a string, we can also define some more interesting operations, such as the *reverse* operation. We define the reverse of a string in definition 2.9.

Definition 2.9. The reverse of s , given by $s^R : \Sigma^* \rightarrow \Sigma^*$.
Given a string s over an alphabet Σ , we define the reverse of s inductively, where a is a symbol in Σ :

$$\begin{aligned} \lambda^R &= \lambda \\ (as)^R &= s^R a \end{aligned}$$

The operation of reversing a string can be used to define palindromes, as in example 2.10.

Example 2.10. *Palindromes.*

A palindrome is a string s such that s is equal to the reverse of s .

We can define a language L over Σ as $L = \{s \in \Sigma^* \mid s = s^R\}$. L then contains all palindromes which can be made using the symbols in Σ . So if we have $\Sigma = \{a, b\}$ we then get $L = \{\lambda, a, b, aa, bb, aaa, aba, bbb, bab, \dots\}$.

2.2 Operations on languages

Since we defined languages as sets of strings, there are a couple of interesting operations which we will define explicitly:

Definition 2.11. *Language concatenation.*

A language L can be formed by concatenating two languages L_1L_2 such that each string $s \in L$ is the concatenation of a string in L_1 and a string in L_2 . To put it more formally: $L_1L_2 = \{s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2\}$.

Example 2.12. Suppose $L_1 = \{aa, bb\}$ and $L_2 = \{ab, ba\}$.

Then $L_1L_2 = \{aaab, aaba, bbab, bbba\}$.

Definition 2.13. Let $L^0 = \{\lambda\}$, we define L^i to be the concatenation $L_0L_1 \cdots L_{i-1}L_i$ for any $i \in \mathbb{N}$ where each L_j is equal to L .

Example 2.14. Suppose $L = \{aa, bb, ab\}$.

Then $L^2 = LL = \{aaaa, aabb, aaab, bbaa, bbbb, bbab, abaa, abbb, abab\}$.

Definition 2.15. *Kleene star:* $L^* = \bigcup_{i=0}^{\infty} L^i$

This means that when $L \neq \emptyset$ and L is at most countable, L^* will be a countably infinite set containing all combinations of elements of L .

Example 2.16. Kleene star.

Suppose $L = \Sigma = \{a, b\}$.

Then $L^* = \{\lambda, a, b, aa, bb, ab, ba, aaa, bbb, \dots\}$

Example 2.17. A language.

Suppose $\Sigma = \{a, b\}$. We can define a language L over the alphabet Σ of words which start with symbol a as follows, with any of the following:

1. *set notation:* $\{aw \mid w \in \Sigma^*\}$
2. *context-free grammar:* $S \rightarrow Sb \mid Sa \mid a$
3. *regular expression:* $a(a + b)^*$

3 Regular expressions

A regular expression is a formal way of describing the class of languages called *regular languages* [2]. They serve a practical purpose in many tasks, such as *input validation, text editing, text searching* and more.

Many programming languages (Java, Python, ...) have their own implementation of regular expressions called *regex engines*, they are mainly used to parse input and quickly scan documents. The implementation of such regex engines may vary in different programming languages, which makes it hard to reason about the complexity of program subroutines.

3.1 Syntax

Given an alphabet Σ , a symbol $a \in \Sigma$, we define the set of regular expressions over Σ inductively, where e, e_1, e_2 are also regular expressions.

Definition 3.1. *Regular expressions.*

We define the syntax of a regular expression inductively, and provide the informal meaning of each construction.

syntax	informal meaning
λ	empty string
a	symbol
e_1e_2	concatenation of two regular expressions
$e_1 + e_2$	non-deterministic choice
e^*	zero-or-more repetitions
(e)	to allow unambiguous syntax
\emptyset	to denote the empty language

3.2 Semantics

Regular expressions are used to describe a class of languages called *regular languages* (**REG**). We call the set of all regular expressions **RegExp**, such that every regular expression is an element of this set. We define the function \mathcal{L} inductively, where e, e_1, e_2 are regular expressions and $\mathbb{P}(\Sigma^*)$ is the set of all subsets of Σ^* , therefore the set of all languages.

Definition 3.2. *The function $\mathcal{L} : \mathbf{RegExp} \rightarrow \mathbb{P}(\Sigma^*)$.*

1. $\mathcal{L}(\lambda) = \{\lambda\}$
2. $\mathcal{L}(a) = \{a\}$
3. $\mathcal{L}(e_1e_2) = \mathcal{L}(e_1)\mathcal{L}(e_2)$
4. $\mathcal{L}(e_1 + e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$
5. $\mathcal{L}(e^*) = \mathcal{L}(e)^*$
6. $\mathcal{L}(\emptyset) = \emptyset$

Example 3.3. $\mathcal{L}(a^*a)$.

We show the language described by the regular expression (a^*a) , using definition 3.2.

$$\begin{aligned}\mathcal{L}(a^*a) &= \mathcal{L}(a^*)\mathcal{L}(a) \\ &= (\mathcal{L}(a))^*\{a\} \\ &= \bigcup_{i=0}^{\infty} (\mathcal{L}(a))^i\{a\} \\ &= \{a, aa, aaa, \dots\}\end{aligned}$$

3.3 Properties

Regular languages have interesting properties. For instance, every *finite* language is regular.

Lemma 3.4. Every finite language is regular.

Suppose we have a finite language L . We know there is a finite number of strings $s \in L$ that is in this language. We prove L is regular by constructing a regular expression which describes L . Take the regular expression $s_0 + \dots + s_i$ for every $s_n \in L$. This regular expression describes exactly L .

Regular languages are not restricted to just finite languages. There are infinitely many regular languages, and for each of these regular languages, there exist infinitely many regular expressions to describe the same regular language.

Example 3.5. Regular languages.

Given an alphabet $\Sigma = \{a, b\}$. The following are all regular languages:

1. $L_1 = \{w \in \Sigma^* \mid w \text{ begins with } a \text{ and ends with } b\}$

This language is described by the regular expressions:

- (a) $a(a+b)^*b$
- (b) $a(a^*b^*)^*b$
- (c) $a(a+b)^*(b+a)^*b$

and infinitely many more variations.

2. $L_2 = \{w \in \Sigma^* \mid w \text{ contains } aaa \text{ as a substring}\}$

This language is described by the regular expressions:

- (a) $(a+b)^*aaa(a+b)^*$
- (b) $(a+b)^*aaa + aaa(a+b)^* + (a+b)^*aaa(a+b)^*$

and infinitely many more variations.

In this example we have shown some regular languages, and that a regular language can be described by multiple regular expressions.

Since there are multiple regular expressions possible to describe the same language, one might wonder if some regular expressions could be considered better than others when describing a language.

3.4 Equivalence

Regular expressions can have many forms as shown earlier. In order to reason about regular expressions we will define equalities in regular expressions as axioms. We say regular expressions are *equivalent* if they describe the same regular language. So for any regular expression e and e' , $e \equiv e'$ if and only if $\mathcal{L}(e) = \mathcal{L}(e')$.

Definition 3.6. (Defined in section 3 in [7]). Regular expression equivalence.

$$e_1 + (e_2 + e_3) \equiv (e_1 + e_2) + e_3 \quad (1)$$

$$e_1(e_2e_3) \equiv (e_1e_2)e_3 \quad (2)$$

$$e_1 + e_2 \equiv e_2 + e_1 \quad (3)$$

$$e_1(e_2 + e_3) \equiv e_1e_2 + e_1e_3 \quad (4)$$

$$(e_1 + e_2)e_3 \equiv e_1e_3 + e_2e_3 \quad (5)$$

$$e + e \equiv e \quad (6)$$

$$\lambda e \equiv e \quad (7)$$

$$\emptyset e \equiv \emptyset \quad (8)$$

$$e + \emptyset \equiv e \quad (9)$$

$$e^* \equiv \lambda + e^*e \quad (10)$$

$$e^* \equiv (\lambda + e)^* \quad (11)$$

These properties were introduced by Salomaa [7] where he used \emptyset^* to indicate an empty regular expression λ .

Lemma 3.7. For each pair of regular expressions $e \equiv e'$ in definition 3.6 we show that $\mathcal{L}(e) = \mathcal{L}(e')$.

Proof:

$$\begin{aligned} \mathcal{L}(e_1 + (e_2 + e_3)) &= \mathcal{L}(e_1) \cup \mathcal{L}(e_2 + e_3) \\ &= \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \cup \mathcal{L}(e_3) \\ &= \mathcal{L}(e_1 + e_2) \cup \mathcal{L}(e_3) \\ &= \mathcal{L}((e_1 + e_2) + e_3) \end{aligned} \quad (1)$$

$$\begin{aligned} \mathcal{L}(e_1(e_2e_3)) &= \mathcal{L}(e_1)\mathcal{L}(e_2e_3) \\ &= \mathcal{L}(e_1)\mathcal{L}(e_2)\mathcal{L}(e_3) \\ &= \mathcal{L}(e_1e_2)\mathcal{L}(e_3) \\ &= \mathcal{L}((e_1e_2)e_3) \end{aligned} \quad (2)$$

$$\begin{aligned} \mathcal{L}(e_1 + e_2) &= \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \\ &= \mathcal{L}(e_2) \cup \mathcal{L}(e_1) \\ &= \mathcal{L}(e_2 + e_1) \end{aligned} \quad (3)$$

$$\begin{aligned}
\mathcal{L}(e_1(e_2 + e_3)) &= \mathcal{L}(e_1)\mathcal{L}(e_2 + e_3) \\
&= \mathcal{L}(e_1)(\mathcal{L}(e_2) \cup \mathcal{L}(e_3)) \\
&= \mathcal{L}(e_1)\mathcal{L}(e_2) \cup \mathcal{L}(e_1)\mathcal{L}(e_3) \\
&= \mathcal{L}(e_1e_2) \cup \mathcal{L}(e_1e_3) \\
&= \mathcal{L}(e_1e_2 + e_1e_3)
\end{aligned} \tag{4}$$

$$\begin{aligned}
\mathcal{L}((e_1 + e_2)e_3) &= (\mathcal{L}(e_1) \cup \mathcal{L}(e_2))\mathcal{L}(e_3) \\
&= \mathcal{L}(e_1)\mathcal{L}(e_3) \cup \mathcal{L}(e_2)\mathcal{L}(e_3) \\
&= \mathcal{L}(e_1e_3) \cup \mathcal{L}(e_2e_3) \\
&= \mathcal{L}(e_1e_3 + e_2e_3)
\end{aligned} \tag{5}$$

$$\begin{aligned}
\mathcal{L}(e + e) &= \mathcal{L}(e) \cup \mathcal{L}(e) \\
&= \mathcal{L}(e)
\end{aligned} \tag{6}$$

$$\begin{aligned}
\mathcal{L}(\lambda e) &= \mathcal{L}(\lambda)\mathcal{L}(e) \\
&= \mathcal{L}(e)
\end{aligned} \tag{7}$$

$$\begin{aligned}
\mathcal{L}(\emptyset e) &= \mathcal{L}(\emptyset)\mathcal{L}(e) \\
&= \emptyset\mathcal{L}(e) \\
&= \emptyset \\
&= \mathcal{L}(\emptyset)
\end{aligned} \tag{8}$$

$$\begin{aligned}
\mathcal{L}(e + \emptyset) &= \mathcal{L}(e) \cup \mathcal{L}(\emptyset) \\
&= \mathcal{L}(e) \cup \emptyset \\
&= \mathcal{L}(e)
\end{aligned} \tag{9}$$

$$\begin{aligned}
\mathcal{L}(e^*) &= (\mathcal{L}(e))^* \\
&= \bigcup_{i=0}^{\infty} (\mathcal{L}(e))^i \\
&= \{\lambda\} \cup \bigcup_{i=1}^{\infty} (\mathcal{L}(e))^i \\
&= \{\lambda\} \cup \mathcal{L}(e^*e) \\
&= \mathcal{L}(\lambda + e^*e)
\end{aligned} \tag{10}$$

$$\begin{aligned}
\mathcal{L}(e^*) &= (\mathcal{L}(e))^* \\
&= \bigcup_{i=0}^{\infty} (\mathcal{L}(e))^i \\
&= \bigcup_{i=0}^{\infty} (\mathcal{L}(\lambda) \cup \mathcal{L}(e))^i \\
&= \bigcup_{i=0}^{\infty} (\mathcal{L}(\lambda + e))^i \\
&= \mathcal{L}((\lambda + e)^*)
\end{aligned} \tag{11}$$

We have proven all cases, which concludes this lemma. \square

4 Parsing expression grammars

The Parsing Expression Grammar (PEG) is a relatively new concept by Ford [4]. It is a formalism to describe languages which a packrat parser [3] can recognize/parse in linear time complexity. Ford based this formalism on earlier work by Birman & Ullman [1]. At first sight, the syntax of parsing expression grammars is similar to that of regular expressions and context-free grammars. However, there are subtle differences in the semantics.

4.1 Syntax

We provide a formal definition of Ford's parsing expression grammars [4], where Σ is the alphabet and N is a set of non-terminal symbols such that $\Sigma \cap N = \emptyset$. The non-terminal symbols will be used as tokens in production rules.

Definition 4.1. (Definition in 3.1 in [4]). Given an alphabet Σ , a symbol $a \in \Sigma$, a set of non-terminals N and a non-terminal symbol $A \in N$, we define the set of parsing expressions over Σ inductively, where p , p_1 and p_2 are also parsing expressions, a is a symbol in Σ and A is a non-terminal symbol in N .

syntax	informal meaning
ε	parse nothing
a	parse a
A	non-terminal symbol
p_1p_2	concatenation of two parsing expressions
p_1/p_2	prioritized choice
p^*	more-or-zero repetitions
$!p$	not-predicate
(p)	to allow for unambiguous syntax

Some important aspects which are also part of the syntax:

1. concatenation is right-associative: the parsing expression $p_1p_2p_3$ is interpreted as $p_1(p_2p_3)$
2. prioritized choice is right-associative: the parsing expression $p_1/p_2/p_3$ is interpreted as $p_1/(p_2/p_3)$
3. the precedence of the repetition operator is higher than the precedence of the not-predicate: the parsing expression $!p^*$ is interpreted as $!(p^*)$
4. the precedence of the not-predicate is higher than the precedence of concatenation: the parsing expression $!p_1p_2$ is interpreted as $!(p_1)p_2$
5. the precedence of concatenation is higher than the precedence of prioritized choice: the parsing expression p_1p_2/p_3 is interpreted as $(p_1p_2)/p_3$

Let P be the set of *parsing expressions*, then we define production rules as follows:

Definition 4.2. We define a set of production rules R as a set of tuples of the form (A, p) , where each non-terminal $A \in N$ has exactly one production rule such that $(A, p) \in R$ for some parsing expression $p \in P$. Rules are also written

$A \rightarrow p$.

We also define a function $\mathcal{R} : N \rightarrow P$ such that for any tuple $(A, p) \in R$, we have $\mathcal{R}(A) = p$.

Using the previous definitions, we complete the definition of Parsing Expression Grammars as:

Definition 4.3. A Parsing Expression Grammar (PEG) is a 4-tuple (N, Σ, R, p_s) where:

1. N is a finite set of non-terminal symbols;
2. Σ is an alphabet;
3. R is a finite set of production rules;
4. p_s is a parsing expression.

4.2 Semantics

We define a relation \rightsquigarrow which, given a parsing expression p and a string s , returns a result x , where $x \in \Sigma^* \cup \{\mathbf{failure}\}$ is either a string (suffix) s , s' or s'' which is not (yet) parsed or a token which indicates the failure of parsing. Our semantics (definition 4.4) are defined similar to Koprowski & Binsztok [5], the differences being that we omit their *step count* and directly apparent *syntactic sugar*.

Definition 4.4. (Figure 2 in [5]). *The relation $(p, s) \rightsquigarrow x$.*

$$\begin{array}{c}
\frac{}{(\varepsilon, s) \rightsquigarrow s} \varepsilon_s \qquad \frac{}{(a, \lambda) \rightsquigarrow \mathbf{failure}} \lambda_f \\
\\
\frac{}{(a, bs) \rightsquigarrow \mathbf{failure}} a_f \qquad \frac{}{(a, as) \rightsquigarrow s} a_s \\
\\
\frac{(\mathcal{R}(A), s) \rightsquigarrow x}{(A, s) \rightsquigarrow x} A_r \\
\\
\frac{(p_1, s) \rightsquigarrow \mathbf{failure}}{(p_1 p_2, s) \rightsquigarrow \mathbf{failure}} \cdot_f \qquad \frac{(p_1, s) \rightsquigarrow s' \quad (p_2, s') \rightsquigarrow x}{(p_1 p_2, s) \rightsquigarrow x} \cdot_s \\
\\
\frac{(p_1, s) \rightsquigarrow \mathbf{failure} \quad (p_2, s) \rightsquigarrow x}{(p_1/p_2, s) \rightsquigarrow x} /_f \qquad \frac{(p_1, s) \rightsquigarrow s'}{(p_1/p_2, s) \rightsquigarrow s'} /_s \\
\\
\frac{(p, s) \rightsquigarrow \mathbf{failure}}{(p^*, s) \rightsquigarrow s} *_f \qquad \frac{(p, s) \rightsquigarrow s' \quad (p^*, s') \rightsquigarrow s''}{(p^*, s) \rightsquigarrow s''} *_s \\
\\
\frac{(p, s) \rightsquigarrow \mathbf{failure}}{(!p, s) \rightsquigarrow s} !_s \qquad \frac{(p, s) \rightsquigarrow s'}{(!p, s) \rightsquigarrow \mathbf{failure}} !_f
\end{array}$$

We present the rules along with their intuitive meaning, where $\lambda, a, b, s \in \Sigma^*$ and $\varepsilon, a, p, p_1, p_2 \in P$.

- ε_s Base: $(\varepsilon, s) \rightsquigarrow s$.
Nothing is parsed.
- λ_f Base: $(a, \lambda) \rightsquigarrow \mathbf{failure}$.
Parsing fails due to lack of input.
- a_f Base: $(a, bs) \rightsquigarrow \mathbf{failure}$.
Parsing fails due to a non-matching symbol.
- a_s Base: $(a, as) \rightsquigarrow s$.
Parsing succeeds, symbol a is consumed.
- A_r If $p = \mathcal{R}(A)$ and $(p, s) \rightsquigarrow x$ then $(A, s) \rightsquigarrow x$.
A non-terminal symbol is replaced by its respective parsing expression.
- \cdot_f If $(p_1, s) \rightsquigarrow \mathbf{failure}$ then $(p_1 p_2, s) \rightsquigarrow \mathbf{failure}$.
Parsing fails because p_1 fails.
- \cdot_s If $(p_1, s) \rightsquigarrow s'$ and $(p_2, s') \rightsquigarrow x$ then $(p_1 p_2, s) \rightsquigarrow x$.
Parsing expression p_1 does not fail and might have consumed input, p_2 will continue with the remaining input.
- $/_f$ If $(p_1, s) \rightsquigarrow \mathbf{failure}$ and $(p_2, s) \rightsquigarrow x$ then $(p_1/p_2, s) \rightsquigarrow x$.
Parsing expression p_1 fails, parsing will continue with p_2 and the input.
- $/_s$ If $(p_1, s) \rightsquigarrow s'$ then $(p_1/p_2, s) \rightsquigarrow s'$ (parsing expression p_2 is not considered when p_1 does not result in a **failure**).
Parsing expression p_1 does not fail and might have consumed input, p_2 is discarded.
- $*_f$ If $(p, s) \rightsquigarrow \mathbf{failure}$ then $(p^*, s) \rightsquigarrow s$ (input is not consumed).
Parsing expression p fails, the input is returned as a result (of p^*).
- $*_s$ If $(p, s) \rightsquigarrow s'$ and $(p^*, s') \rightsquigarrow s''$ then $(p^*, s) \rightsquigarrow s''$.
Parsing expression p does not fail and might have consumed input, p^* will continue with the remaining input.
- $!_s$ If $(p, s) \rightsquigarrow \mathbf{failure}$ then $(!p, s) \rightsquigarrow s$ (input is not consumed).
Parsing expression p fails, the input is returned as a result (of $!p$).
- $!_f$ If $(p, s) \rightsquigarrow s'$ then $(!p, s) \rightsquigarrow \mathbf{failure}$.
Parsing expression p does not fail and might have consumed input, a failure is returned as a result (of $!p$).

Definition 4.5. The language L parsed by a parsing expression grammar $G = (N, \Sigma, R, p_s)$ is denoted $\mathcal{P}(G)$, its formal definition is:

$$\mathcal{P}(G) = \{s \mid s \in \Sigma^* \text{ and } (p_s, s) \rightsquigarrow \lambda\}$$

Example 4.6. $\mathcal{P}(G)$ with $G = (\emptyset, \{a, b, c\}, \emptyset, (a/ab)c$.

We will show the subtle difference in the prioritized choice by trying to parse the string abc using parsing expression grammar G , with parsing expression $(a/ab)c$.

$$\frac{\frac{\overline{(a, abc) \rightsquigarrow bc} \quad a_s}{((a/ab), abc) \rightsquigarrow bc} \quad /_s \quad \frac{\overline{(c, bc) \rightsquigarrow \mathbf{failure}} \quad a_f}{\cdot_s}}{\overline{((a/ab)c, abc) \rightsquigarrow \mathbf{failure}}}$$

The result of parsing expression $(a/ab)c$ will depend on the result of parsing expression (a/ab) , a branch is created in the proof tree to derive that result. That is, the result $((a/ab), abc) \rightsquigarrow r$. Since the branch does not result in a

failure, result r will be used in another branch to derive $(c, r) \rightsquigarrow r'$. We observe that, in $((a/ab), abc) \rightsquigarrow r$, the second option ab in the prioritized choice will not be considered since a is successfully parsed and consumed, thus $/_s$ is applied. Then we fill in $r = bc$ in (c, r) and have $(c, bc) \rightsquigarrow r'$ in the right branch. We observe that r' is a **failure** since c does not match b . As a result, the language parsed by G is $\mathcal{P}(G) = \{ac\}$.

Example 4.7. $\mathcal{P}(G)$ with $G = (\emptyset, \{a\}, \emptyset, a^*a)$.

We will show the subtle difference in the repetition by trying to parse the string aa using parsing expression grammar G , with parsing expression a^*a .

$$\frac{\frac{\frac{}{(a, a) \rightsquigarrow \lambda} a_s \quad \frac{\frac{}{(a, \lambda) \rightsquigarrow \mathbf{failure}} \lambda_f}{(a^*, \lambda) \rightsquigarrow \lambda} *f}}{(a^*, a) \rightsquigarrow \lambda} *s \quad \frac{}{(a, \lambda) \rightsquigarrow \mathbf{failure}} \lambda_f}{(a^*a, a) \rightsquigarrow \mathbf{failure}} \cdot_s$$

First, the parsing expression a^*a is split in two branches, the left branch will derive the result $(a^*, a) \rightsquigarrow r$, which will then be used in the right branch in $(a, r) \rightsquigarrow r'$. We observe that, in $(a^*, a) \rightsquigarrow r$, the input string a is entirely consumed, such that $r = \lambda$. Then we fill in $r = \lambda$ in (a, r) and have $(a, \lambda) \rightsquigarrow r'$ in which r' is a **failure** since the input string is empty. We therefore have that the language parsed by G is $\mathcal{P}(G) = \emptyset$. Compared to example 3.3, this is a subtle difference.

Examples 4.6 and 4.7 show subtle differences in semantics when compared to context-free grammars or regular expressions. These subtle differences make it non-trivial to transform regular expressions to parsing expression grammars which recognize the same language, as we will discuss later.

Theorem 4.8. *Given a parsing expression p and a string $s \in \Sigma^*$, if $(p, s) \rightsquigarrow x$ and $(p, s) \rightsquigarrow y$ then $x = y$. In other words, the relation \rightsquigarrow is a function.*

Proof:

Suppose there is a parsing expression p and a string $s \in \Sigma^*$ such that $(p, s) \rightsquigarrow x$ and $(p, s) \rightsquigarrow y$ with $x \neq y$, then there is at least one rule for which the semantics are ambiguous, meaning multiple proof trees are possible for the same parsing expression. We show that there is no ambiguity in proof trees for any pair (p, s) , by induction on the parsing expression p .

- $p = \varepsilon \quad \varepsilon_s$ This rule is applicable if $p = \varepsilon$. Thus there is exactly one proof tree with $x = s = y$.
- $p = a \quad \lambda_f$ This rule is applicable if $p = a$ and $s = \lambda$. Thus there is exactly one proof tree with $x = \mathbf{failure} = y$.
- $p = a \quad a_f$ This rule is applicable if $p = a$ and $s = bs'$. Thus there is exactly one proof tree with $x = \mathbf{failure} = y$.
- $p = a \quad a_s$ This rule is applicable if $p = a$ and $s = as'$. Thus there is exactly one proof tree with $x = s' = y$.

As induction hypothesis we assume: if $(p, s) \rightsquigarrow x$ and $(p, s) \rightsquigarrow y$ then $x = y$.

Note that concatenation is right-associative, such that $p_1p_2p_3$ is interpreted as $p_1(p_2(p_3))$.

In the case of concatenation we have:

- $p = p_1p_2 \quad \cdot_f$ This rule is applicable if $(p_1, s) \rightsquigarrow \mathbf{failure}$. Since concatenation is right-associative, there is exactly one proof tree for $(p_1p_2, s) \rightsquigarrow x$, $(p_1p_2, s) \rightsquigarrow y$ with $x = \mathbf{failure} = y$.
- $p = p_1p_2 \quad \cdot_s$ This rule is applicable if $(p_1, s) \rightsquigarrow s'$. By application of the induction hypothesis we know that if $(p_1, s) \rightsquigarrow x$ and $(p_2, s) \rightsquigarrow y$ then $x = s' = y$. Another application of the induction hypothesis tells us that if $(p_2, s') \rightsquigarrow x'$ and $(p_2, s') \rightsquigarrow y'$, then $x' = y'$. Since concatenation is right-associative, there is exactly one proof tree for $(p_1p_2, s) \rightsquigarrow x''$, $(p_1p_2, s) \rightsquigarrow y''$ with $x'' = x' = y' = y''$.

Note that prioritized choice is right-associative, such that $p_1/p_2/p_3$ is interpreted as $p_1/(p_2/p_3)$.

In the case of prioritized choice we have:

- $p = p_1/p_2 \quad /_f$ This rule is applicable if $(p_1, s) \rightsquigarrow \mathbf{failure}$. Since prioritized choice is right-associative, we apply the induction hypothesis on $(p_2, s) \rightsquigarrow z$ and there is exactly one proof tree for $(p_1/p_2, s) \rightsquigarrow x$, $(p_1/p_2, s) \rightsquigarrow y$ with $x = z = y$.
- $p = p_1/p_2 \quad /_s$ This rule is applicable if $(p_1, s) \rightsquigarrow s'$. By applying the induction hypothesis we know that if $(p_1, s) \rightsquigarrow x$ and $(p_1, s) \rightsquigarrow y$ then $x = y$, thus there is exactly one proof tree for $(p_1/p_2, s) \rightsquigarrow x'$, $(p_1/p_2, s) \rightsquigarrow y'$ with $x' = x = y = y'$.

In the case of a non-terminal we have:

- $p = A \quad A_r$ This rule is applicable if $(p, s) \rightsquigarrow z$ with $p = \mathcal{R}(A)$. We apply the induction hypothesis on $(p, s) \rightsquigarrow z$ and have exactly one proof tree for $(A, s) \rightsquigarrow x$, $(A, s) \rightsquigarrow y$ with $x = z = y$.

Note that the precedence of not-predicates is higher than the precedence of concatenation, such that $!p_1p_2$ is interpreted as $!(p_1)p_2$. In the case of a not-predicate we have:

- $p = !p' \quad !_s$ This rule is applicable if $(p', s) \rightsquigarrow \mathbf{failure}$. By application of the induction hypothesis we know that if $(p', s) \rightsquigarrow x$ and $(p', s) \rightsquigarrow y$ then $x = y$. Therefore we have exactly one proof tree for $(!p', s) \rightsquigarrow x'$ and $(!p', s) \rightsquigarrow y'$ with $x' = s = y'$.
- $p = !p' \quad !_f$ This rule is applicable if $(p', s) \rightsquigarrow s'$. By application of the induction hypothesis we know that if $(p', s) \rightsquigarrow x$ and $(p', s) \rightsquigarrow y$ then $x = s' = y$. Therefore we have exactly one proof tree for $(!p', s) \rightsquigarrow x'$ and $(!p', s) \rightsquigarrow y'$ with $x' = \mathbf{failure} = y'$.

All rules have been shown to contain no ambiguity, thus there is no parsing expression p for which we have $(p, s) \rightsquigarrow x$ and $(p, s) \rightsquigarrow y$ with $x \neq y$.

Thus, if $(p, s) \rightsquigarrow x$ and $(p, s) \rightsquigarrow y$ then $x = y$. □

Note that the proof did not include a case for parsing expressions of the form p^* because we prove in lemma 4.9 that the semantics of p^* can be derived by introducing a non-terminal A with production rule $A \rightarrow pA/\varepsilon$. So having proved $p = A$ implies proving $p = p^*$.

4.3 Syntactic sugar

There are numerous ways to extend the syntax of parsing expressions, we call such extensions syntactic sugar. They are essentially shorter ways to write expressions which were already covered by the base semantics. You might have noticed that the construction p^* is just syntactic sugar for $A \rightarrow pA / \varepsilon$, thus the $*$ and the rules $*_s$ and $*_f$ are redundant. However, Ford's definition of parsing expression grammars contains semantics for p^* constructions, so we have left it in. Subsequent proofs on parsing expression grammars will not consider the $*$ construction because of this.

sugar	desugared	informal meaning
p^*	A with $A \rightarrow pA / \varepsilon$	introduce a new non-terminal A
$\&p$	$!!p$	continue if the input matches p , does not consume input
p^+	pp^*	more-or-one, see example 4.7
$p?$	p/ε	one-or-none (also known as optional)
$.$	$(a_1/\cdots/a_n)$	parse any character $a_k \in \Sigma$

Lemma 4.9. Any parsing expression of the form p^* can be written as a non-terminal A with the production rule $A \rightarrow pA/\varepsilon$ such that for all $s \in \Sigma^*$, there exists an $s' \in \Sigma^*$ for which we have $(p^*, s) \rightsquigarrow s'$ if and only if $(A, s) \rightsquigarrow s'$.

Proof: Given any parsing expression p , we show that $(p^*, s) \rightsquigarrow s'$ if and only if $(A, s) \rightsquigarrow s'$ using the production rule $A \rightarrow pA/\varepsilon$.

Suppose $(p, s) \rightsquigarrow s'$, we then have the following proof tree for the rule $*_s$:

$$\frac{(p, s) \rightsquigarrow s' \quad (p^*, s') \rightsquigarrow s''}{(p^*, s) \rightsquigarrow s''} *_s$$

The same conclusion is achieved by the derivation of (A, s) :

$$\frac{\frac{\frac{(p, s) \rightsquigarrow s' \quad (A, s') \rightsquigarrow s''}{(pA, s) \rightsquigarrow s''} \cdot_s}{(pA / \varepsilon, s) \rightsquigarrow s''} /_s}{(A, s) \rightsquigarrow s''} A_r$$

Suppose $(p, s) \rightsquigarrow \mathbf{failure}$, we then have the following proof tree for the rule $*_f$:

$$\frac{(p, s) \rightsquigarrow \mathbf{failure}}{(p^*, s) \rightsquigarrow s} *_f$$

The same conclusion is achieved by the derivation of (A, s) :

$$\frac{\frac{(p, s) \rightsquigarrow \mathbf{failure}}{(pA, s) \rightsquigarrow \mathbf{failure}} \cdot_f \quad \frac{}{(\varepsilon, s) \rightsquigarrow s} \varepsilon_s}{(pA / \varepsilon, s) \rightsquigarrow s} /_f}{(A, s) \rightsquigarrow s} A_r$$

Which concludes the proof for this lemma. □

4.4 Properties

Parsing expression grammars were conceptualized as a result of Ford's paper [3] on linear-time parsing. The formalism has a subclass Ford refers to as *well-formed* parsing expression grammars[4], which ensures the totality (completeness) of the parsing expression grammar semantics. We show in example 4.10 that the semantics for parsing expression ε^* are not defined.

Example 4.10. *A badly formed parsing expression.*

$$\frac{\frac{}{(\varepsilon, s) \rightsquigarrow s} \varepsilon_s \quad (\varepsilon^*, s) \rightsquigarrow \dots}{(\varepsilon^*, s) \rightsquigarrow \dots} *_s$$

The application of the rule $*_s$ followed by the application of ε_s on its left branch will result in the reproduction of the ε^* expression in its right branch, without consuming any input. Thus there is no $x \in \Sigma^* \cup \{\mathbf{failure}\}$ such that $(\varepsilon, s) \rightsquigarrow x$.

Not only parsing expressions of the form ε^* have undefined semantics, parsing expression grammars do not allow for left-recursive production rules either, as demonstrated in example 4.11.

Example 4.11. *A left-recursive production rule.*

Suppose we have a non-terminal A and a production rule $A \rightarrow Ap$, we show that the semantics of the parsing expression A are undefined.

$$\frac{(Ap, s) \rightsquigarrow \dots}{(A, s) \rightsquigarrow \dots} A_r$$

Rewriting the non-terminal A to Ap using A_r should be followed by either \cdot_s or \cdot_f , depending on the success in deriving (A, s) . Since there is no $x \in \Sigma^* \cup \{\mathbf{failure}\}$ such that $(A, s) \rightsquigarrow x$, the semantics for parsing expression A are undefined.

We have shown two examples of parsing expressions where the semantics are undefined.

Definition 4.12. A parsing expression grammar $G = (N, \Sigma, R, p_s)$ is complete if and only if for every string $s \in \Sigma^*$ we have $(p_s, s) \rightsquigarrow x$ for some $x \in \Sigma^* \cup \{\mathbf{failure}\}$.

Following this definition, we observe that the parsing expression grammars in examples 4.10 and 4.11 are *not complete*.

Example 4.13. *A complete parsing expression grammar.*

Let $G = (N = \{A\}, \Sigma = \{a\}, R = \{A \rightarrow aA/\varepsilon\}, p_s = A)$.

We show that G is complete and $\mathcal{P}(G) = \{a^n \mid n \in \mathbb{N}\}$.

By induction on the structure of $s \in \Sigma^*$.

Suppose $s = \lambda$, we then have the following proof tree:

$$\frac{\frac{\frac{}{(a, \lambda) \rightsquigarrow \mathbf{failure}} \lambda_f}{(aA, \lambda) \rightsquigarrow \mathbf{failure}} \cdot_f \quad \frac{}{(\varepsilon, \lambda) \rightsquigarrow \lambda} \varepsilon_s}{\frac{(aA/\varepsilon, \lambda) \rightsquigarrow \lambda}{(A, \lambda) \rightsquigarrow \lambda} A_r} /_f$$

We assume $(A, s) \rightsquigarrow \lambda$ as induction hypothesis.

Suppose $s = as'$, we then have the following proof tree:

$$\frac{\frac{\frac{(a, as') \rightsquigarrow s'}{a_s} \quad \frac{(A, s') \rightsquigarrow \lambda}{\text{IH}}}{(aA, as') \rightsquigarrow \lambda} \quad /_s}{\frac{(aA/\varepsilon, as') \rightsquigarrow \lambda}{(A, as') \rightsquigarrow \lambda} A_r} \cdot_s$$

Thus for any $s \in \Sigma^*$ we have $(A, s) \rightsquigarrow \lambda$, which implies we have a complete grammar G with $\mathcal{P}(G) = \{a^n \mid n \in \mathbb{N}\}$. \square

4.5 Well-formedness

We introduce Ford's concept of well-formed parsing expression grammars [4], which will guarantee the completeness of parsing expression grammars, which is shown in theorem 4.15. As shown in examples 4.10 and 4.11 parsing expression grammars are not necessarily complete. A well-formed parsing expression grammar contains no direct left-recursive rules (as in 4.11).

Definition 4.14. (Defined in 3.6 in [4]). We define the set of *well-formed* parsing expressions P_{WF} by induction. We write $WF(p)$ to indicate that parsing expression p is *well-formed* with respect to its parsing expression grammar G .

1. $WF(\varepsilon)$
2. $WF(a)$
3. $WF(p_1p_2)$ if $WF(p_1)$ and $(p_1, s) \rightsquigarrow s$ implies $WF(p_2)$
4. $WF(p_1/p_2)$ if $WF(p_1)$ and $WF(p_2)$
5. $WF(A)$ if $WF(\mathcal{R}(A))$
6. $WF(!p)$ if $WF(p)$

A parsing expression grammar $G = (N, \Sigma, R, p_s)$ is well-formed if p_s and all parsing expressions in $\bigcup_{A \in N} \mathcal{R}(A)$ are well-formed.

Notice there is no definition for $WF(p^*)$, it is omitted since we have shown that p^* can be derived (lemma 4.9). Using the definition of well-formedness, one can verify whether parsing expression grammars are complete.

Theorem 4.15. *If a parsing expression grammar $G = (N, \Sigma, R, p_s)$ is well-formed, then for any input string $s \in \Sigma^*$ we have an $x \in \Sigma^* \cup \{\mathbf{failure}\}$ such that $(p_s, s) \rightsquigarrow x$, which implies that G is also complete.*

Proof:

We prove this by induction on the structure of parsing expression p_s .

The base cases are:

ε We have $(\varepsilon, s) \rightsquigarrow s$ for any $s \in \Sigma^*$ by definition of rule ε_s .

a We have three subcases:

1. Suppose $s = \lambda$, then $(a, s) \rightsquigarrow \mathbf{failure}$ by definition of rule λ_f .
2. Suppose s starts with a , then $(a, s) \rightsquigarrow s'$ for some $s' \in \Sigma^*$ by definition of rule a_s .
3. Suppose s does not start with a , then $(a, s) \rightsquigarrow \mathbf{failure}$ by definition of rule a_f .

As induction hypothesis we assume that for any parsing expression p that is well-formed with respect to its parsing expression grammar G and any $s \in \Sigma^*$ we have $(p, s) \rightsquigarrow x$ for some $x \in \Sigma^* \cup \{\mathbf{failure}\}$.

The inductive step cases are:

A We have $\mathcal{R}(A) = p'$, which is a well-formed parsing expression, by the induction hypothesis we know that $(p', s) \rightsquigarrow x$ for some $x \in \Sigma^* \cup \{\mathbf{failure}\}$.

$p_1 p_2$ We have well-formed parsing expression p_1 such that $(p_1, s) \rightsquigarrow x$ for some $x \in \Sigma^* \cup \{\mathbf{failure}\}$. We have three cases:

1. Suppose $(p_1, s) \rightsquigarrow \mathbf{failure}$, then we have $(p_1 p_2, s) \rightsquigarrow \mathbf{failure}$ by definition of rule \cdot_f .
2. Suppose $(p_1, s) \rightsquigarrow s$, then we have (p_2, s) by definition of rule \cdot_s , by the induction hypothesis we know $(p_2, s) \rightsquigarrow x$ for some $x \in \Sigma^* \cup \{\mathbf{failure}\}$.
3. Suppose $(p_1, s) \rightsquigarrow s'$, then we have (p_2, s') by definition of rule \cdot_s , by the induction hypothesis we know $(p_2, s') \rightsquigarrow x$ for some $x \in \Sigma^* \cup \{\mathbf{failure}\}$.

p_1/p_2 We have $WF(p_1)$ and $WF(p_2)$ by definition, application of the induction hypothesis gives us three cases:

1. Suppose $(p_1, s) \rightsquigarrow \mathbf{failure}$, then we have (p_2, s) by definition of rule $/_f$ and by the induction hypothesis we know that $(p_2, s) \rightsquigarrow x$ for some $x \in \Sigma^* \cup \{\mathbf{failure}\}$.
2. Suppose $(p_1, s) \rightsquigarrow s$, then by definition of rule $/_s$ we have $(p_1/p_2) \rightsquigarrow s$.
3. Suppose $(p_1, s) \rightsquigarrow s'$, then by definition of rule $/_s$ we have $(p_1/p_2) \rightsquigarrow s'$.

$!p$ We have $WF(p)$, by the induction hypothesis we know that $(p, s) \rightsquigarrow x$ for some $x \in \Sigma^* \cup \{\mathbf{failure}\}$, we have two cases:

1. Suppose $(p, s) \rightsquigarrow \mathbf{failure}$, then $(!p, s) \rightsquigarrow s$ by definition of rule $!_s$.
2. Suppose $(p, s) \rightsquigarrow s'$, then $(!p, s) \rightsquigarrow \mathbf{failure}$ by definition of $!_f$.

We have shown all cases, concluding the proof that if parsing expression grammar G is well-formed then G is complete. \square

5 Chomsky hierarchy

Chomsky defined a distinction in classes of languages [2]. The hierarchy of languages can be summarized as

$$\mathbf{REG} \subset \mathbf{CF} \subset \mathbf{CS} \subset \mathbf{RE}$$

Here we have

0. **RE**, the set of recursively enumerable languages
1. **CS**, the set of context-sensitive languages
2. **CF**, the set of context-free languages
3. **REG**, the set of regular languages

Parsing expression grammars are similar to context-free grammars in terms of syntax and semantics. The main differences being that parsing expression grammars are deterministic (by the prioritized choice construction p_1/p_2) and have some form of context-sensitivity (by using the construction $!p$). Context-free grammars do not enforce any order in which a non-terminal should be replaced by some production, and there is no context-sensitivity. We show, by transforming regular expressions to parsing expression grammars, that any regular language can be described using a parsing expression grammar.

5.1 Transforming regular expressions to parsing expression grammars

Medeiros et al. [6] have constructed an algorithm which *transforms* a regular expression e to a parsing expression grammar which *parses* the regular language $\mathcal{L}(e)$. Their transformation algorithm essentially splits the regular expression interpreted left-associatively and (repeatedly) transforms the right-most part of the regular expression until it has processed the entire regular expression to construct a parsing expression grammar.

To start the construction of a parsing expression grammar given a regular expression over an alphabet Σ , we initialize a parsing expression grammar $G = (N, \Sigma, R, p_s)$ with $N = \emptyset$, $R = \emptyset$ and $p_s = \varepsilon$. We then transform the regular expression using the following rules[6]:

Definition 5.1. (Figure 3 in [6]). Function Π , where $G = (N, \Sigma, R, p_s)$.

$$\begin{aligned}
 \Pi(\lambda, G) &:= G \\
 \Pi(a, G) &:= (N, \Sigma, R, ap_s) \\
 \Pi(e_1e_2, G) &:= \\
 &\quad \text{if } \Pi(e_2, G) = G' \\
 &\quad \text{then } \Pi(e_1e_2, G) = \Pi(e_1, G') \\
 \Pi(e_1 + e_2, G) &:= \\
 &\quad \text{if } \Pi(e_1, G) = (N', \Sigma, R', p'_s) \\
 &\quad \text{and } \Pi(e_2, (N', \Sigma, R', p_s)) = (N'', \Sigma, R'', p''_s) \\
 &\quad \text{then } \Pi(e_1 + e_2, G) = (N'', \Sigma, R'', p'_s/p''_s) \\
 \Pi(e^*, G) &:= \\
 &\quad \text{if } A \notin N \\
 &\quad \text{and } \Pi(e, (N \cup \{A\}, \Sigma, R, A)) = (N', \Sigma, R', p'_s) \\
 &\quad \text{then } \Pi(e^*, G) = (N', \Sigma, R' \cup \{A \rightarrow p'_s/p_s\}, A)
 \end{aligned}$$

To transform a regular expression e to a parsing expression grammar $G = (N, \Sigma, R, p_s)$, we can use the equivalence of regular expressions (definition 3.6) to our advantage. For instance, the regular expression a^*a describes the language $\mathcal{L}(a^*a) = \{a, aa, aaa, \dots\}$. The parsing expression grammar with parsing expression a^*a , however, describes \emptyset , as shown in example 4.7. Through the equivalence of e^* (rule 10 in definition 3.6) we observe that a^* is equal to $\lambda + aa^*$. This equality is used in the function Π , as we transform any regular expression e^* to a non-terminal A with $A \rightarrow p'_s/p_s$ where p'_s is the transformation of e and p_s is the transformation of the regular expression appended to the right of e^* , if applicable. The formal semantics are described in definition 5.1.

We now transform the regular expression used in example 4.7 to show that the transformation of regular expression a^*a indeed forms a parsing expression grammar G with $\mathcal{P}(G) = \mathcal{L}(a^*a)$.

Example 5.2. *The transformation of regular expression $e = a^*a$ to a parsing expression grammar G such that $\mathcal{L}(e) = \mathcal{P}(G)$.*

First, we construct a new parsing expression grammar $G_0 = (N, \Sigma, R, p_s)$ with $N = \emptyset, \Sigma = \{a\}, R = \emptyset$ and $p_s = \varepsilon$. We now use the function Π to transform the regular expression as follows.

$$\Pi(a^*a, G_0) = \text{if } \Pi(a, G_0) = G_1 \text{ then } \Pi(a^*a, G_0) = \Pi(a^*, G_1)$$

We first construct G_1 as an intermediate step:

$$\begin{aligned} G_1 &= \Pi(a, G_0) \\ &= \Pi(a, (\emptyset, \Sigma, \emptyset, \varepsilon)) \\ &= (\emptyset, \Sigma, \emptyset, a) \end{aligned}$$

We then construct $G_3 = \Pi(a^*, G_1)$:

$$\begin{aligned} \text{if } G_2 &= \Pi(a, (\{A\}, \Sigma, \emptyset, A)) \\ &= (\{A\}, \Sigma, \emptyset, aA) \\ \text{then } G_3 &= \Pi(a^*, G_1) \\ &= (\{A\}, \Sigma, \{A \rightarrow aA/a\}, A) \end{aligned}$$

And indeed, $\mathcal{P}(G_3) = \mathcal{L}(a^*a)$.

Suppose we want to transform a regular expression like $(a + ab)c$, which describes the language $\mathcal{L}((a + ab)c) = \{ac, abc\}$. Example 4.6 shows that the parsing expression grammar with parsing expression $(a/ab)c$ does not parse the string abc . Therefore, we use the equivalence of $(e_1 + e_2)e_3$ (rule 5 in definition 3.6) to show that the regular expression $(a + ab)c$ is equivalent to $ac + abc$. The parsing expression grammar with parsing expression ac/abc correctly parses the strings in $\mathcal{L}((a + ab)c)$. This equivalence is also used in the function Π , where any regular expression with a non-deterministic choice will be converted in a deterministic parsing expression.

Example 5.3. *The transformation of regular expression $e = (a + ab)c$ to a parsing expression grammar G such that $\mathcal{L}(e) = \mathcal{P}(G)$.*

First, we construct a new parsing expression grammar $G_0 = (N, \Sigma, R, p_s)$ with

$N = \emptyset$, $\Sigma = \{a, b, c\}$, $R = \emptyset$ and $p_s = \varepsilon$. We now use the function Π to transform the regular expression as follows.

$$\Pi((a + ab)c, G_0) = \text{if } \Pi(c, G_0) = G_1 \text{ then } \Pi((a + ab)c, G_0) = \Pi((a + ab), G_1)$$

We first construct G_1 as an intermediate step:

$$\begin{aligned} G_1 &= \Pi(c, G_0) \\ &= \Pi(c, (\emptyset, \Sigma, \emptyset, \varepsilon)) \\ &= (\emptyset, \Sigma, \emptyset, c) \end{aligned}$$

We then construct $G_4 = \Pi((a + ab), G_1)$:

$$\begin{aligned} \text{if } G_2 &= \Pi(a, G_1) \\ &= \Pi(a, (\emptyset, \Sigma, \emptyset, c)) \\ &= (\emptyset, \Sigma, \emptyset, ac) \\ &= (N', \Sigma, R', p'_s) \\ \text{and } G_3 &= \Pi(ab, (N', \Sigma, R', p'_s)) \\ &= \Pi(ab, (\emptyset, \Sigma, \emptyset, c)) \\ &= \Pi(\emptyset, \Sigma, \emptyset, abc) \\ &= \Pi(N'', \Sigma, R'', p''_s) \\ \text{then } G_4 &= \Pi((a + ab), G_1) \\ &= (N'', \Sigma, R'', p'_s/p''_s) \\ &= (\emptyset, \Sigma, \emptyset, ac/abc) \end{aligned}$$

And indeed, $\mathcal{P}(G_4) = \mathcal{L}((a + ab)c)$.

In general, Medeiros et al. [6] have shown the transformation of regular expressions to parsing expression grammars to be correct for what they refer to as *well-formed regular expressions*. That is, for any well-formed regular expression e a parsing expression grammar G can be constructed using Π such that $\mathcal{P}(G) = \mathcal{L}(e)$. They have expanded their work by providing a general algorithm which transforms any regular expression to a well-formed regular expression, such that parsing expression languages is a strictly larger set of languages than the set of regular languages.

5.2 Class of languages

Parsing expression grammars resemble the syntax and semantics of context-free grammars, and they are able to parse any regular expression (by transforming the regular expression to an equivalent parsing expression grammar). The parsing expression grammar formalism is not limited to regular languages, however. We show (lemma 5.4) that parsing expression grammars are able to describe context-free languages.¹

Since parsing expression grammars also possess some kind of context-sensitivity by the $!_s$ and $!_f$ semantics, we also show (lemma 5.5) that the parsing expression grammar formalism is able to describe some context-sensitive languages.

¹not necessarily all, this is an open problem.

Lemma 5.4. Parsing expression grammars are able to describe some non-regular languages.

Proof:

We define a parsing expression grammar G such that $\mathcal{P}(G) = \{a^n b^n \mid n \in \mathbb{N}\}$ to show parsing expression grammars are not limited to regular languages. Since we know that $L = \{a^n b^n \mid n \in \mathbb{N}\}$ is not a regular language.

Let $G = (N = \{S\}, \Sigma = \{a, b\}, R = \{S \rightarrow aSb/\varepsilon\}, p_s = S)$.

By induction on the structure of $s \in L$.

Suppose $s = \lambda$, we then have the following proof tree:

$$\frac{\frac{\frac{(a, \lambda) \rightsquigarrow \mathbf{failure}}{\lambda_f} \cdot f}{(aSb, \lambda) \rightsquigarrow \mathbf{failure}} \cdot f}{(aSb/\varepsilon, \lambda) \rightsquigarrow \lambda} \cdot f}{(S, \lambda) \rightsquigarrow \lambda} A_r \quad \frac{(\varepsilon, \lambda) \rightsquigarrow \lambda}{\varepsilon_s} \cdot f$$

We assume $(S, s) \rightsquigarrow \lambda$ as induction hypothesis.

Suppose $s = as'b$, we then have the following proof tree:

$$\frac{\frac{\frac{(a, as'b) \rightsquigarrow s'b}{a_s} \cdot a_s}{(aSb, as'b) \rightsquigarrow \lambda} \cdot a_s}{(aSb/\varepsilon, as'b) \rightsquigarrow \lambda} \cdot a_s}{(S, as'b) \rightsquigarrow \lambda} A_r \quad \frac{\frac{\frac{(S, s'b) \rightsquigarrow b}{\text{IH}} \cdot \text{IH}}{(Sb, s'b) \rightsquigarrow \lambda} \cdot s}{(b, b) \rightsquigarrow \lambda} \cdot s}{(Sb, s'b) \rightsquigarrow \lambda} \cdot s$$

We have shown that the language $L = \{a^n b^n \mid n \in \mathbb{N}\}$, which is typically used as an example of a *non-regular* language, can be parsed by a parsing expression grammar. \square

Intuitively, the parsing expression grammar simply uses a production rule similar to what one could use to define the language with a context-free grammar. Since they are not left-recursive, the same production rules can be used.

Lemma 5.5. Parsing expression grammars are able to describe some context-sensitive languages.

Proof:

We define a parsing expression grammar G such that $\mathcal{P}(G) = \{a^n b^n c^n \mid n \in \mathbb{N}\}$.

Since we know that $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not a context-free language.

Let $G = (N, \Sigma, R, p_s)$ where

$$N = \{A, X, Y, S\}$$

$$\Sigma = \{a, b, c\}$$

$$R = \{A \rightarrow aA/\varepsilon, X \rightarrow aXb/\varepsilon, Y \rightarrow bYc/\varepsilon, S \rightarrow !(X!b)AY\}$$

$$p_s = S$$

Suppose $s = a^n b^n c^n$ with $n \in \mathbb{N}$, we then have the following proof tree:

$$\frac{\frac{\text{(analogue to lemma 5.4)}}{(X, a^n b^n c^n) \rightsquigarrow c^n} \quad \frac{\frac{\frac{(b, c^n) \rightsquigarrow \mathbf{failure}}{(!b, c^n) \rightsquigarrow c^n} a_f}{!_s}}{!_s}}{(X!b), a^n b^n c^n) \rightsquigarrow c^n} \cdot_s}{\frac{\frac{\frac{(!!(X!b), a^n b^n c^n) \rightsquigarrow a^n b^n c^n}{!_s}}{!_f}}{!_s}}{(!!(X!b)AY, a^n b^n c^n) \rightsquigarrow \lambda} \quad \frac{\frac{\text{(analogue to example 4.13)}}{(A, a^n b^n c^n) \rightsquigarrow b^n c^n} \quad \frac{\text{(analogue to lemma 5.4)}}{(Y, b^n c^n) \rightsquigarrow \lambda}}{(AY, a^n b^n c^n) \rightsquigarrow \lambda} \cdot_s} \cdot_s}
\frac{(!!(X!b)AY, a^n b^n c^n) \rightsquigarrow \lambda}{(S, a^n b^n c^n) \rightsquigarrow \lambda} A_r$$

We have shown that the language $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$, which is typically used as an example of a *non-context-free* language, can be parsed by a parsing expression grammar. \square

6 Discussion

In this thesis, we have introduced the concept of parsing expression grammars by Ford [4] and explored the semantics of this formalism in a proof-tree style by Koprowski & Binsztok [5]. We have shown that the class of languages of parsing expression grammars is non-trivial. Medeiros et al. [6] have shown that all regular expression can be transformed to parsing expression grammars, which implies that any regular language can be parsed in linear-time by a packrat parser [3]. Whether parsing expression grammars are able to parse *all context-free languages* is still an open question.

References

- [1] Alexander Birman and Jeffrey D Ullman. “Parsing algorithms with back-track”. In: *IEEE Conference Record of 1970 Eleventh Annual Symposium on Switching and Automata Theory*. IEEE. 1970, pp. 153–174.
- [2] Noam Chomsky. “On certain formal properties of grammars”. In: *Information and control* 2.2 (1959), pp. 137–167.
- [3] Bryan Ford. “Packrat parsing:: simple, powerful, lazy, linear time, functional pearl”. In: *ACM SIGPLAN Notices*. Vol. 37. 9. ACM. 2002, pp. 36–47.
- [4] Bryan Ford. “Parsing expression grammars: a recognition-based syntactic foundation”. In: *ACM SIGPLAN Notices*. Vol. 39. 1. ACM. 2004, pp. 111–122.
- [5] Adam Koprowski and Henri Binsztok. “TRX: A formally verified parser interpreter”. In: *Programming Languages and Systems*. Springer, 2010, pp. 345–365.
- [6] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimsky. “From regexes to parsing expression grammars”. In: *Science of Computer Programming* 93 (2014), pp. 3–18.
- [7] Arto Salomaa. “Two complete axiom systems for the algebra of regular events”. In: *Journal of the ACM (JACM)* 13.1 (1966), pp. 158–169.