Bachelor thesis
Computer Science

Radboud University

Parsing expression grammars, constructing a linear-time parser

Author:
Jan Martens
s4348435

First supervisor/assessor:
Herman Geuvers
herman@cs.ru.nl

Second assessor:
Freek Wiedijk
freek@cs.ru.nl

January 23, 2017

Abstract

Parsing is the process of analysing a string of symbols. Within parsing lookahead is a very useful concept. However, unlimited lookahead often causes the time complexity of a parser to increase significantly. Bryan Ford has introduced a recognition-based grammar called *parsing expression grammars*[6]. With parsing expression grammars one has unlimited lookahead and one can write a linear-time parser for every language that can be described using a parsing expression grammar.

In this thesis we will show how these parsing expression grammars work in practice and how one can construct a linear-time parser.

Contents

Chapter 1

Introduction

Since the introduction of Chomsky's generative system of grammars[4]. It is very popular to use generative grammars such as context-free grammars (CFG's) and regular expressions, to express syntax and programming language languages. Because these generative grammars are non-deterministic it is hard to create an efficient parser for these grammars.

Minimal recognition schemas such as TDPL [2] provide linear time parsing. They are very minimal, and provide little tools for software engineers, and are very impractical to use. Based on the ideas of TDPL, Ford[6] created parsing expression grammars. Parsing expression grammars(PEGs) look very much like context-free grammars, while still being deterministic. This is caused by using *prioritized-choice* instead of the *non-deterministic choice* in context free grammars.

In this paper we will introduce parsing expression grammars, showing the power but also the differences with CFGs. We will also show that it is not trivial that we can parse any PEG in linear time. After that we will explain how one can reduce parsing expression grammars to a more minimal form, and finally we will show how a PEG can be reduced to a minimal recognition schema with guaranteed linear-time parsing.

Chapter 2

Preliminaries

2.1 Formal Languages

A formal language is a set of words that consists of symbols out of an alphabet.

Definition 2.1. ***Alphabet***: *An alphabet is a finite set, often denoted as Σ. Elements of an alphabet we call symbols, or letters.*

Definition 2.2. ***Word***: *A word is a combination of letters in an alphabet. We call ε the empty word and is the combination of zero elements over an alphabet Σ. We call Σ^* the set of all possible words over Σ. This set is inductively defined by:*

- $\varepsilon \in \Sigma^*$
- $wa \in \Sigma^*$ *if* $w \in \Sigma^*$ *and* $a \in \Sigma$

Definition 2.3. ***Language***: *Given an alphabet Σ, the language L is a subset of all the possible words in that alphabet. Formally that is, $L \subseteq \Sigma^*$.*

We can define operations such as concatenation or reverse on words, this can be very useful if we want to formally describe languages.

Definition 2.4. ***Concatenation:*** *We can define concatenation inductively as an operation: $\Sigma^* \times \Sigma^* \to \Sigma^*$.*

- $w \cdot \varepsilon = w$
- $w \cdot (va) = (w \cdot v)a$

Definition 2.5. ***Length:*** *We define the length of a word $|.| : \Sigma^* \to \mathbb{N}$ inductively as follows.*

- $|\varepsilon| = 0$

- $|wa| = |w| + 1$

Example 2.6.

1. *We define $\Sigma = \{a, b\}$, now Σ^* contains the words with only a's and b's.*
2. *$\{w|w$ does not contain $aa\}$ is a language over Σ.*
3. *$\{a^n b^n c^n | n \in \mathbb{N}\}$ is a language over the alphabet $\{a, b, c\}$*

2.2 Regular expressions

A well known formalism to describe languages are regular expressions. Regular expression are widely used in modern day programming languages and are very useful to quickly recognize and find patterns in strings. Languages which can be described using a regular expression are called the regular languages[4].

Definition 2.7. ***Regular expressions****: Given a finite alphabet Σ the set of regular expressions over Σ are defined inductively as follows.*

1. *The following base cases are regular expressions over Σ:*
   - *$\emptyset$, representing the empty set.*
   - *ε, representing the language containing only the empty word ε.*
   - *$a \in \Sigma$, representing the language containing only the word a.*
2. *Let R and S be regular expressions over Σ. Then,*
   - *$R + S$*
   - *RS*
   - *R^**

   are also regular expressions

Now we define the language $\mathcal{L}(e)$ of regular expression e over alphabet Σ as:

1. *$\mathcal{L}(\emptyset) = \emptyset$*
2. *$\mathcal{L}(\varepsilon) = \{\varepsilon\}$*
3. *$\mathcal{L}(a) = \{a\}$, with $a \in \Sigma$*
4. *$\mathcal{L}(e_1 + e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$ with e_1 and e_2 regular expressions*

5. *$\mathcal{L}(e_1 e_2) = \mathcal{L}(e_1)\mathcal{L}(e_2)$ with e_1 and e_2 regular expressions*

6. *$\mathcal{L}(e^*) = (\mathcal{L}(e))^*$ with e regular expressions*

Example 2.8. *With $\Sigma = \{a, b\}$, the following are regular expressions.*

- *$L_1 = \{a^n |\ n \in \mathbb{N}\}$ $L_1 = \mathcal{L}(a^*)$*
- *$L_2 = \{w |\ |w|$ is even$\}$ $L_2 = \mathcal{L}(((a+b)(a+b))^*)$*
- *$L_3 = \{a^n b^n |\ n \in \mathbb{N}\}$ is not a regular language since one can't make a regular expression describing the language.[4]*

2.3 Context free grammars

The language $\{a^n b^n |\ n \in \mathbb{N}\}$ is not regular. However it belongs to another widely used language class called the context-free languages. Context-free languages are all languages that can be described with a CFG (context-free grammar). We will now define the notation of context-free grammars to formally describe context-free languages.

Definition 2.9. ***Context-free grammars*** *A CFG G is given by a 4-tuple (V, Σ, R, S) where*

- *V is a finite set of non-terminals*
- *Σ is the finite alphabet also called terminals. We chose $\Sigma \cap V = \emptyset$, there are no symbols both non-terminal and terminal.*
- *R is a relation $R : V \to (V \cup \Sigma)^*$, these relations are called the production rules, and are written in the form $A \to w$, where $A \in V$ and $w \in (V \cup \Sigma)^*$.*
- *S is the start expression and $S \in V$*

Example 2.10. ***Context free grammars***
Let CFG $G = (V, \Sigma, R, S)$, where:

- *$V = S, B$*
- *$\Sigma = \{a, b\}$*
- *$R = \{S \to aSb |\ \varepsilon\}$*
- *$S = S$*

This CFG has two production rules, $S \to aSb$ and $S \to \varepsilon$. We can now use these production rules to replace any non-terminal symbol $A \in V$ with the right hand side of a rule. For example , if we have $A \to w \in R$ than $xAz \to xwz$. We denote a derivation path $w \to w_1 \to w_2 \to ... \to w_n$ by $w \Rightarrow w_n$

Definition 2.11. ***Context-free languages*** *The language of CFG $G = (V, \Sigma, R, S)$ is defined by $\mathcal{L}(G) = \{w|\ S \Rightarrow w\}$. That is, there is a derivation path from the start expressions to the given word w.*

Example 2.12. *Using the CFG from example 3, we can see that ab is part of the language since:*

$S \rightarrow aSb \rightarrow a\varepsilon b \rightarrow ab$
aabb is also part of the language generated by G since:

$S \rightarrow aSb \rightarrow aaSSbb \rightarrow aaSSbb \rightarrow aabb$

Note that the order of replacing non-terminal symbols does not matter.

We can see that the language we made is in fact $\mathcal{L}(G) = \{a^n b^n|\ n \in \mathbb{N}\}$. You can't construct all languages with context-free grammars. A well-known language which is not context-free is $L = \{a^n b^n c^n | n \in \mathbb{N}\}$

Chapter 3

Parsing Expression Grammars

3.1 Parsing expressions

In this chapter we introduce the concept of parsing expression grammars introduced by Bryan Ford[6]. We also will provide an alternative syntax to simplify the derivation of parsing expression grammars.

Definition 3.1. *A Parsing expression grammar (PEG) is a tuple $G = (V_N, V_T, R, e_s)$, where:*

- *V_N is a finite set of non terminal symbols.*
- *V_T is a finite set of terminal symbols.*
- *R is a set of tuples (A, e). Where $A \in V_N$ and e is a parsing expression. R must be deterministic, so there are no two pairs $(A, e), (A, e') \in R$. The set R forms the production rules. If $(A, e) \in R$ we write $A \rightarrow e$.*
- *e_s is the start expression.*

Definition 3.2. *Given the set of terminals V_T and a set of non terminals V_N. The set of parsing expressions $\mathcal{E}$, is defined inductively with the following seven rules.*

Rule	***Meaning***
1. $\varepsilon \in \mathcal{E}$	*Parse nothing.*
2. $a \in V_T$, $a \in \mathcal{E}$	*Parse terminal a.*
3. $A \in V_N$, $A \in \mathcal{E}$	*Non terminal A.*
4. If $e_1, e_2 \in \mathcal{E}$, than $e_1e_2 \in \mathcal{E}$	*Concatenate two expressions.*
5. If $e_1, e_2 \in \mathcal{E}$, than $e_1/e_2 \in \mathcal{E}$	*Prioritized choice.*
6. If $e \in \mathcal{E}$, than $e^* \in \mathcal{E}$	*Zero or more repetitions of e.*
7. If $e \in \mathcal{E}$, than $!e \in \mathcal{E}$	*Not-predicate e.*

3.2 Grammar

In this section we will introduce rules which define the syntactic meaning of a parsing expression grammar $G = (V_N, V_T, R, e_s)$. This definition is slightly different from Ford's original definition[6] in the sense that it uses a tree syntax, and the step counter is omitted. We define a relation $\rightarrow_G$ between pairs in the form (e, x) and output o where e is a parsing expression, $x \in V_T^*$ is the input string and $o \in V_T^* \cup \{fail\}$ is the output. The output o is either a string or the representation for a fail: $fail \notin V_T$.

The relation $\rightarrow_G$ is defined inductively using derivation rules. This means we will write rules in the form of *If* $(e, x) \rightarrow_G o$*, than* $(e', y) \rightarrow_G u$ as a derivation tree, that is:

$$\frac{(e, x) \rightarrow_G o}{(e', y) \rightarrow_G u}$$

For the pairs $((e, x), o)) \in\rightarrow_G$ we will write $(e, x) \rightarrow o$ omitting the reference to G for readability.

1. **Parse nothing**:

$$(\varepsilon, x) \rightarrow \varepsilon$$

2. **Terminal (success)**: Parse $a \in V_T$ from the begin of the word.

$$(a, ax) \rightarrow a$$

3. **Terminal (failure)**: Try to parse $a \in V_T$ but find $b \in V_T$ with $a \neq b$, a can't be matched so return $fail$.

$$(a, bx) \rightarrow fail$$

4. **Non-terminal**: Replace non-terminal $A \in V_N$ with e, if $(A, e) \in R$.

$$\frac{(e, x) \rightarrow o}{(A, x) \rightarrow o}$$

5. **Concatenation(success)** : Match e_1 and e_2 in that order. If both succeed consuming x_1 and x_2 consecutively, the concatenation succeeds consuming x_1x_2.

$$\frac{(e_1, x_1x_2y) \rightarrow x_1 \qquad (e_2, x_2y) \rightarrow x_2}{(e_1e_2, x_1x_2y) \rightarrow x_1x_2}$$

6. **Concatenation(failure 1)** : If e_1 results in a $fail$, the concatenation fails without trying to match e_2.

$$\frac{(e_1, x_1x_2y) \to fail}{(e_1e_2, x_1x_2y) \to fail}$$

7. **Concatenation(failure 2)** : If e_1 succeeds consuming x_1 and e_2 results in $fail$, the concatenation results in $fail$.

$$\frac{(e_1, x_1x_2y) \to x_1 \qquad (e_2, x_2y) \to fail}{(e_1e_2, x_1x_2y) \to fail}$$

8. **Prioritized choice(case 1)**: If choice e_1 succeeds on the input string consuming x. Choice e_2 will not be matched and the expression will succeed consuming x.

$$\frac{(e_1, xy) \to x}{(e_1/e_2, xy) \to x}$$

9. **Prioritized choice(case 2)** : If choice e_1 is matched on input xy but fails. Match e_2 on the same input and return its output. Note that e_2 can either $fail$ or succeed consuming a part of the input.

$$\frac{(e_1, xy) \to fail \qquad (e_2, xy) \to o}{(e_1/e_2, xy) \to o}$$

10. **Repetition(base case)** : If e fails, e^* will succeed without consuming any input.

$$\frac{(e, x) \to fail}{(e^*, x) \to \varepsilon}$$

11. **Repetition(recursive case)** : e succeeds consuming x_1 and call e^* recursively on the remaining part.

$$\frac{(e, x_1x_2y) \to x_1 \qquad (e^*, x_2y) \to x_2}{(e^*, x_1x_2y) \to x_1x_2}$$

12. **Not-predicate(success)** : If e fails on the input string, $!e$ will succeed without consuming any input. This is parsing zero repetitions of e.

$$\frac{(e, x) \to fail}{(!e, x) \to \varepsilon}$$

13. **Not-predicate(failure)** : If e succeeds on the input $!e$ will result in $fail$.

$$\frac{(e, xy) \to x}{(!e, xy) \to fail}$$

3.3 Syntactic sugar

For readability of our parsing expressions we introduce syntactic sugar.

Definition 3.3.

1. *We consider . as any character in V_T. If $V_T = \{a_1, a_2, ..., a_n\}$. We can denote . by $a_1/a_2/.../a_n$.*
2. *We define &e $=!(!e)$. Which will behave like an and-predicate. If e succeeds continue and consume nothing, if e fails return $fail$.*

3.4 Behaviour

In this section we will show some example parsing expression grammars describing key languages and we will highlight the differences with Chomsky's generative way of describing formal languages.

Definition 3.4. *Given a PEG $G = (V_N, V_T, R, e_s)$, we define the language produced by G as $L(G) = \{w \in V_T^* \mid (e_s, w) \to_G w\}$. This is a different definition as explained in [6] but we will show their equivalence in Chapter 4.*

This means that a word is only accepted when the derivation does not result in $fail$ and the word is consumed. In Chapter 4 we will show some alternative language definitions.

Example 3.5. *We formally describe the language shown in Example 2.10 that is $L = \{a^n b^n | n \in \mathbb{N}\}$,using a parsing expression grammar. We define the peg $G = (\{a, b\}, \{A\}, R, A)$ where the set of production rules R consists of one rules:*

- $A \to aAb/\varepsilon$

The derivation of $aabb \in L$ is:

$$\dfrac{\dfrac{(a, aabb) \to a \qquad \dfrac{\dfrac{\dfrac{\dfrac{(a, ab) \to a \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(a, b) \to fail}{(aAb, b) \to fail} \qquad (\varepsilon, b) \to \varepsilon}{(aAb/\varepsilon, b) \to \varepsilon}}{(aAb/\varepsilon, b) \to \varepsilon}}{(A, b) \to \varepsilon} \qquad (b, b) \to b}{(Ab, b) \to b}}{(aAb, abb) \to ab}}{(aAb/\varepsilon, abb) \to ab}}{(A, abb) \to ab} \qquad (b, b) \to b}{(Ab, abb) \to abb}}{(aAb, aabb) \to aabb}}{\dfrac{(aAb/\varepsilon, aabb) \to aabb}{(A, aabb) \to aabb}}$$

Note that A parses $a^n b^n$ in a similar way as the CFG in Example 2.10.

The derivation tree of $aab \notin L$ would be as follows, where the derivation ending with I *should be plugged onto the top of place* I*:*

$$\dfrac{\dfrac{\dfrac{\text{I}}{(aAb, aab) \to fail} \qquad (\varepsilon, aab) \to \varepsilon}{(aAb/\varepsilon, aab) \to \varepsilon}}{(A, aab) \to \varepsilon}$$

$$\dfrac{(a, aab) \to a \qquad \dfrac{\dfrac{\dfrac{\dfrac{(a, ab) \to a \qquad \dfrac{\dfrac{\dfrac{\dfrac{(a, b) \to fail}{(aAb, b) \to fail} \qquad (\varepsilon, b) \to \varepsilon}{(aAb/\varepsilon, ab) \to \varepsilon}}{(A, b) \to \varepsilon} \qquad (b, b) \to b}{(Ab, b) \to b}}{(aAb, ab) \to ab}}{(aAb/\varepsilon, ab) \to ab}}{(A, ab) \to ab} \qquad (b, \varepsilon) \to fail}{(Ab, ab) \to fail}}{\dfrac{(aAb, aab) \to fail}{\text{I}}}$$

We see that aab is not accepted since it is not parsed fully.

Example 3.6. *Although The form of the production rules in this example may look like the form of the production rules in 2.10, it is important to see that the syntax has some important differences.*

In the following example we will show a more practical example to show the difference between the non-deterministic choice in context free grammar rules and the prioritized choice with parsing expression grammars. Let's take this small edited part out of the declaration of the C-syntax[3].

```
<if-statement> ::= if <expression> then <statement>
                 | if <expression> then <statement> else <statement>
```

Note this is in Backus-Naur form (BNF)[5] which is a well known way of formally defining the syntax of a programming language. We can translate this statement to a CFG $G = (\{IF\}, \{if, else, then, e, s\}, R, IF\}$.

- $V = \{IF\}$
- $\Sigma = \{if, else, s, e, then\}$
- $R = \{IF \to$ *if e then s* $|$*if e then s else s* $\}$
- $V = \{IF\}$

We see that this would actually parse either an if statement or an if-else statement. If we straightforwardly translate this in a parsing expression grammar without paying attention to the prioritized choice difference, we would make PEG $G' = \{V_N, V_T, R, e_s\}$, where:

- $V_N = \{IF\}$
- $V_T = \{if, else, s, e, then\}$
- $R = \{IF \rightarrow (\textit{if e then s })/(\textit{if e then s else s}) \}$
- $e_s = IF$

Parsing an if-else statement using this PEG would not result in the correct parsing.

$$\dfrac{\dfrac{\dfrac{\ldots}{(\textit{if e than s}, \textit{if e than s else s}) \rightarrow \textit{if e than s}}}{((\textit{if e than s})/(\textit{if e than s else s}), \textit{if e than s}) \rightarrow \textit{if e than s}}}{(IF, \textit{if e than s else s}) \rightarrow \textit{if e than s}}$$

As seen in the derivation tree this example would only parse the if statement, because the first choice of the prioritized choice is always parsed first. The effect that a part of the grammar is not reached because the deterministic choice is called grammar hiding.

3.5 Properties

We define a function which calculates the number of steps needed to parse a word. This comes in handy if we want to talk about properties such as parse time.

Definition 3.7. *Assume that $(e, x) \rightarrow o$ is derivable. We define the length of a derivation as the total number of nodes in the derivation tree. We denote this length by $|(e, x)|$, but note that this is only defined if $(e, x) \rightarrow o$ exists for some o. We define the length of a derivation inductively as follows.*

- $|(\varepsilon, x)| = 1$
- $|(a, ax)| = 1,\ a \in V_T$
- $|(b, ax)| = 1,\ a, b \in V_T$
- $|(A, x)| = |(e, x)| + 1$ *with* $A \rightarrow e \in R$
- $$|(e_1e_2, xy)| = \begin{cases} |(e_1, xy)| + 1 & \textit{if } (e_1, xy) \rightarrow fail \\ |(e_1, xy)| + |(e_2, y)| + 1 & \textit{if } (e_1, xy) \rightarrow x \end{cases}$$

- $$|(e_1/e_2, x)| = \begin{cases} |(e_1, x)| + |(e_2, x)| + 1 & \textit{if } (e_1, x) \to fail \\ |(e_1, x)| + 1 & \textit{otherwise} \end{cases}$$
- $$|(e^*, xy)| = \begin{cases} |(e, xy)| + 1 & \textit{if } (e, xy) \to fail \\ |(e, xy)| + |(e^*, y)| + 1 & \textit{if } (e, xy) \to x \end{cases}$$
- $|(!e, x)| = |(e, x)| + 1$

A property of pegs is that they only parse a prefix of the input word. If a word with a given parsing expressions gives a result which is not $fail$ it is a prefix of the input word.

Theorem 3.8. *Given parsing expression e, if the parsing expression does not fail on given word $x \in V_T^*$ it results in a prefix of x.*

If $(e, x) \to y$ where $y \neq fail$, then $x = yz$ for some $z \in V_T^*$

Proof. We will prove this by using induction on the the derivation, distinguishing cases according to the last derivation rule used.

Base cases are the cases which are single rules. We only have three base rules.

$$(\varepsilon, x) \to \varepsilon,\ x = \varepsilon z \text{ for } z = x$$
$$(a, ax) \to a,\ ax = az \text{ for } z = x$$
$$(b, ax) \to fail, \text{ correct since the assumption fails to hold.}$$

Now we apply case distinction on the last rule used. We assume that for every rule the proposition holds, in all but the last derivation rule used, this is our induction hypothesis. Now we prove for every rule that if the rule is last used our proposition holds.

$(e, x) \to y$ there exists a z such that $x = yz$, we proof this for every rule in our syntax:

1. $(A, x) \to o$, $(A, e) \in R$

   According to the rules we get this derivation tree.

   $$\frac{(e, x) \to o}{(A, x) \to o}$$

   In the derivation $(e, x) \to o$ we can apply our induction hypothesis, so if $o \neq fail$ we know $x = oz$ for some $z \in V_T^*$

2. Concatenation(success)

   According to the rules we get this derivation tree.

$$\frac{(e_1, x_1x_2y) \to x_1 \qquad (e_2, y) \to x_2}{(e_1e_2, x_1x_2y) \to x_1x_2}$$

By the definition of this rule our proposition holds since $x_1x_2y = x_1x_2z$ for some $z \in V_T^*$, that is $z = y$

3. Concatenation(failure 1)

   By construction e is formed as $e = e_1e_2$. We get this derivation if this is the last rule used:

   $$\frac{(e_1, x) \to fail}{(e_1e_2, x) \to fail}$$

   Since this derivation results in $fail$, our proposition holds.

4. Concatenation(failure 2)

   Using this as last rule we get this derivation

   $$\frac{(e_1e_2, xy) \to x \qquad (e_2, y) \to fail}{(e_1e_1, xy) \to fail}$$

   Since the assumption fails, our proposition holds.

5. Prioritized choice(case 1)

   $$\frac{(e_1, xy) \to x}{(e_1/e_2, xy) \to x}$$

   By construction, our proposition holds.

6. Prioritized choice(case 2)

   $$\frac{(e_1, x) \to fail \qquad (e_2, x) \to o}{(e_1/e_2, x) \to o}$$

   By the induction hypothesis we know that $x = oz$ so our proposition holds.

7. Repetition(base case):

   $$\frac{(e, x) \to fail}{(e^*, x) \to \varepsilon}$$

   Our proposition holds since ε is always a prefix $x = \varepsilon z$ for $z = x$.

8. Repetition(recursive case):

$$\frac{(e, x_1x_2y) \to x_1 \qquad (e^*, x_2y) \to x_2}{(e^*, x_1x_2y) \to x_1x_2}$$

   Our proposition holds by construction. $x_1x_2y = x_1x_2z$ for $z = y$.

9. Not-predicate(success):

$$\frac{(e, x) \to fail}{(!e, x) \to \varepsilon}$$

   By construction is ε a prefix of x.

10. Not-predicate(failure):

$$\frac{(e, xy) \to x}{(!e, xy) \to fail}$$

   This results in fail so our proposition holds.

   We have proven that for every rule, if it is the last rule applied and the property holds for its precedents the proposition hold. We have also proven all base cases, so we have proven our theorem. That is: if a derivation is successful, the result is a prefix of its input word, $(e, x) \to y$ if $y \neq fail$, that $x = yz$ for some $z \in V_T^*$.

□

Another property we can prove about parsing expressions is the $*$-loop condition. The derivation of (e^*, x) is impossible if $(e, x) \to \varepsilon$.

Theorem 3.9. *For any $x \in V_T^*$, if $(e, x) \to \varepsilon$, then there exists no derivation $(e^*, x) \to o$, that means that for any o: $(e^*, x) \nrightarrow o$.*

Proof. We can try to construct the following derivation tree, and we see it will loop until infinity.

$$\dfrac{(e, xy) \to \varepsilon \qquad \dfrac{(e, xy) \to \varepsilon \qquad \dfrac{(e, xy) \to \varepsilon \qquad \dfrac{\dots}{(e^*, xy) \to}}{\dots}}{(e^*, xy) \to}}{(e^*, xy) \to}$$

Formally we would prove this by assuming there is such a derivation tree. so assume the following derivation is valid:

$$\frac{(e, xy) \to \varepsilon \qquad (e^*, xy) \to o}{(e^*, xy) \to o}$$

We can show a contradiction by calculating the derivation length of this derivation.

$$|(e^*, xy)| = |(e, xy)| + |(e^*, xy)| + 1$$
$$|(e, xy)| = -1$$

Derivation length can only be a positive number, hence the derivation of (e^*, xy) can't exist, since this would result in a contradiction. $\square$

Example 3.10. *Define PEG $G = (\{A\}, \{a, b\}, \{A \to Ae\}, A)$. Where e is an arbitrary parsing expression. Note that a derivation of $(A, x) \to o$ would also be impossible on a given input x since it would look like this.*

$$\cfrac{\cfrac{\cfrac{\cdots}{(Aee, x) \to \cdots}}{(Ae, x) \to \cdots}}{(A, x) \to \cdots}$$

As a corollary in addition to the *-loop condition parsing expressions that have rules in the form of $A \to Ae$, also can't be parsed since there is no derivation possible.

3.6 Well-formedness

As shown in the above example, not all parsing expressions have a derivation. In this section we introduce the concept of well-formed grammars as in [6]. Removing the parsing expressions that have no derivation, according to the *-loop condition, and all the rules in the form of $A \to Ae$. This way we have a set called the *well-formed* grammars.

Definition 3.11. *We define a set of well-formed parsing expression gramamrs inductively as follows. We denote a grammar is well-formed as $WF_G(e)$, the meaning is that parsing expression e is well-formed under PEG $G = (V_N, V_T, R, e_s)$.*

- $WF_G(\varepsilon)$
- $WF_G(a)$, $a \in V_T$
- $WF_G(A)$ *if* $WF_G(e)$ *where* $A \in V_N$ *and* $A \to e \in R$
- $WF_G(e_1e_2)$ *if* $WF_G(e_1)$ *and if* $(e_1, x) \to \varepsilon$ *implies* $WF_G(e_2)$
- $WF_G(e_1/e_2)$ *if* $WF_G(e_1)$ *and* $WF_G(e_2)$
- $WF_G(e^*)$ *if* $WF_G(e)$ *and* $(e, x) \nrightarrow \varepsilon$ *(the *-loop condition)*
- $WF_G(!e)$ *if* $WF_G(e)$

Example 3.12. *Given this definition we can see the following:*

- *the expression ε^* is not well-formed since $(\varepsilon, x) \to \varepsilon$.*
- *the expression A with $R = \{A \to Aa/\varepsilon\}$ is not well-formed since A is only well-formed if Aa/ε is well-formed and that is only well-formed if A is well-formed, leaving an endless loop.*

Since grammars that are not well-formed have little use, we will restrict our research to well-formed grammars. We see that well-formed grammars are complete.

Definition 3.13. *We call a parsing expression grammar $G = (V_N, V_T, R, e_s)$ complete, if for every input word $x \in V_T^*$ there exists a derivation $(e_s, x) \to o$ and $o \in V_T^* \cup \{fail\}$*

3.7 Expressive power

Parsing expression grammar are capable of parsing languages that are not context-free. In this section we will demonstrate how to construct a parsing expression grammar G so that $\mathcal{L}(G) = \{a^n b^n c^n | n \in \mathbb{N}\}$ which is a well-known example of a language that is not context-free.

Example 3.14. *The set of languages constructed by parsing expression grammars, contain languages that are not context-free. The language $L = \{a^n b^n c^n | n \in \mathbb{N}\}$ is recognized with PEG $G = (\{A, B, S\}, \{a, b, c\}, R, S)$ where R has the following production rules:*

- $A \to aAb/\varepsilon$
- $B \to bBc/\varepsilon$
- $S \to \&(A!(a/b))a^*B$

Given $n \in \mathbb{N}$ you get the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\textit{Shown in Example 3.5}}{(A, a^n b^n c^n) \to a^n b^n} \quad \dfrac{\dfrac{(a, c^n) \to fail \qquad (b, c^n) \to fail}{(a/b, c^n) \to fail}}{(!(a/b), c^n) \to \varepsilon}}{(A!(a/b), a^n b^n c^n) \to a^n b^n}}{!(A!(a/b)), a^n b^n c^n) \to fail}}{!!(A!(a/b)), a^n b^n c^n) \to \varepsilon}}{\&(A!(a/b)), a^n b^n c^n) \to \varepsilon}\,\textit{desugar} \quad \dfrac{\dfrac{\textit{Shown in Example 6.3}}{(a^*, a^n b^n c^n) \to a^n} \quad \dfrac{\textit{Shown in Example 3.5}}{(B, b^n c^n) \to b^n c^n}}{(a^* B, a^n b^n c^n) \to a^n b^n c^n}}{\dfrac{(\&(A!(a/b))a^* B, a^n b^n c^n) \to a^n b^n c^n}{(S, a^n b^n c^n) \to a^n b^n c^n}}$$

3.7.1 Failure behaviour

When constructing these grammars one should be very careful when parts of the expression result in $fail$ or ε. In [6] Ford included a PEG that should accept language $\{a^nb^nc^n | n \in \mathbb{N}\}$, however this had some tricky behaviour with ε results. We will show this in following example

Example 3.15. *Let PEG $G = (\{A, B, D\}, \{a, b, c\}, R, D)$, where R consists of the following rules:*

- $A \to aAb/\varepsilon$
- $B \to bBc/\varepsilon$
- $D \to \&(A!b)a^*B$

Note that this PEG looks pretty similar as in Example 3.14. With the only difference in the start expression being $\&(A\boldsymbol{!b})a^*B$ while our definition was $\&(A\boldsymbol{!(a/b)})a^*B$

This results in the following parsing behavior on the word $a^{n+m}b^nc^n$ given $n, m \in \mathbb{N}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{Interesting failure behaviour}}{(A, a^{n+m}b^nc^n) \to \varepsilon} \qquad \dfrac{(b, a^{n+m}b^nc^n) \to fail}{(!b, a^{n+m}b^nc^n) \to \varepsilon}}{(A!(a/b), a^{n+m}b^nc^n) \to \varepsilon}}{!(A!(a/b)), a^{n+m}b^nc^n) \to fail}}{!!(A!(a/b)), a^{n+m}b^nc^n) \to \varepsilon}}{\&(A!(a/b)), a^{n+m}b^nc^n) \to \varepsilon}\,\textit{desugar} \qquad \dfrac{\dfrac{\text{Shown in Example 6.3}}{(a^*, a^{n+m}b^nc^n) \to a^{n+m}} \qquad \dfrac{\text{Shown in Example 3.5}}{(B, b^nc^n) \to b^nc^n}}{(a^*B, a^{n+m}b^nc^n) \to a^{n+m}b^nc^n}}{(\&(A!(a/b))a^*B, a^{n+m}b^nc^n) \to a^{n+m}b^nc^n}}{(D, a^{n+m}b^nc^n) \to a^{n+m}b^nc^n}$$

We will show this interesting failure behaviour with an example derivation on $(e_s, aabc) \to aabc$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{See Example 3.5 with } aab}{(A, aabc) \to \varepsilon} \qquad \dfrac{(b, aabc) \to fail}{(!b, aabc) \to \varepsilon}}{(A!(a/b), aabc) \to \varepsilon}}{!(A!(a/b)), aabc) \to fail}}{!!(A!(a/b)), aabc) \to \varepsilon}}{\&(A!(a/b)), aabc) \to \varepsilon}\,\textit{desugar} \qquad \dfrac{\dfrac{\text{Shown in Example 6.3}}{(a^*, aabc) \to aa} \qquad \dfrac{\text{Shown in Example 3.5}}{(B, bc) \to bc}}{(a^*B, aabc) \to aabc}}{(\&(A!(a/b))a^*B, aabc) \to aabc}}{(D, aabc) \to aabc}$$

Chapter 4

Language definitions

4.1 Ford's definition

When Ford introduced pegs he defined the language of a PEG G as the set of strings $w \in V_T^*$ for which the start expression matches w, that is the derivation does not result in $fail$.

Definition 4.1. *Start expression e_s matches input string w if $(e_s, w) \to x$. Where $x \neq fail$.*

Formally we define this as L_1, the alternative definition of a language given a PEG.

Definition 4.2. *We define the alternative definition of the language of given PEG $G = (V_N, V_T, R, e_s)$ as $L_1(G) = \{w \in V_T^* | (e_s, w) \to_G x \ for \ some \ x \in V_T^*\}$*

Note that in this definition a word is accepted when the PEG does not fail on the word. The word does not necessarily have to be consumed as we defined in our Definition 3.4. This results in some interesting behaviour at the ending of a word. You can use the not-predicate to look ahead and it does not need to consume that part of a word.

In this section we will talk about the equality of our definition and Ford's definitions and see that we can prove their equivalence. At first sight this looks like an easy problem, however this is not as easy as it looks since predicates provide the possibility for a PEG to look ahead in a word.

Theorem 4.3. *Given a PEG $G = (V_N, V_T, R, e_s)$ we can construct $G' = (V_N^{'}, V_T^{'}, R^{'}, e_s^{'})$ such that $L_1(G) = L(G^{'})$*

Proof. Given PEG $G = (V_N, V_T, R, e_s)$ we construct $G' = (V_N, V_T, R, e_s(.*))$. By construction $(e, w) \to_G v \iff (e, w) \to_G' v$ since the start expression e_s has no influence on the relation $\to_G$. we prove $L_1(G) \subseteq L(G')$ and $L(G') \subseteq L_1(G)$ which implies $L_1(G) = L(G')$.

Given $xy \in L_1(G)$ we can make the following derivation tree.

$$\frac{(e_s, xy) \rightarrow_{G'} x \qquad (.^*, y) \rightarrow_{G'} y}{(e_s(.^*), xy) \rightarrow_{G'} xy}$$

- $(e_s, xy) \rightarrow_{G'} x$ holds because we constructed G' to only differ from G in the start expression. This implies that $\rightarrow_G$ and $\rightarrow_{G'}$ act identical. We know $(e_s, xy) \rightarrow_G x$ because we chose $xy \in L_1(G)$.
- $(.^*, y) \rightarrow_{G'} y$ holds by construction, since $y \in v_T^*$.

Thus $xy \in L(G^{'})$, and $L_1(G) \subseteq L(G^{'})$.

Given $x_1x_2 \in L(G^{'})$ we can make an similar looking derivation tree.

$$\frac{(e_s, x_1x_2y) \rightarrow_{G'} x_1 \qquad (.^*, x_2y) \rightarrow_{G'} x_2}{(e_s(.^*), x_1x_2y) \rightarrow_{G'} x_1x_2}$$

We know by construction that $(.^*)$ will consume anything so $y = \varepsilon$. Given that $y = \varepsilon$ we can conclude that $x_1x_2 = x_1x_2y$. With this derivation tree, we know $(x_1x_2y, e_s) \rightarrow_{G'} x$ where x is not a $fail$. This implies that $x_1x_2y \in L_1(G)$ and because $x_1x_2 = x_1x_2y$ this implies $x_1x_2 \in L_1(G) \implies L(G') \subseteq L_1(G)$. □

Theorem 4.4. *Given a PEG $G = (V_N, V_T, R, e_s)$ we can construct $G' = (V_N^{'}, V_T^{'}, R^{'}, e_s^{'})$ such that $L(G) = L_1(G^{'})$*

Proof. Out of the PEG $G = (V_N, V_T, R, e_s)$ we construct $G' = (V_N, V_T, R, e_s(!.))$. Since start expression has no influence on the relation $\rightarrow_G$ and $\rightarrow_{G'}$ we will omit this reference and write the derivation as $\rightarrow$. we prove $L(G) \subseteq L_1(G')$ and $L(G') \subseteq L_1(G)$ which implies $L_1(G) = L(G')$.

Given $w \in L(G)$ by definition we know there exists a derivation such that $(e_s, w) \rightarrow w$. Given this derivation we can construct the derivation under G'.

$$\frac{\dfrac{\text{Holds by construction}}{(e_s, w) \rightarrow w} \qquad \dfrac{(., \varepsilon) \rightarrow fail}{(!., \varepsilon) \rightarrow \varepsilon}}{(e_s(!.), w) \rightarrow w}$$

This means $w \in L_1(G^{'})$, and $L(G) \subseteq L_1(G^{'})$.

Given $x \in L_1(G^{'})$ we know there exists a derivation such that.

$$\frac{(e_s, xy) \rightarrow x \qquad (!., y) \rightarrow \varepsilon}{(e_s(!.), xy) \rightarrow x}$$

We know that $(!., y) \rightarrow \varepsilon$ only holds if and only if $y = \varepsilon$. This makes the derivation $(e_s, xy) = (e_s, x\varepsilon) = (e_s, x) \rightarrow x$ that gives $x \in L(G)$. That rounds up our proof because $L_1(G^{'}) \subseteq L(G)$ and $L(G) \subseteq L_1(G^{'})$ so $L(G) = L_1(G^{'})$ □

Chapter 5

Reducing the grammar

In this chapter we will show how one reduces a parsing expression grammar into more basic form. First we show how one can reduce expression grammars to a form where the repetition operator is eliminated, than we will eliminate the predicate operator.

5.1 Repetition elimination

While easy to read, repetition operators can be eliminated from parsing expression grammars pretty easily. Repetition operators can be eliminated by converting them into recursive non-terminals. Simply rewrite every expression e^* to a new non-terminal A with production rule $A \to eA/\varepsilon$.

Example 5.1. *Given the PEG $G = (\{A, B, S\}, \{a, b, c\}, R, S)$ from Example 3.14 with R consisting of these production rules:*

- $A \to aAb/\varepsilon$
- $B \to bBc/\varepsilon$
- $S \to \&(A!(a/b))a^*B$

We remove the repetition operator by constructing PEG $G' = (\{A, B, S, C\}$ $, \{a, b, c\}, R, S)$, adding the new non-terminal C to eliminate the a^* expression. We add the new production rule for C to the set R, and we replace every occurence of a^* with C. The set of production rules R now consists of these rules.

- $A \to aAb/\varepsilon$
- $B \to bBc/\varepsilon$
- $S \to \&(A!(a/b))CB$
- $C \to aC/\varepsilon$

5.2 Predicate elimination

Predicates are one of the key concepts of parsing expression grammars. However they can be really tricky if you try to prove the equality in language definitions or try to reduce a parsing expression grammar to other formalisms. That's why we will show how one can eliminate predicates from a PEG. Ford showed how this is done and in this section we will show how this process works.

In order to reduce a PEG to a different formalism it is very useful to eliminate both predicate and repetition operators. For this reason we will not use any repetition operators while eliminating predicate operators.

We start to add a non-terminals to help us in the proccess. We add Z to V_N and the rule $Z \to .Z/\varepsilon$. This way Z will match and consume everything left of the input.

Lemma 5.2. *$(Z, w) \to w$ for any $w \in V_T^*$*

Given a expression $!e_1e_2$ one can eliminate the predicate by rewriting the expression to $(e_1Z/\varepsilon)e_2$. For this to work e_2 must not accept ε.

Example 5.3. *Given a PEG $G = (V_N, V_T, R, !e_1e_2)$ and $\varepsilon \notin L(G)$ than $G' = (V_N \cup \{Z\}, V_T, R \cup \{Z \to .Z/\varepsilon\}, (e_1Z/\varepsilon)e_2)$ describes the same language. We show the derivation on input $w \in V_T^*$ given $(e_2, w) \to y$. We will show both the cases $(e_1, x) \to fail$ and $(e_1, x) \not\to fail$.*

In the case that e_1 fails to match on input w, $(e_1, w) \to fail$.

Derivation under G:

$$\dfrac{\dfrac{(e_1, w) \to fail}{(!e_1, w) \to \varepsilon} \qquad (e_2, w) \to y}{(!e_1e_2, w) \to y}$$

Derivation under G':

$$\dfrac{\dfrac{\dfrac{(e_1, w) \to fail}{(e_1Z, w) \to fail} \qquad (\varepsilon, w) \to \varepsilon}{(e_1Z/\varepsilon) \to \varepsilon} \qquad (e_2, w) \to y}{((e_1Z/\varepsilon)e_2, w) \to y}$$

In the case that e_1 does not fail, let $w = xy$, than $(e_1, xy) \to x$. Than the expression $!e_1e_2$ should $fail$.

Derivation under G:

$$\dfrac{\dfrac{(e_1, xy) \to x}{(!e_1, xy) \to fail}}{(!e_1e_2, xy) \to fail}$$

Derivation under G':

$$\dfrac{\dfrac{\dfrac{(e_1,xy)\to x \qquad \dfrac{\mathit{Lemma\ 5.2}}{(Z,y)\to y}}{(e_1Z,xy)\to xy}}{(e_1Z/\varepsilon,xy)\to xy} \qquad (e_2,\varepsilon)\to fail}{((e_1Z/\varepsilon)e_2,xy)\to fail}$$

Note that for this method to work it is important e_2 fails on ε. This is the reason you can't eliminate a predicate out of a PEG that accepts ε.

Example 5.4. *Using the PEG G from Example 3.14, we will construct a repetition- and predicate-free PEG G' which has similar behaviour. Only because the ε limitation we won't be able to parse $a^0b^0c^0$, our language will become $L(G') = \{a^nb^nc^n | n \in \mathbb{N}, n \geq 1\}$. As a first step we will alter the grammar to fail on ε. We get our new PEG $G = (\{A, B, S\}, \{a, b, c\}, R, S)$ where R has the following production rules:*

- $A \to aAb/\varepsilon$
- $B \to bBc/\varepsilon$
- $C \to aC/\varepsilon$
- $S \to !!(Ac)aCB$

We changed non-terminal S changing $A!(a/b)$ to Ac, meaning non-terminal A parses a^1b^1 or more. We also changed C to aC, so it guarantees to parse something. We also desugared the &-operator into two not-predicates. We add the non-terminal Z and corresponding production rule to the PEG and than start altering non-terminal S to eliminate both not-predicates.

1. $S \to !!(Ac)aCB$
2. $S \to (!(Ac)Z/\varepsilon)aCB$
3. $S \to ((AcZ/\varepsilon).Z/\varepsilon)aCB$

Z allows ε so we change it to $.Z$ so it has to parse a minimum of one terminal. The reason we do this is because that part has to trigger a fail if AcZ succeeds.

Chapter 6

Linear time parser

6.1 Parsing

Definition 6.1. *We define a parsing algorithm for any PEG G as any algorithm that given a pair (e, x) computes $P((e, x)) = y$ if $(e, x) \to_G y$. We call the number of computations the algorithm needs, the run-time complexity.*

Ford claimed in his paper that for every PEG one can construct a linear-time parser. In this chapter we will introduce a simple form of parsing and show it is not linear-time. After that we will show how we can use predicate elimination to construct a linear-time parser.

6.2 Intuitive parsing

Definition 6.2. *We define our parsing algorithm $\mathcal{P}$, to simply make the derivation tree according to the derivation rules explained in 3.2. Our number of computations are given by the length of the derivation, defined in Definition 3.7. We have implemented our own version of this parsing algorithm given in Appendix A.*

First we look at how a derivation tree looks when you have the parsing expression $e = a^*$. With the PEG $G = (\emptyset, \{a, b\}, \emptyset, a^*)$. This simply describes the language $\mathcal{L}(G) = \{a^n | n \in \mathbb{N}\}$

Example 6.3. *The derivation tree with a^3 looks like follows. We see $|(a^*, aaa)| = 8$ since the derivation has 8 computation, and we can calculate it using the definition in Definition 3.7.*

$$\dfrac{(a,aaa) \to a \qquad \dfrac{(a,aa) \to a \qquad \dfrac{(a,a) \to a \qquad \dfrac{(a,\varepsilon) \to fail}{(a^*,\varepsilon) \to \varepsilon}}{(a^*,a) \to a}}{(a^*,aa) \to aa}}{(a^*,aaa) \to aaa}$$

We can see that the derivation of (a^*, aa) is right above the derivation of a^3, and so is (a^*, a) above (a^*, aa). So given the derivation of (a^*, aaa) we can construct $(a^*, aaaa)$ as follows:

$$\frac{(a, aaaa) \to a \qquad (a^*, aaa) \to aaa}{(a^*, aaaa) \to aaaa}$$

The derivation length, $|(a^*, a^4)| = |(a^*, a^3)| + |(a, a^4)| + 1 = |(a^*, a^3)| + 2$. As a corollary we can see our derivation length is given as a recursive formula.

Theorem 6.4. *The derivation length of the derivation $(a^*, a^n) \to a^n$ is given by the following formula*

- $|(a^*, a^0)| = 2$
- $|(a^*, a^{n+1})| = |(a^*, a^n)| + 2$

Rewritten into a direct formula it gives $|(a^*, a^n)| = 2 * n + 2$, so our intuitive parser on this PEG is linear, $\mathcal{P}(G)$ runs in linear-time.

Example 6.5. *If we take the concept of Example 6.3 and obfuscate it a bit more we get the parsing expression $e = (!(a^*b)a)$, now e^* is: check if the input word contains b, parse one a and repeat this process.*

PEG $G' = (\emptyset, \{a, b\}, \emptyset, (!(a^*b)a)^*)$, makes our intuitive parser $\mathcal{P}$ run in quadratic time complexity. The derivation of $((!(a^*b)a)^*, a)$ is:

- *$(e^*, \varepsilon) \to \varepsilon$ and the derivation length $|(e^*, \varepsilon)| = 8$*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(a,\varepsilon) \to fail}{(a^*,\varepsilon) \to \varepsilon} \qquad (b,\varepsilon) \to fail}{((a^*b),\varepsilon) \to fail}}{(!(a^*b),\varepsilon) \to \varepsilon} \qquad (a,\varepsilon) \to fail}{(!(a^*b)a,\varepsilon) \to fail}}{(e^*,\varepsilon) \to \varepsilon}$$

- *$(e^*, a) \to a$ and the derivation length $|(e^*, a)| = 5 + |(a^*, a)| + |(e^*, \varepsilon)| = 5 + 4 + 8 = 17$*

$$\dfrac{\dfrac{\dfrac{\dfrac{\textit{see Example 6.3}}{(a^*,a) \to a} \qquad (b,\varepsilon) \to fail}{(!(a^*b),a) \to \varepsilon} \qquad (a,a) \to a}{(e,a) \to a} \qquad \dfrac{\textit{See previous}}{(e^*,\varepsilon) \to \varepsilon}}{(e^*,a) \to a}$$

- *$(e^*, aa) \to aa$ and the derivation length $|(e^*, aa)| = 5 + |(a^*, aa)| + |(e^*, a)| = 5 + 6 + 17 = 28$:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\textit{see Example 6.3}}{(a^*, aa) \to aa} \qquad (b, \varepsilon) \to fail}{(!(a^*b), aa) \to \varepsilon} \qquad (a, aa) \to a}{(e, aa) \to a} \qquad \dfrac{\textit{See previous}}{(e^*, a) \to a}}{(e^*, a) \to a}$$

Note that since the length of the derivation $(a^*, a^n) \to a^n$ increases as n increases. The run time complexity of $(e^*, a^n) \to a^n$ increases more. The formula of the derivation length would be given by:

- $|(e^*, a^0)| = 8$
- $|(e^*, a^{n+1})| = |(e^*, a^n)| + |(a^*, a^{n+1})| + 5$ That is our expression is recursive and the length is the sum of the predecessor, the derivation of a^n and some constant time computations.

We know that $|(a^*, a^n)| = 2n + 2$ as seen in Theorem 6.4.

Theorem 6.6. *The derivation length of the derivation $(e^*, a^n) \to a^n$ is:*

- $|(e^*, a^0)| = 8$
- $|(e^*, a^{n+1})| = |(e^*, a^n)| + 2n + 9$

Rewritting this as a direct formula we get: $|(e^*, a^n)| = n(n + 8) + 8 = n^2 + 8n + 8$

We see that in this example the intuitive parser $\mathcal{P}(G')$ does not parse in linear-time.

Chapter 7

TDPL

7.1 grammar

TDPL is a basic formalism developed by Alexander Birman[1] under the name the TMG Scheme(TS). Later it was named "Top-Down Parsing Language" (TDPL) by Aho and Ullman[2]. In [2] it is shown how one can construct a lineair time parser for a TDPL, so in this chapter we will show you how to reduce a parsing expression grammar to a TDPL.

Definition 7.1. *A TDPL T is a tuple containing these elements:*

- *A finite set of non-terminals V_N*
- *A finite set of terminals V_T with the limitation $V_N \cap V_T = \emptyset$*
- *A finite set of production rules P where all production rules have one of these forms:*
  - $A \to \varepsilon$*, always succeed*
  - $A \to a$*, where* $a \in V_T$
  - $A \to f$*, unconditional fail.*
  - $A \to BC/D$*, where* $B, C, D \in V_N$
- *A non-terminal S representing the start symbol.*

One can already see that a TDPL is a very limited form of a PEG. The / operator in the production rule $A \to BC/D$ has the same behaviour as in parsing expressions.

We will now show how a predicate-free and repetition-free PEG $G = (V_N, V_T, R, e_s)$ can be reduced to a TDPL $T = (V'_N, V_T, R', S)$.

We start by adding non-terminals E, F and S and corresponding rules to PEG G

- $S \rightarrow e_s$, the start expression. Note that this might be illegal syntax for a TDPL but it will be fixed in the next step.
- $F \rightarrow f$, representing failure.
- $E \rightarrow \varepsilon$, representing success.

Now we rewrite every rule A making a distinction on the form of the rule. Read the following rules as A being the rule to be rewriten B, C non-terminals who might be newly added to V_N.

Rule	**New rules**
1. $A \rightarrow B$	$A \rightarrow BE/F$
2. $A \rightarrow e_1e_2$	$A \rightarrow BC/F$
	$B \rightarrow e_1$
	$C \rightarrow e_2$
3. $A \rightarrow e_1/e_2$	$A \rightarrow BE/C$
	$B \rightarrow e_1$
	$C \rightarrow e_2$

Now rewrite all the rules in R which are not in the form of a TDPL-rule, and repeat that process until all the rules are valid TDPL-rules.

Example 7.2. *Recall Example 5.3 Where we constructed a predicate- and repetition-free PEG G which produces the language $L = \{a^n b^n c^n | n \in \mathbb{N}, n \geq 1\}$. We start by adding the rules according the first step now we got PEG $G_1 = (\{A, B, C, S, E, F\}, \{a, b, c\}, R, S)$ with R initialy consiting of:*

- $A \rightarrow aAb/\varepsilon$
- $B \rightarrow bBc/\varepsilon$
- $C \rightarrow aC/\varepsilon$
- $S \rightarrow ((AcZ/\varepsilon).Z/\varepsilon)aCB$
- $Z \rightarrow (a/b/c)Z/\varepsilon$
- $F \rightarrow f$
- $E \rightarrow \varepsilon$

We will now rewrite every rule in order to create a valid TDPL.

Rewritten Rule	***New rules added in*** R
1. $A \rightarrow aAb/\varepsilon$	$A \rightarrow A_1E/A_2$
	$A_1 \rightarrow aAb$
	$A_2 \rightarrow \varepsilon$
2. $A_1 \rightarrow aAb$	$A_1 \rightarrow A_3A_4/F$
	$A_3 \rightarrow a$
	$A_4 \rightarrow Ab$
3. $A_4 \rightarrow Ab$	$A_4 \rightarrow A_5A_6/F$
	$A_5 \rightarrow A$
	$A_6 \rightarrow b$
4. $A_5 \rightarrow A$	$A_5 \rightarrow AE/F$

Now we have successfully rewritten the rules corresponding to non-terminal A to valid TDPL syntax, we repeat this process with every rule left which is not valid. Since the rewriting of rules B,C and Z go in a similar way we leave out intermediate steps.

Rewritten Rule	***New rules added in*** R
1. $B \rightarrow bAc/\varepsilon$	$B \rightarrow B_1E/B_2$
	$B_1 \rightarrow B_3B_4/F$
	$B_2 \rightarrow \varepsilon$
	$B_3 \rightarrow b$
	$B_4 \rightarrow B_5B_6/F$
	$B_5 \rightarrow BE/F$
	$B_6 \rightarrow c$
2. $C \rightarrow aC/\varepsilon$	$C \rightarrow C_1E/C_2$
	$C_1 \rightarrow C_3C_4$
	$C_2 \rightarrow \varepsilon$
	$C_3 \rightarrow a$
	$C_4 \rightarrow CE/F$
3. $Z \rightarrow (a/b/c)Z/\varepsilon$	$Z \rightarrow Z_1E/Z_2$
	$Z_2 \rightarrow \varepsilon$
	$Z_1 \rightarrow Z_3Z/F$
	$Z_3 \rightarrow Z_4E/Z_5$
	$Z_4 \rightarrow a$
	$Z_5 \rightarrow Z_6E/Z_7$
	$Z_6 \rightarrow b$
	$Z_7 \rightarrow c$

While rewriting S looks a bit more complicated since the expression is longer, the idea is the same.

Rewritten Rule	***New rules added in*** R	***Rules not yet valid***
1. $S \to ((AcZ/\varepsilon).Z/\varepsilon)aCB$	$S \to S_1S_2/F$	S_1, S_2
	$S_1 \to (AcZ/\varepsilon).Z/\varepsilon$	
	$S_2 \to aCB$	
2. $S_1 \to (AcZ/\varepsilon).Z/\varepsilon$	$S_1 \to S_3E/S_e$	S_2, S_3
	$S_e \to \varepsilon$	
	$S_3 \to (AcZ/\varepsilon).Z$	
3. $S_2 \to aCB$	$S_2 \to S_4S_5/F$	S_3
	$S_4 \to a$	
	$S_5 \to CB/F$	
4. $S_3 \to (AcZ/\varepsilon).Z$	$S_3 \to S_6S_7/F$	S_6, S_7
	$S_6 \to AcZ/\varepsilon$	
	$S_7 \to .Z \xrightarrow{desugar} (a/(b/c))Z$	
5. $S_7 \to (a/(b/c))Z$	$S_7 \to S_8Z/F$	S_6, S_8
	$S_8 \to (a/(b/c))$	
6. $S_8 \to (a/(b/c))$	$S_8 \to S_aE/S_9$	S_6
	$S_9 \to S_bE/S_c$	
	$S_a \to a$	
	$S_b \to b$	
	$S_c \to c$	
7. $S_6 \to AcZ/\varepsilon$	$S_6 \to S_{10}E/S_e$	S_{10}
	$S_{10} \to AcZ$	
8. $S_{10} \to AcZ$	$S_{10} \to AS_{11}/F$	
	$S_{11} \to S_cZ/F$	

If we combine these new rules we have a valid TDPL T, which parses language $\{a^nb^nc^n | n \in \mathbb{N}, n \geq 1\}$

The TDPL we created in this example can be parsed in linear-time using the tabular parsing technique described in [2]. We will not go into further detail on this, as it falls out of the scope of this thesis.

Chapter 8

Conclusions & related work

8.1 Related work

The definition of the language of a parsing expression grammar given in definition 3.4, is inspired from the definition given by Medeiros et al[8]. We think that our definition and the definition given in [8] are equivalent.

The idea of using a derivation system for PEGs comes is inspired by Koprowski & Binsztok [7].

8.2 Conclusions

Parsing expression grammars are a relatively new way to formally describe languages, and construct parsers. In this paper we explained the way parsing expression grammars work and discussed the semantics.

While discussing the expressive power of parsing expression grammars, we showed that operators which seem to add computational power can in fact be eliminated. We showed this by first reducing a PEG to a form where it has no repetition-operator. After that we showed how one can eliminate the predicate-operator from parsing expressions.

In Chapter 6 we introduced the concept of parsing and showed that linear-time parsing is not a trivial property of parsing expression grammars. We also implemented our own intuitive PEG parser included as Appendix A and computed all examples in this thesis using this implementation. Using this intuitive way of parsing we could construct a PEG that had non-linear time-complexity.

Using the reduced grammars from Chapter 5 we showed how a PEG can be reduced to the minimalistic recognition schema TDPL. Without predicates and repetition this was a fairly straightforward process, but it did greatly increase the number of production rules used. Having a TDPL we can use tabular parsing explained in [2] to create a linear-time complexity parser.

Bibliography

[1] Alexander Birman. *The TMG Recognition Schema*. PhD thesis, Princeton University, 1970.

[2] Jeffrey D. Ullman Alfred V. Aho. *The theory of Parsing, Translation and Compiling - Vol. 1*. Prentice Hall, 1972.

[3] Dennis M. Ritchie Brian W. Kernighan. *The C programming language*. 1988.

[4] Noam Chomsky. *On certain formal properties of grammars*. Information and Control, 1959.

[5] Edwin D. Reilly Daniel D. McCracken. *Encyclopedia of Computer Science*. 2003.

[6] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. 2004. POPL Venice.

[7] Adam Koprowski and Henri Binsztok. Trx: A formally verified parser interpreter. *Programming Languages and Systems*, 2010. Springer.

[8] Roberto Ierusalimschy Sérgio Medeiros, Fabio Mascarenhas. From regexes to parsing expression grammars. 2012.

Appendix A

Intuitive PEG parser

```
def match (e, w):
    if e == "E":
        e=""
    if len(e) == 0:
        return (1,"")
    if len(e) == 1 and e[0] in terminals:
        if len(w) > 0 and e == w[0]:
            return (1, e)
        else:
            return (1, FAIL)
    if len(e) == 1 and e[0] in nonterminals:
        rule = rules[e[0]]
        return match(rule + e[1:], w)
    if len(e) > 1:
        # negation
        if e[0] == "!":
            expressions = split(e[1:])
            result = match(expressions[0], w)
            if result[1] != FAIL:
                return (result[0] + 1, FAIL)
            result1 =  match(expressions[1],w)
            return result1[0] + result[0] + 1, result1[1]
        expressions = split(e)
        if len(expressions[1]) == 0:
            return match(expressions[0],w)
        # alternation
        if expressions[1][0] == "/":
            result = match(expressions[0], w)
            if result[1] != FAIL:
                return result
            result2 = match(expressions[1][1:],w)
            return result2[0] + result[0], result2[1]
        # repetitions
        if expressions[1][0] == "*":
            result = match(expressions[0], w)
            if result[1] == FAIL:
                result2 = match(expressions[1][1:], w)
                if result2[1] == FAIL:
                    return result[0] + result2[0] + 1, FAIL
                return result[0] + result2[0] + 1, result2[1]
            result2 = match("(" + expressions[0] + ")" + expressions[1]
                        , w.replace(result[1],"", 1))
            if result2[1] == FAIL:
                return (result2[0] + result[0] + 1, FAIL)
            return (result2[0] + result[0] + 1, result[1] + result2[1])
        # concat
        result = match(expressions[0], w)
        if result[1] == FAIL:
            return (result[0] + 1, FAIL)
        w = w.replace(result[1], "",1)
        result2 = match(expressions[1], w)
        if result2[1] == FAIL:
            return (result[0] + result2[0] + 1, FAIL)
        else:
            return (result2[0] + result[0] + 1, result[1] + result2[1])
    if len(w) == 0:
        return (1,FAIL)
    else:
        return (1, PARSEFAIL)



#split expressions
def split(e):
    if e[0] == "(":
        expres1 = ""
        count = 0
        for s in e[1:]:
            if s == "(":
                count +=1
            if count == 0 and s == ")":
                return expres1, e.replace("(" + expres1 + ")","",1)
            if s == ")":
                count -= 1
            expres1 += s
    else:
        return e[0], e[1:]
```