

# The computational power of Parsing Expression Grammars

Bruno Loff<sup>a,b,\*</sup>, Nelma Moreira<sup>a,c</sup>, Rogério Reis<sup>a,c</sup>

<sup>a</sup>*DCC, Faculdade de Ciências da Universidade do Porto*

<sup>b</sup>*CRACS, INESC-Tec*

<sup>c</sup>*CMUP, Universidade do Porto*

---

## Abstract

We study the computational power of parsing expression grammars (PEGs). We begin by constructing PEGs with unexpected behaviour, and surprising new examples of languages with PEGs, including the language of palindromes whose length is a power of two, and a binary-counting language.

We then propose a new computational model, the *scaffolding automaton*, and prove that it exactly characterises the computational power of parsing expression grammars (PEGs).

Several consequences will follow from this characterisation: (1) we show that PEGs are computationally “universal”, in a certain sense, which implies the existence of a PEG for a P-complete language; (2) we show that there can be no pumping lemma for PEGs; and (3) we show that PEGs are strictly more powerful than online Turing machines which do  $o(n/(\log n)^2)$  steps of computation per input symbol.

**Keywords:** parsing expression grammar, context-free grammar, pumping lemma, real-time Turing machine, scaffolding automata

**2000 MSC:** 68Q05, 68Q42, 68Q45

---



---

\*Corresponding author

*Email addresses:* [bruno.loff@gmail.com](mailto:bruno.loff@gmail.com) (Bruno Loff), [nam@dcc.fc.up.pt](mailto:nam@dcc.fc.up.pt) (Nelma Moreira), [rvr@dcc.fc.up.pt](mailto:rvr@dcc.fc.up.pt) (Rogério Reis)

This is a revised and expanded version of a paper presented at the 22nd International Conference on Developments in Language Theory (DLT), held in Tokyo, Japan, September 10-14, 2018.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Preliminaries</b>	<b>6</b>
<b>3</b>	<b>Illustrative Examples</b>	<b>9</b>
3.1	Power-Length PEGs . . . . .	9
3.2	PEG for Sometimes-Palindromes . . . . .	10
3.3	PEG for a Counting Language . . . . .	12
<b>4</b>	<b>Scaffolding Automata</b>	<b>14</b>
4.1	Formal Definition . . . . .	14
4.2	Illustrative Examples, Revisited . . . . .	16
4.3	Equivalence with PEGs . . . . .	21
<b>5</b>	<b>Applications</b>	<b>27</b>
5.1	“Universality” . . . . .	27
5.2	Impossibility of a Pumping Lemma . . . . .	29
5.3	PEGs vs. Online Turing Machines . . . . .	32

## 1. Introduction

Parsing expression grammars are a recognition-based system for parsing of formal languages. They were defined by Ford [1], who showed equivalence with earlier parsing systems by Birman and Ullman [2, 3] that are able to recognise the class of *top-down parsing languages* (TDPLs [4]).

As a language formalism, PEGs offer an attractive syntax and an efficient linear-time parsing algorithm which is nonetheless simple to implement. This led to a recent trend, which pushes for the adoption of PEGs, both as a theoretical subject [5, 6, 7, 8, 9, 10, 11, 12], and as a practical tool for parser generators [13, 14, 15, 16, 17, 18, 19, 20, 21]. See Ford’s webpage [22] for an extensive bibliography of work around PEGs.

The influence of PEGs is illustrated by the surprising fact that, despite having been introduced only fifteen years ago, the number of available PEG-based parser generators already seems to nearly-match or even supersede the number of parser generators based on any other single parsing method, even when compared with methods which are many decades older.<sup>2</sup> This seems

---

<sup>2</sup>We estimate this to be true, based on consulting the Wikipedia page “Comparison of

to be due to the simplicity of the formalism, which allows for the quick appearance of many small DIY projects; the situation is reversed if limits one’s attention to high-quality projects, and there does not yet appear to be any serious global tendency to replace older technologies by PEGs. Nonetheless, a few high-quality PEG-based parser generators do exist (e.g. *rats!* [14], or the *Scala Standard Parser-Combinator Library*), and there was at least one serious, influential attempt at creating a programming language which intrinsically relied on PEG as a parsing technology — the *Fortress* programming language [23], which was being developed by Guy Steele’s team at Sun Microsystems. The project is now defunct, but Fortress was once considered as a possible *next-generation* replacement for the *Java* programming language [24]!

Despite this enthusiasm for PEGs, we have also started seeing some objections of a theoretical nature. On one hand, proving the correctness of a given parsing expression grammar is often more difficult than one would like, even for simple examples<sup>3</sup>. This makes PEGs somewhat problematic as a model of formal languages. On the other hand, there is no natural example of a language which is proven not to have PEGs. We believe that the present work will help in understanding why this is the case.

A first naive look at PEGs may suggest that their computational power should be roughly similar to that of deterministic context-free grammars [1]. Indeed it is known that deterministic context-free languages have PEGs [2]. But already Aho and Ullman [4] had shown that the  $a^n b^n c^n$  language, which is not context-free, is still a TDPL, and hence has a PEG [1].

One may still hope that the computational power of PEGs can be contained, in some way, akin to how we can use pumping lemmas to separate the Chomsky hierarchy (e.g. [25, 26, 27, 28, 29]). The following question appears in Aho and Ullman’s book [4], and in Ford’s article [1]:

---

parser generators”, and searching GitHub for “parser generator X”, and then counting how many projects appear which use a given method X. Doing so, one obtains the following numbers (ca. September 2019):

	LR	LL	LALR	GLR	Earley	<b>PEG</b>
Wikipedia	26	33	63	23	7	<b>48</b>
GitHub	62	86	77	10	9	<b>122</b>

<sup>3</sup>For example, the relatively simple grammar for the  $a^n b^n c^n$  language which appears in Ford’s original paper [1], has a (fixable) bug, which eluded discovery for over a decade (including to us, when we read Ford’s paper) until the bug was pointed out by a recent paper of Garnock-Jones et al. [6].

*Is there a context-free language without a parsing expression grammar?*

It is possible to prove that if any such language exists, then Greibach's *hardest context-free language*  $\mathcal{H}$  [30] also has no PEGs. So the above problem is equivalent to asking for a proof that  $\mathcal{H}$  has no parsing expression grammar. But no PEG is known, even for the much simpler language of *palindromes*. The following questions are both open:

*Can a parsing expression grammar recognise the language of palindromes?*

*Is there any linear-time language without a parsing expression grammar?*

In fact, the only method we know to prove that a language has no PEG is by using the time-hierarchy theorem of complexity theory [31]: using diagonalisation one may construct some language  $L_2$  which is decidable, say, in time  $n^2$  (by a random-access machine), but not in linear time, and because PEGs can be recognised in linear time using the tabular parsing algorithm of Birman and Ullman [2] (or packrat parsing [32, 33]), there will be no parsing expression grammar for  $L_2$ .

This stands in stark contrast with our understanding of, say, context-free languages. In that scenario, one may also construct a language  $L_4$  which is decidable in time  $n^4$ , which cannot be decided in time  $n^3$ , and hence  $L_4$  cannot be context-free (since the CYK algorithm decides any context-free language in time  $n^3$ , see, e.g., Hopcroft's book [34]). But this brings us no real insight on what it means to be context-free. To understand this, we make use of *pumping lemmas*, and using such lemmas we can easily provide, say, a linear-time-decidable language which is not context-free. A pumping lemma implies a serious limitation on the computational power of context-free languages, which does not apply to universal models of computation, such as Turing machines or random-access machines.

Our current understanding of universal computation, by contrast, is extremely poor. For example, it is a longstanding open problem, to show that linear-time random-access machines cannot be simulated by two-tape Turing machines in linear time, even though it seems intuitive that this should be true. Indeed this problem is well beyond the current state of the art in computational complexity, where such lower-bounds are notoriously difficult to come by. It is also an open problem to provide any context-free language which cannot be decided by a two-tape Turing machine in linear time — for one-tape Turing machines such a separation is known<sup>4</sup>.

---

<sup>4</sup>This was first proven for palindromes; see Li and Vitanyi [26, §6.1 and §6.13].

A principal claim of this article is that the recognition procedure underlying parsing expression grammars is, in some sense, “universal”, and so it will be as difficult to understand as that of a multi-tape Turing machine. A solution to the above questions, thus, may well require a breakthrough in our ability to prove computational complexity lower-bounds.

With this in mind, the layout of the article is as follows. In Section 2, we provide a formal definition of PEGs, and in Section 3 we show a few examples of PEGs with surprising behaviour, and of languages which, unexpectedly, have PEGs. This includes the language of palindromes whose length is a power of two, and it is also shown that PEGs can do a form of counting.

In Section 4, we describe a new computational model, the *scaffolding automaton*, and show that it exactly characterises the computational power of PEGs. This is our main result, and provides what we believe to be the right machine model for parsing expression grammars. We will make good use of this characterisation in Section 5, where we show the following results.

- We revisit the example languages of Section 3, and construct scaffolding automata for them, for the sake of becoming familiar with the model.
- We show that PEGs are computationally “universal”, in the following sense: take any computable function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ ; then there exists a computable function  $g : \{0, 1\}^* \rightarrow \mathbb{N}$  such that

$$\{f(x) \$^{g(x)} x \mid x \in \{0, 1\}^*\}$$

has a PEG. This result may be used to construct a PEG language which is complete for  $\mathsf{P}$  under logspace reductions. This stands in contrast to context-free languages, which cannot be  $\mathsf{P}$  complete under logspace reductions unless  $\mathsf{P} \subseteq \mathsf{NC}_2$ .

- We show that there can be no pumping lemma for PEGs. There is no total computable function  $A$  with the following property: for every PEG  $G$ , there exists  $n_0$  such that for every string  $x \in \mathcal{L}(G)$  of size  $|x| \geq n_0$ , the output  $y = A(G, x)$  is in  $\mathcal{L}(G)$  and has  $|y| > |x|$ .
- We show that PEGs are strongly non real-time for Turing machines: There exists a language with a PEG, such that neither it nor its reverse can be recognised by any multi-tape online Turing machine which is allowed to do only  $o(n/(\log n)^2)$  steps after reading each input symbol.

## 2. Preliminaries

In this section we will cover some notation, and give a formal definition of parsing expression grammars.

*Notation..* For each  $k \in \mathbb{N}$ , let  $(k)_2 \in \{0, 1\}^*$  be its shortest binary representation, and  $(k)_2^r$  to denote the reversal of its shortest binary representation. An *alphabet*  $\Gamma$  is a finite set of symbols such that  $\emptyset \notin \Gamma$ . For a natural number  $n \geq 0$ , we denote  $[n] = \{0, \dots, n\}$ ,  $[n) = \{0, \dots, n - 1\}$ , and  $(n] = \{1, \dots, n\}$ . We will use  $\lambda$  to denote the empty word, and  $\varepsilon$  to denote a parsing expression which accepts the empty word.

**Definition 1.** Let  $\Sigma, \text{NT}$  be two disjoint alphabets; the symbols in  $\Sigma$  are called *terminal* symbols, and those in  $\text{NT}$  are called *non-terminal* symbols. Then, the set  $\mathcal{E}(\Sigma, \text{NT})$  of *parsing-expressions over  $\Sigma$  and  $\text{NT}$*  is defined inductively.

- At the base of the induction we have  $\Sigma \cup \text{NT} \cup \{\varepsilon, \text{FAIL}\} \subseteq \mathcal{E}(\Sigma, \text{NT})$ .
- If  $e \in \mathcal{E}(\Sigma, \text{NT})$ , we will have  $!e$  and  $\&e$  in  $\mathcal{E}(\Sigma, \text{NT})$ .
- If  $e_1, e_2 \in \mathcal{E}(\Sigma, \text{NT})$ , we will have  $e_1e_2$  and  $e_1/e_2$  in  $\mathcal{E}(\Sigma, \text{NT})$ .

**Definition 2.** A *parsing expression grammar*  $\mathcal{G}$  is a tuple  $\langle \Sigma, \text{NT}, R, S \rangle$ , where

- $\Sigma$  is an alphabet of so-called *terminal symbols*.
- $\text{NT}$  is an alphabet of so-called *non-terminal symbols*, disjoint from  $\Sigma$ .
- $R : \text{NT} \rightarrow \mathcal{E}(\Sigma, \text{NT})$  is a function defining the *rules* of  $\mathcal{G}$ , and associates a  $(\Sigma, \text{NT})$ -parsing-expression to each non-terminal symbol.
- $S \in \text{NT}$  is the *starting non-terminal*.

When writing down a parsing expression grammar, the notation  $A \leftarrow e$  is used to signify  $R(A) = e$ . The reason one uses the left arrow notation is to emphasise that PEGs correspond to a *recognition procedure*, and are not to be thought of as a generative model.

Ford [1] defines parsing expressions that allow for various operations, such as the *zero-or-more repetitions* operator “ $*$ ”, or the *any character* symbol “ $.$ ”. As explained in Ford’s paper [1], these operators can be expressed by

using the operators appearing in Definition 1, together with the grammars of Definition 2. This is similar to how one would define such operators using context-free grammars, so we will not explicitly include these operators as part of Definition 1. For the sake of example, the *zero-or-more repetitions* operator  $A^*$ , applied to a non-terminal  $A$ , may be replaced by a new non-terminal  $A_{\text{star}}$  together with the rule  $A_{\text{star}} \leftarrow A A_{\text{star}} / \varepsilon$ .

☞ The *any character* symbol “.”, which we will be using extensively throughout, may be replaced with  $(a/b/\dots)$  for each terminal symbol  $a, b, \dots$  of  $\Sigma$ . After we define the recognition procedure underlying a parsing expression grammar, in Definition 3 below, it may be seen that the parsing expression “!” recognizes exactly the empty string at the end of the input.

In order to define a rule  $A \leftarrow B/C/\dots$ , we will write rules of the form  $A \leftarrow B$ ,  $A \leftarrow C$ , *etc*, and say they are *alternatives* of the non-terminal symbol  $A$ . So, for example, if we say  $A \leftarrow BA$  and  $A \leftarrow \varepsilon$  are alternatives of  $A$ , we mean that the rule for  $A$  is  $R(A) = BA / \varepsilon$ . We will only do this when the order in which the alternatives appear in the rule is indifferent.

Each parsing expression grammar defines an associated recognition procedure. This procedure gives an operational meaning to each PEG.

**Definition 3** (Recognition). Let  $\mathcal{G} = \langle \Sigma, \text{NT}, R, S \rangle$  be a parsing expression grammar. The *recognition map* is a partial function

$$\text{Rec}_{\mathcal{G}} : \mathcal{E}(\Sigma, \text{NT}) \times \Sigma^* \rightarrow \Sigma^* \cup \{\text{FAIL}\};$$

this map is defined by Algorithm 1 appearing below. If  $\text{Rec}_{\mathcal{G}}(e, x) = \text{FAIL}$ , we say that expression  $e$  *rejects* input  $x$ ; and if  $\text{Rec}_{\mathcal{G}}(e, x) = x'$  outputs a prefix  $x'$  of  $x$ , we say that expression  $e$  *accepts*  $x$ , and *consumes*  $x'$ . If  $\text{Rec}_{\mathcal{G}}(e, x) = x$ , i.e.  $e$  accepts  $x$  and consumes all of  $x$ , then we say the expression  $e$  *recognises*  $x$ . Otherwise  $\text{Rec}_{\mathcal{G}}(e, x)$  is *undefined*, which happens precisely when the recognition procedure entered an infinite loop. We say that  $\mathcal{G}$  is *total* if its recognition map is total, i.e. if it never enters an infinite loop, on any input.

☞ The notions *rejects*, *accepts*, *consumes* and *recognises* will be frequently used throughout the paper, and the reader may refer to the above definition to remember what they mean. It is important to understand that a parsing expression  $e$  may accept a string  $x$ , without consuming all of it. For example the expression  $\&(aa)$  accepts the string  $aa$  but consumes no symbol in it.

---

**Algorithm 1** Recognition Procedure  $\text{Rec}_{\mathcal{G}}(E, x)$  :

---

**Input:**  $E \in \mathcal{E}(\Sigma, \text{NT}), x \in \Sigma^*$ **Output:**  $\text{Rec}_{\mathcal{G}}(E, x) \in \Sigma^* \cup \{\text{FAIL}\}$ 

```
1: if  $E = \varepsilon$  then return the empty string  $\lambda$ 
2: else if  $E = \text{FAIL}$  then return FAIL
3: else if  $E = a \in \Sigma$  then
4:   if  $x = az$  for some  $z$  then return  $a$  else return FAIL
5: else if  $E = !e$  then
6:   if  $\text{Rec}_{\mathcal{G}}(e, x) = \text{FAIL}$  then return  $\lambda$  else return FAIL
7: else if  $E = \&e$  then
8:   if  $\text{Rec}_{\mathcal{G}}(e, x) \in \Sigma^*$  then return  $\lambda$  else return FAIL
9: else if  $E = e_1e_2$  then
10:  if  $\text{Rec}_{\mathcal{G}}(e_1, x) = y_1 \in \Sigma^*$  and  $x = y_1z$  and  $\text{Rec}_{\mathcal{G}}(e_2, z) = y_2 \in \Sigma^*$  then
11:    return  $y_1y_2$ 
12:  else return FAIL
13: else if  $E = e_1/e_2$  then
14:  if  $\text{Rec}_{\mathcal{G}}(e_1, x) \in \Sigma^*$  then return  $\text{Rec}_{\mathcal{G}}(e_1, x)$ 
15:  else return  $\text{Rec}_{\mathcal{G}}(e_2, x)$ 
16: else if  $E = A \in \text{NT}$  then return  $\text{Rec}_{\mathcal{G}}(R(A), x)$ 
```

---

**Definition 4.** A total PEG  $\mathcal{G} = \langle \Sigma, \text{NT}, R, S \rangle$  is said to *recognise* the language  $\mathcal{L}(\mathcal{G}) = \{x \in \Sigma^* \mid \text{Rec}_{\mathcal{G}}(S, x) = x\}$ .

Then PEG is the class of languages recognised by total PEGs.

One consequence of the results in this paper is that no algorithm can decide whether a PEG is total. Ford's original paper [1] defined a notion, that of *well-formed* parsing expression grammar, which was inherited from Birman and Ullman [2]. A well-formed PEG is a PEG which obeys a certain syntactic restriction; this restriction guarantees that the above recognition procedure will not enter an infinite loop (but not all total PEGs are well-formed).

Informally, a PEG is well-formed if it avoids left recursion. To avoid excessive formalism, in this paper we will not concern ourselves with the formal definition of well-formed PEGs. All the PEGs appearing in this paper are total, and, for the readers familiar with the notion of well-formedness, it will be possible to see that they are also well-formed. Furthermore, every theorem in this paper referring to “total” PEGs will still hold if one restricts our attention to “well-formed” PEGs.

Furthermore, there is an algorithm which accepts a PEG  $\mathcal{G}$  as input, and outputs a well-formed PEG  $\mathcal{G}'$ , such that  $\mathcal{G}'$  recognises the same language as  $\mathcal{G}$  whenever  $\mathcal{G}$  is total. This is akin to the fact that, despite it being



undecidable if a given Turing machine runs in time  $n^2$ , one can take any Turing machine  $\mathcal{M}$  and convert it into a (multitape) Turing machine  $\mathcal{M}'$  which does run in time  $n^2$ , and which decides the same language as  $\mathcal{M}$  if  $\mathcal{M}$  also runs in time  $n^2$  [see 35, 36].

### 3. Illustrative Examples

In this section we will study some examples which were instrumental for us to understand the computational power of the model.

#### 3.1. Power-Length PEGs

Our initial expectations for the computational power of PEGs were that we should be able to treat them in a similar way as with context-free grammars, by showing a pumping lemma for them.

This owed not so much to what we knew about the computational power of PEGs — which already Birman and Ullman [2], and Ford [1], had shown surpasses that of CFGs — but rather to the context in which one studies PEGs: *if PEGs are regarded in the context of formal languages, then we should be able to prove some kind of pumping lemma*. But soon we stumbled on the following example from the PhD thesis of Birman [3]:

**Theorem 5.** The unary language of words whose length is a power-of-2

$$\mathcal{P}_2 = \{a^{2^n} \mid n \geq 0\}$$

is in PEG.

How does this relate to pumping lemmas? The known pumping lemmas are able to produce, given a sufficiently large string  $x$  in the language, a strictly larger string  $y$ , also in the language, which is not much larger —  $|y| \leq |x| + O(1)$  is sufficient. But here is a language with a PEG, for which  $|y|$  is always at least  $2|x|$ . And soon after conjecturing that  $c \cdot |x|$  might be sufficient, for some universal constant  $c$ , one is disabused of that notion by the following generalisation of the above:

**Theorem 6.** For every  $\ell \in \mathbb{N}$ , the language  $\mathcal{P}_\ell = \{a^{\ell^n} \mid n \geq 0\}$  is in PEG.

*Proof.* Consider the following parsing expression grammar  $\mathcal{G}$ :

$$\begin{aligned} \text{IAmPowerLength} &\leftarrow a! \quad / \quad \text{Helper} !. \\ \text{Helper} &\leftarrow a^{\ell-1} \text{Helper } a \quad / \quad a^{\ell-1}(\&\text{Helper})a \quad / \quad a((! \text{Helper})a)^{\ell-1} \end{aligned}$$

Let us analyse the behaviour of the recognition procedure  $\text{Rec}_G(\text{Helper}, x)$  for each  $x \in \{a\}^*$ . The shortest  $x$  to be accepted will be  $a^\ell$ ; this string is accepted via the third alternative of the **Helper** non-terminal, and every symbol will be consumed, so  $a^\ell$  is recognised by **Helper**. Then the second string to be accepted will be  $a^{\ell-1}aa^{\ell-1}$ , via the second alternative — the first alternative must have failed because it won't find the last  $a$ . So the second alternative is triggered, but only the first  $\ell$ -many  $a$  symbols will be consumed, leaving  $a^{\ell-1}$  symbols unconsumed (hence the string will be “accepted”, but it won't be “recognised”). Then the first alternative will trigger for each new sequence of  $\ell - 1$   $a$ s, each time consuming a new  $a$  symbol closer to the end of the input. Hence at this point in total we will have consumed  $(\ell - 1)\ell$  new symbols, which together with the  $\ell$  symbols give us  $\ell^2$  consumed symbols, and at this point the non-terminal **Helper** will have consumed the entire input. Thus  $a^{\ell^2}$  is accepted by **Helper**. Then again the second alternative is triggered, and then the first, until  $\ell^3$  symbols are consumed.

In the end, we conclude that **Helper** accepts any string of the form

$$a^{s(\ell-1)}a^s z,$$

where the first position of the  $a^s$ -part is the first position at a power-of- $\ell$  distance from the end of the input, and in this case it consumes the first  $s\ell$ -many  $a$  symbols.  $\square$

### 3.2. PEG for Sometimes-Palindromes

One may get a sense for the limitations of parsing expression grammars when trying to produce a PEG for recognising palindromes. One quickly comes to the conjecture that PEGs cannot find the middle bit of the input. In the case of palindromes, we make the following conjecture:

**Conjecture 7.** The language of even-length palindromes has no PEG, i.e.

$$P = \{ww^r \mid w \in \{0,1\}^*\} \notin \text{PEG}.$$

However, the above PEG for  $\mathcal{P}_2$  is able to find the middle bit of every string whose length is a power of two. This allows us to prove the following result:

**Theorem 8.** The language of palindromes of power-of-two length has a PEG:

$$SP = \{ww^r \mid w \in \{0,1\}^{2^n}, n \geq 0\} \in \text{PEG}.$$

*Proof.* The following parsing expression grammar will do:

$$S \leftarrow \&(\text{IAmPowerTwoLength}) \text{ Palindrome}$$

$$\text{Palindrome} \leftarrow P! . / 00! . / 11! .$$

$$\begin{aligned} P \leftarrow & 0 !(\text{IAmPowerTwoLength}) P 0 \\ & / 1 !(\text{IAmPowerTwoLength}) P 1 \\ & / 1 \&(\text{IAmPowerTwoLength}) 1 \\ & / 0 \&(\text{IAmPowerTwoLength}) 0 \end{aligned}$$

$$\text{IAmPowerTwoLength} \leftarrow \text{Helper} ! .$$

$$\text{Helper} \leftarrow \text{Bit Helper Bit} \quad / \quad \text{Bit Bit}$$

$$\text{Bit} \leftarrow 0/1$$

As in the proof of Theorem 6, the non-terminal `IAmPowerTwoLength` accepts exactly at the positions whose distance from the end-of-input is a positive power of two, and consumes the entire input in that case. Hence the expression `(&IAmPowerTwoLength)` accepts exactly at positions whose distance from end-of-input is a positive power of two, and when it accepts it will not consume any input. On the other hand the expression `(!IAmPowerTwoLength)` accepts exactly at positions which are *not* at positive-power-of-two distance away from the end-of-input.

The recognition procedure associated with the non-terminal `P` now behaves as follows: one of the first two alternatives will be chosen repeatedly, until the first position which is a positive power-of-two is reached; then, at that position, one of the last two alternatives is chosen. (In each case, which of the two alternatives gets chosen is determined by the next bit.) It follows that `P` accepts exactly at those positions  $i$  such that the input after (and including) position  $i$  is of the form:

$$x y z$$

where  $x = y^r$ , and the leftmost position after  $i$  which is at a positive-power-of-two distance away from the end-of-input, is the first bit of  $y$ . And when `P` accepts such a string  $xyz$ , `P` consumes exactly the prefix  $xy$ .

Inspection of the rules for `Palindrome` and `S` concludes the proof.  $\square$

### 3.3. PEG for a Counting Language

The next example will be crucial in Sections 5.1 and 5.3, for reasons which we will explain in Section 4.2.

**Theorem 9.** The following *reversed counting language*, over the alphabet  $\{0, 1, \#, \circ\}$ , has a parsing expression grammar:

$$\{(n)_2^r \circ (n)_2 \# (n-1)_2^r \circ (n-1)_2 \# \cdots \# (0)_2^r \circ (0)_2 \# \mid n \geq 0\}.$$

The characters  $\#$  and  $\circ$  are part of the input alphabet, and are being used as separators, with no other special meaning. We will call  $\#$  the *outer separator*, and  $\circ$  the *inner separator*.

*Proof.* The proof relies on the intuition built in the previous two proofs. Roughly speaking, it implements the simple increment-by-one algorithm.

Let us begin by presenting only part of the grammar. We will omit the rules associated with the non-terminal `AddOneBlock`, for now. The grammar begins with the rules:

$$\text{Sequence} \leftarrow \&(\text{AddOneBlock}) \text{ InvertedBlock Sequence} \quad / \quad 0 \circ 0 \#$$

$$\text{InvertedBlock} \leftarrow \text{Inverted} \#$$

$$\text{Inverted} \leftarrow 1 \text{ Inverted } 1 \quad / \quad 0 \text{ Inverted } 0 \quad / \quad \circ$$

The first thing to notice is that `InvertedBlock` recognises exactly “inverted blocks” of the form  $w^r \circ w \#$ , where  $w \in \{0, 1\}^*$ . Thus the inputs recognised by `Sequence` are exactly sequences of inverted blocks which additionally are accepted by the `AddOneBlock` non-terminal; the rules for this non-terminal are:

$$\text{AddOneBlock} \leftarrow \text{Bit}^+ \circ \text{AddOneCheck}$$

$$\text{AddOneCheck} \leftarrow \text{AddOneDigit AddOneCheck} \quad / \quad \#$$

Now `AddOneBlock` accepts strings of the form  $x \circ y \#$ , such that  $x \in \{0, 1\}^*$ , and such that `AddOneDigit` accepts the input at every position of  $y$ . This will be defined in such a way that, at the  $i$ -th bit of  $y$  (starting from the right), `AddOneDigit` will accept if and only if the  $i$ -th bit of  $(n+1)_2$  is  $y_i$ , where  $n$  is the number encoded in the following block (i.e. after the  $\#$ ).

To enforce this behaviour, we use the following rules:

$$\begin{aligned} \text{AddOneDigit} \leftarrow & \text{\&Nextls1 \&Carry 0} \\ & / \text{\&Nextls0 \&Carry 1} \\ & / \text{\&Nextls1 !Carry 1} \\ & / \text{\&Nextls0 !Carry 0} \\ & / \text{\&NextlsCircle \&Carry 1} \end{aligned}$$

$$\text{Carry} \leftarrow . \text{\&Nextls1 \&Carry} \quad / \quad \text{Bit \#}$$

The non-terminals `Nextls0`, `Nextls1`, and `NextlsCircle` will verify that the input symbol in the corresponding position in the next block is a 0, a 1 or a  $\circ$ , respectively. So, for example, if the input after the current position is

$$y_i y_{i+1} \cdots y_k \# x_k \cdots x_{i-1} x_i,$$

then `Nextls0` will accept iff  $x_i = 0$ , `Nextls1` will accept iff  $x_i = 1$ , and `NextlsCircle` will accept iff  $x_i = \circ$ .

It results from this that the non-terminal `Carry` accepts if and only if there is a carry at the current position, when we add 1 to the number after the `#` separator: we implement the incremented 1 by setting the carry to 1 at the least significant bit, and then the carry propagates as long as the number after the separator has a 1. Then `AddOneDigit` successfully checks a single digit in the increment, in the usual way: a 1 and a carry sum to 0, a 0 and a carry sum to 1, *etcetera*.

All we are left to do is defining the auxiliary non-terminals:

$$\text{Nextls0} \leftarrow \text{Bit SameLength 0}$$

$$\text{Nextls1} \leftarrow \text{Bit SameLength 1}$$

$$\text{NextlsCircle} \leftarrow \text{Bit SameLength } \circ$$

$$\text{SameLength} \leftarrow \text{Bit SameLength Bit} \quad / \quad \#$$

$$\text{Bit}^+ \leftarrow \text{Bit Bit}^+ / \text{Bit}$$

$$\text{Bit} \leftarrow 0 / 1 \quad \square$$

Let us here make an important remark. The simple increment-by-one algorithm works by scanning the bits from right to left. However it does not appear to be possible to implement such a right-to-left scanning using PEGs, but left-to-right scanning can be done, and this is what the `Nextls*`

non-terminals are doing, and checking inversion is possible, as shown by the **Inverted** non-terminal. So we may implement right-to-left scanning by inverting at each block and then using left-to-right scanning. This trick will be called “reverse and scan”, and will be used in our simulation of Turing machines by PEGs (in Section 5.1), as well as in our construction of a non-real-time PEG language (in Section 5.3).

*Conclusion.* While carefully considering the examples above, one will get a sense that the computational power of PEGs is much greater than it seems at first glance. When considering why and how these examples work, one is slowly drawn to a generalisation of the above: a computational model for languages recognised by parsing expression grammars. This is what we present in the next section.

#### 4. Scaffolding Automata

Let us begin by giving an informal description of a scaffolding automaton. Such an automaton is a computing machine which constructs a labelled, directed, acyclic graph of bounded degree, which we call a *scaffold*. At the start of the computation, the graph is a single node with a special end-marker label; this is the *base* of the scaffold. Then as the computation proceeds new input symbols are read and new nodes are added; the node which was last added is called the *top* of the scaffold. At each step of computation, the scaffolding automaton sees a new input symbol, and is allowed to look at a finite-distance neighbourhood of the top; based on the edges which are present, on the labels it sees, on the input symbol it just read, and on the current state of its finite control, the automaton adds a new node to the scaffold (the new top), and chooses the edges of this new node to point to some nodes in the finite-distance neighbourhood it has just observed. This is repeated until all input symbols are read.

##### 4.1. Formal Definition

**Definition 10** (Scaffold). Let  $d \geq 1$ ,  $t \geq 0$  be natural numbers, and let  $\Gamma$  be an alphabet. An *edge list* of degree  $d$  is a tuple

$$e = (e(0), \dots, e(d-1)) \in (\mathbb{N} \cup \{\emptyset\})^d.$$

A  $(d, \Gamma)$ -*scaffold* of size  $t+1 \in \mathbb{N}$  is a labelled multidigraph  $S = (V, E, L)$  with set of nodes  $V = [t]$ , a set of edge lists  $E = \{ e_v \in (\mathbb{N} \cup \{\emptyset\})^d \mid v \in [t] \}$ , where

$$\forall v \in [t] \forall i \in [d] \quad e_v(i) \in [v] \cup \{\emptyset\}, \quad (\text{“edges point backwards”})$$

and a labelling function  $L : V \rightarrow \Gamma \cup \{\emptyset\}$ .

We call  $t$  the *top* of the scaffold  $S$ . If  $e_v(i) = \emptyset$ , one says that *node  $v$  is missing edge  $i$* , otherwise we say that *edge  $i$  is present at node  $v$* . If  $L(v) = \emptyset$ , one says  *$v$  is unlabelled*. Let  $\mathbb{S}(d, \Gamma)$  be set of all  $(d, \Gamma)$ -scaffolds (of any length).

Given a tuple  $p \in [d]^k$ , and a node  $v \in V$  in a  $(d, \Gamma)$ -scaffold  $S = (V, E, L)$ , we may inductively define the sequence

$$v_0 = v \text{ and } v_{j+1} = \begin{cases} e_{v_j}(p_j) & \text{if } v_j \in V, \\ \emptyset & \text{if } v_j = \emptyset. \end{cases}$$

If this sequence has  $v_i = \emptyset$  for some  $i \in [k]$ , we say  $p$  is an *invalid path from  $v$  in  $S$* . Otherwise we say  $p$  is a (valid) *path from  $v$  to  $v_k$  in  $S$* .

**Definition 11** (Neighbourhood). Given  $S = (V, E, L) \in \mathbb{S}(d, \Gamma)$ ,  $k \geq 0$  and  $v \in V$ , the  *$k$ -neighbourhood of  $v$  in  $S$* ,  $N_k(S, v)$ , is given inductively by  $N_0(S, v) = L(v)$  and  $N_{k+1}(S, v) = (L(v), N_k(S, e_v(0)), \dots, N_k(S, e_v(d-1)))$ , where we set  $N_k(S, \emptyset) = \emptyset$ .

The set of  *$k$ -neighbourhoods for  $(d, \Gamma)$ -scaffolds*,  $\mathcal{N}_k(d, \Gamma)$ , is the set of partial,  $d$ -ary,  $\Gamma$ -labelled trees. It may be inductively defined by letting  $\mathcal{N}_0(d, \Gamma) = \Gamma \cup \{\emptyset\}$  and  $\mathcal{N}_{k+1}(d, \Gamma) = (\Gamma \cup \{\emptyset\}) \times (\mathcal{N}_k(d, \Gamma) \cup \{\emptyset\})^d$ .

**Definition 12** (Scaffolding automaton). A *scaffolding automaton*  $\mathcal{A}$  is a tuple  $\mathcal{A} = \langle \Sigma, d, \Gamma, k, Q, \delta, q_0, F \rangle$ , where,

- $\Sigma$  is an alphabet, called the *input alphabet*,
- $d \geq 1, k \geq 0$  are natural numbers, called *degree* and *distance*, respectively,
- $\Gamma$  is an alphabet, called the *working alphabet*,
- $Q$  is a finite set of *states*,
- $q_0 \in Q$  is the *initial state*,
- $F \subseteq Q$  gives the *accepting states*, and
- the *transition function* is of type

$$\delta : Q \times \Sigma \times \mathcal{N}_k(d, \Gamma) \rightarrow Q \times \Gamma \times ([d]^{\leq k} \cup \{\text{SELF}, \emptyset\})^d.$$

A scaffolding automaton builds a scaffold while reading the input. The initial scaffold is  $S_0 = (\{0\}, \{\}, L)$  where  $L(0) = \emptyset$ . The transition function  $\delta$  transforms a scaffold as follows.

**Definition 13** (Single step of computation). Let  $S = ([t], E, L) \in \mathbb{S}(d, \Gamma)$ , and  $\delta$  be a transition function. For some  $q \in Q$  and  $\sigma \in \Sigma$ , let

$$(q', \gamma, p_0, \dots, p_{d-1}) = \delta(q, \sigma, N_k(S, t)).$$

The *single-step function* is then given by  $\text{Step}_{\delta, \sigma}(q, S) = (q', S')$ , where  $S' = ([t+1], E', L') \in \mathbb{S}(d, \Gamma)$ , with  $L'(t+1) = \gamma$ ,  $L'(v) = L(v)$  for  $v \in [t]$ , and  $E' = E \cup \{e_{t+1}\}$ , for the edge list  $e_{t+1} = (v_0, \dots, v_{d-1})$ , where  $v_i$  is obtained by following path  $p_i$  from  $t$  in  $S$  (and equals  $\emptyset$  if  $p_i$  is an invalid path from  $t$  in  $S$ ); if  $p_i = \emptyset$ , then  $e_{t+1}(i) = \emptyset$  also, and if  $p_i = \text{SELF}$ , then  $e_{t+1}(i) = t+1$ .

We now formally define how the computation proceeds.

**Definition 14.** Let  $\mathcal{A} = \langle \Sigma, d, \Gamma, k, Q, \delta, q_0, F \rangle$  be a scaffolding automaton, and  $x = \sigma_1 \cdots \sigma_n \in \Sigma^n$ . Then the *computation of  $\mathcal{A}$  on  $x$* , denoted  $\mathcal{A}(x)$ , is a sequence

$$\mathcal{A}(x) = ((q_0, S_0), (q_1, S_1), \dots, (q_n, S_n)) \in (Q \times \mathbb{S}(d, \Gamma))^{1+n}.$$

Having defined  $(q_i, S_i)$  up to some  $i < n$  — notice that  $q_0$  is the initial state and  $S_0$  is the initial scaffold — we let  $(q_{i+1}, S_{i+1}) = \text{Step}_{\delta, \sigma_{i+1}}(q_i, S_i)$ .

**Definition 15.** Let  $\mathcal{A} = \langle \Sigma, d, \Gamma, k, Q, \delta, q_0, F \rangle$  be a scaffolding automaton, and  $x = \sigma_1 \cdots \sigma_n \in \Sigma^n$ . Let  $\mathcal{A}(x) = ((q_0, S_0), (q_1, S_1), \dots, (q_n, S_n))$  be the computation of  $\mathcal{A}$  on  $x$ . We say that  $\mathcal{A}(x)$  is *accepting* if  $q_n \in F$ ; otherwise we say it is *rejecting*. This defines the *language decided by  $\mathcal{A}$* :

$$\mathcal{L}(\mathcal{A}) = \{x \in \Sigma^* \mid \mathcal{A}(x) \text{ is accepting}\}.$$

#### 4.2. Illustrative Examples, Revisited

We will soon prove that a language has a parsing expression grammar if and only if its reverse is decided by a scaffolding automaton — this is Theorem 16 of Section 4.3. However, in order to become more familiar with the model, let us begin by directly constructing scaffolding automata for the reverse of the languages seen in Section 3.

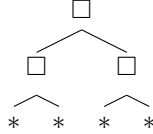
For each  $\ell \in \mathbb{N}$ , the power-length language  $\mathcal{P}_\ell^r = \mathcal{P}_\ell = \{a^{\ell^n} \mid n \geq 0\}$  is its own reversal, so let us construct a scaffolding automaton  $\mathcal{A}_\ell$  which decides  $\mathcal{P}_\ell$ . Informally, an automaton for  $\mathcal{P}_\ell$  behaves as follows. The automaton makes sure that every node in the scaffold has an edge to the previous node. It first accepts after reading the first  $a$ , and then after reading the first  $\ell$ -many  $a$ 's — so it accepts  $a$  and  $a^\ell$ . From this point onward a second edge will be maintained that goes backward in the scaffold; we call this edge



the *backtracking edge*; the idea is that for each  $\ell - 1$  new symbols read, the backtracking edge in the new top node will be moved a single position backwards (towards the base of the scaffold); once the backtracking edge reaches the base, the automaton enters an accepting state and again points the backtracking edge to the new top. This way, the next accepted string will have  $\ell$ -times as many symbols as the previous accepted string.<sup>5</sup>

Let us translate this informal description to the formal definitions given in the previous section. This will be the only scaffolding automaton for which we will do such a translation.

The scaffolding automaton for  $\mathcal{P}_\ell$  is given by  $\mathcal{A}_\ell = \langle \Sigma = \{a\}, d = 2, \Gamma = \{\boxtimes, \square\}, k = 2, Q, \delta, q_0, F = \{q_1, q_\ell, q''_{\ell-1}\} \rangle$ , where  $Q = \{q_0, q_1, \dots, q_\ell, q'_1, \dots, q'_{\ell-1}, q''_1, \dots, q''_{\ell-1}\}$ . The degree  $d$  equals 2, and at each node in the scaffold edge 0 will always point to the previous node, and edge 1 will be the backtracking edge. We will use wildcards when describing elements of  $\mathcal{N}_k(\Gamma, d)$ , so for example  $*$  means *any element* of  $\mathcal{N}_k(\Gamma, d)$  and



means any element of  $\mathcal{N}_k(\Gamma, d)$  (which consists of trees of depth 2, not trees of depth 1) whose topmost three nodes are labelled as in the picture above.

The transition function for  $\mathcal{A}_\ell$  may now be defined. In pages 19 and 20 below, we include the diagrams of the two scaffolds resulting from executing  $\mathcal{A}_2$  and  $\mathcal{A}_3$  on the string  $a^{10}$ . It might be helpful to follow those pictures, to get a sense of how  $\mathcal{A}_\ell$  works.

- If we are in the initial state and scaffold, the new top will point to the base, will be labelled by  $\boxtimes$ , and we move to state  $q_1$ :

$$\delta(q_0, a, *) = (q_1, \boxtimes, \lambda, \emptyset).$$

Above,  $\lambda$  denotes the empty path, i.e., it is the path to the top node. This edge, edge number 0, will always be set in this way, so that we may always refer to the previous top node by following edge 0. The label  $\boxtimes$  will be used to distinguish the first node from the rest.

---

<sup>5</sup>Because  $\ell^k = \ell^{k-1} + \ell^{k-1}(\ell - 1)$ .

- We then count  $\ell - 1$  symbols, as follows: For every  $i \in \{1, \dots, \ell - 1\}$  we set

$$\delta(q_i, a, *) = (q_{i+1}, \square, \lambda, \emptyset).$$

- The state  $q_\ell$  is accepting. The next symbol — symbol number  $\ell + 1$  — triggers the beginning of two nested loops, the *outer loop* and the *inner loop*. As we begin the inner loop we point the backtracking edge to the current node in the scaffold (given by the empty path  $\lambda$ ):

$$\delta(q_\ell, a, *) = (q'_1, \square, \lambda, \lambda).$$

The inner loop will loop between the states  $q'_1, \dots, q'_{\ell-1}$ , in such a way that, for each sequence of  $\ell - 1$  input symbols, the backtracking edge is moved backwards a single position in the scaffold. This happens until the backtracking edge reaches the node immediately before the base of the scaffold, at which point we enter the state  $q''_1$ , which runs the inner loop one last time until reaching state  $q''_{\ell-1}$ , which is accepting; at state  $q''_{\ell-1}$ , we “reset” the backtracking edge, and we restart the inner loop at  $q'_1$ . The outer loop consists of this resetting and restarting of the inner loop.

Let us implement the inner and outer loops. The inner loop counts  $\ell - 1$  symbols, as follows: for every  $i \in \{1, \dots, \ell - 2\}$  we set

$$\delta(q'_i, a, *) = (q'_{i+1}, \square, \lambda, (1)).$$

When we have finished the inner cycle but have still not found the  $\boxtimes$ -marked node, we move the backtracking edge backwards, and loop the inner cycle:

$$\delta \left( q'_{\ell-1}, a, \begin{array}{c} \square \\ \swarrow \quad \searrow \\ \square \quad \square \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ * \quad * \quad \square \quad * \end{array} \right) = (q'_1, \square, \lambda, (1, 0)).$$

- Eventually the top node sees node 1 of the scaffold at distance 2 through the backtracking edge — which we may detect since node 1 is labelled with  $\boxtimes$  instead of  $\square$ . At this point we will finish running the inner loop using the  $q'$  states, and then run it one last time using the  $q''$  states, which behave just like the  $q'$  states, except that  $q''_{\ell-1}$  is an accepting state whereas  $q'_{\ell-1}$  is not, and  $q''_{\ell-1}$  resets the backtracking edge.

This is implemented by setting

$$\delta \left( q'_{\ell-1}, a, \begin{array}{c} \square \\ \swarrow \quad \searrow \\ \square \quad \square \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ * \quad * \quad \boxtimes \quad * \end{array} \right) = (q''_1, \square, \lambda, (1, 0)),$$

and, for each  $i \in \{1, \dots, \ell - 2\}$ ,

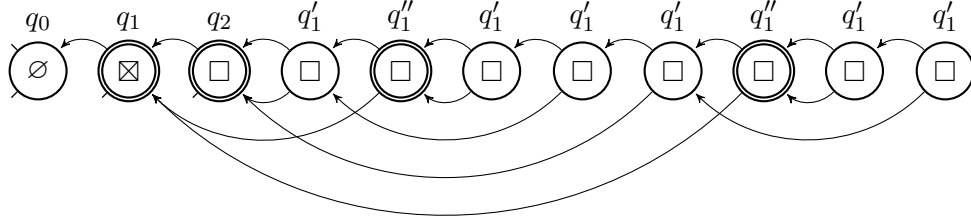
$$\delta(q''_i, a, *) = (q''_{i+1}, \square, \lambda, (1)),$$

and finally

$$\delta(q'_{\ell-1}, a, *) = (q'_1, \square, \lambda, \lambda).$$

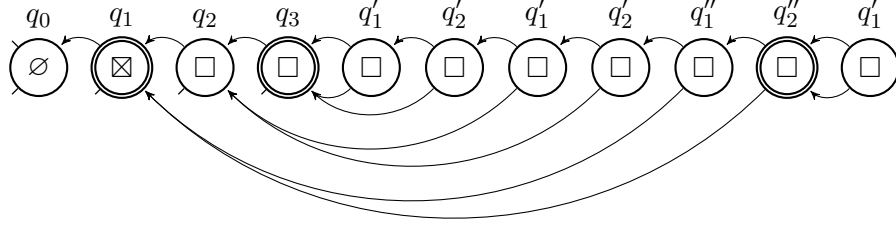
Compare  $q''_{\ell-1}$  with  $q'_{\ell-1}$ :  $q''_{\ell-1}$  is an accepting state whereas  $q'_{\ell-1}$  is not, and  $q''_{\ell-1}$  resets the backtracking edge, whereas  $q'_{\ell-1}$  moves the backtracking edge one node backwards.

In the setup above, each run of the outer cycle consumes  $\ell - 1$ -times as many symbols as the previous run, thus multiplying the total number of consumed symbols by  $\ell$ . For example, let us picture the run of  $\mathcal{A}_2$  on the string  $a^{10}$ .



In the picture, the upper edge points to the previous node, and the lower edge is the backtracking edge. The state of the automaton when reading each node of the scaffold appears above the node, and the node is drawn as a double circle if this state is an accepting state. As required, the automaton accepts after seeing 1, 2, 4, and 8 symbols.

For further illustration, let us picture the run of  $\mathcal{A}_3$  on  $a^{10}$ :



We started by describing the behaviour for  $\mathcal{A}_\ell$  in some detail, and then provided a fully formal specification. We will now limit ourselves to describing the behaviour in *sufficient* detail, so that the reader may be convinced that a fully formal specification may also be done.

Let us now sketch the scaffolding automata for the remaining two examples of Section 3.

Recognising the language of palindromes of power-two length (which also is its own reversal) uses the same idea of maintaining a backtracking edge, and it is similar to the  $\ell = 2$  case of the implementation just shown. The backtracking edge is used not only to ensure that the length of the input is a power of two, but is also used to compare the last read symbol with its corresponding symbol. The corresponding symbol, as it turns out, is exactly the symbol under the backtracking edge, as may be verified by the reader by inspecting the run of  $\mathcal{A}_2$  on  $a^{10}$ , pictured above. In order to make this comparison, thus, the scaffolding automaton may simply label each node with the symbol which was read at that position, and then compare the label of the node under the backtracking edge with the symbol which is now being read. The automaton remembers any violation of this requirement in its finite control, and at each power-of-two length, it accepts if and only if no violation was found.

A scaffolding automaton for recognising the counting language works as follows. The first item in the sequence is of fixed finite length and thus may be recognised —  $\#0^r \circ 0$ . Then noticing that if we have recognised the sequence up to  $\cdots (n-1)_2^r \circ (n-1)_2 \#$  and have an edge pointing to the rightmost bit of  $(n-1)_2$ , then we may verify, one by one from left-to-right, the bits of  $(n)_2^r$  by the usual algorithm for addition. Then we must see a  $\circ$ , and, having kept an edge pointing to the rightmost bit of  $(n)_2^r$ , we may now recognise a reversal of  $(n)_2^r$ , i.e.  $(n)_2$ . Then we must see a  $\#$ . So we have now recognised  $\cdots (n)_2^r \circ (n)_2 \#$ , and we repeat.

This trick, which we have called *reverse and scan*, will be used in the proofs of Theorems 18 and 23.

#### 4.3. Equivalence with PEGs

The rest of this section is devoted to proving that scaffolding automata exactly characterise parsing expression grammars:

**Theorem 16.** A language  $L \subseteq \Sigma^*$  is in PEG if and only if its reverse  $L^r$  is decided by some scaffolding automaton.

The question of whether PEG languages are closed under reverse now arises quite naturally. We conjecture that they are not, but Theorem 18 below suggests it will be very hard to prove such a result.

*Proof of Theorem 16, necessary direction.* We begin by proving that a parsing expression grammar for a language  $L \subseteq \Sigma^*$  gives rise to a scaffolding automaton for  $L^r$ . A reader who is familiar with the tabular parsing algorithm of Birman and Ullman [2] for TDPLs should be able to easily see that a scaffolding automata can simulate this algorithm (the edges will correspond to entries in the table). Since Ford [1] has shown TDPLs are equivalent to PEGs, that suffices for obtaining the result.

But Ford's proof of equivalence between PEGs and TDPLs is complex and delicate, whereas scaffolding automata are powerful enough to simulate PEGs directly. So we will prove the result here in full.

Let  $\mathcal{G} = \langle \Sigma, \text{NT}, R, S \rangle$  be a total parsing expression grammar. Without loss of generality, we may assume that every rule of  $\mathcal{G}$ , has one of the forms:

- $A \leftarrow \varepsilon$ ,  $A \leftarrow \text{FAIL}$ , or  $A \leftarrow t$ , with  $A \in \text{NT}$  a non-terminal symbol and  $t \in \Sigma$  a terminal symbol.
- $A \leftarrow !B$ ,  $A \leftarrow \&B$  with  $A, B \in \text{NT}$ .
- $A \leftarrow BC$ ,  $A \leftarrow B/C$  with  $A, B, C \in \text{NT}$ .

Indeed, any grammar may be converted into the form above by replacing sub-expressions with new non-terminal symbols.<sup>6</sup>

We then construct a scaffold automaton  $\mathcal{A} = \langle \Sigma, d, \Gamma, k, Q, \delta, q_0, F \rangle$ , where

- $d = |\text{NT}|$  and  $k = |\text{NT}|$ .

---

<sup>6</sup>For example, one would convert the rule  $A \leftarrow \&BCD/EF/!G$  to the rules  $A \leftarrow A_1/A_3$ ,  $A_1 \leftarrow B_1A_2$ ,  $B_1 \leftarrow \&B$ ,  $A_2 \leftarrow CD$ ,  $A_3 \leftarrow A_4/A_5$ ,  $A_4 \leftarrow EF$  and  $A_5 \leftarrow !G$ .

- $\Gamma = \{\square\}$ , as we will use a single label, to distinguish the end of the input from the remaining nodes.
- $Q = \{q_{\text{yes}}, q_{\text{no}}\}$ , as we will use only two states, which will behave identically except that only one is accepting.
- $q_0 = q_{\text{yes}}$  if  $\lambda \in \mathcal{L}(G)$  and  $q_0 = q_{\text{no}}$  otherwise.
- $F = \{q_{\text{yes}}\}$ .

For  $q \in Q$ ,  $\sigma \in \Sigma$  and  $N = (V, E, L) \in \mathcal{N}_k(d, \Gamma)$ , the transition function has

$$\delta(q, \sigma, N) = (q', \square, p_0, \dots, p_{d-1}),$$

defined as follows. Fix some ordering of **NT**, and if  $A$  is the  $i$ -th non-terminal symbol in **NT**, let us use  $p_A$  in place of  $p_i$ . Then:

- If  $A \leftarrow \varepsilon$ , set  $p_A = \text{SELF}$ , i.e., create a self loop in the new top node.
- If  $A \leftarrow \text{FAIL}$ , or  $A \leftarrow \sigma'$  with  $\sigma' \neq \sigma$ , then set  $p_A = \emptyset$  — the new top node will be missing edge  $A$ .
- If  $A \leftarrow \sigma$ , then set  $p_A = \lambda$ , i.e., create an edge from the new top to the previous top node.
- If  $A \leftarrow !B$ , then we must first compute  $p_B$ , and then we set  $p_A = \text{SELF}$  if  $p_B = \emptyset$ , and  $p_A = \emptyset$  otherwise.
- If  $A \leftarrow \&B$ , then we must first compute  $p_B$ , and then we set  $p_A = \text{SELF}$  if  $p_B \neq \emptyset$ , and  $p_A = \emptyset$  otherwise.
- If  $A \leftarrow BC$ , then we must first compute  $p_B$ ; if  $p_B = \emptyset$ , then we set  $p_A = \emptyset$  also; otherwise  $p_B$  is a path to some node  $v_B$  in  $N$ ; this node will have some edge to  $v_{BC} = e_{v_B}(C)$  in  $N$  corresponding to  $C$ ; we then let  $p_A$  be a path to  $v_{BC}$ , which is one edge longer than  $p_B$ . This is where we require  $k \geq |\mathbf{NT}|$ .<sup>7</sup>

---

<sup>7</sup>It may be proven by induction on  $|\mathbf{NT}|$  that whenever we set an edge of the new top node, it will be at a distance no greater than  $|\mathbf{NT}|$  from the previous top node of the scaffold. Indeed, the only rule which may cause the required distance to increase is the concatenation rule  $A \leftarrow BC$ . In this case, when the edge  $p_B$  points to a node  $v_B$  which is a distance  $i$  from the previous top node in the scaffold, then  $p_A$  will point to the same node  $v_{BC}$  as the edge  $e_{v_B}(C)$  of  $v_B$  corresponding to the non-terminal  $C$ . So the distance from the previous top node to  $v_{BC}$  is now the distance to  $v_B$  plus one, i.e.,  $i + 1$ . Since, as we argue later, there are no circular dependencies, the maximum distance is then  $|\mathbf{NT}|$ .

- If  $A \leftarrow B/C$ , then we must first compute  $p_B$  and  $p_C$ , and then we set  $p_A = p_B$ , if  $p_B \neq \emptyset$ , and otherwise we set  $p_A = p_C$ .

In the above procedure, we may assume that  $p_B$  and  $p_C$  are computed before  $p_A$ , when the rule for  $A$  depends on  $B$  and  $C$ . This is because the dependencies of the above procedure (when we say “we must first compute ...”) correspond exactly to the subroutine calls of the recognition procedure  $\text{Rec}_{\mathcal{G}}$ . Hence, if we have a cyclic dependency above this will cause  $\text{Rec}_{\mathcal{G}}$  to enter an infinite loop, and our assumption that  $\mathcal{G}$  is total implies that this never happens on any input. Hence if at some point a cyclic dependency is triggered, e.g. “before computing  $p_A$  we must first compute  $p_B$  and before computing  $p_B$  we must compute  $p_A$ ”, then it may safely be ignored by setting the edge  $p_A = \emptyset$ , since we are guaranteed, by the totality of  $\mathcal{G}$ , that  $\text{Rec}_{\mathcal{G}}$  will not be called for the non-terminal  $A$  at this position, on any input.<sup>8</sup>

The above definition ensures that the following property always holds:

**Claim 17.** Let  $x^r = x_n \cdots x_1 \in \Sigma^n$  and consider the scaffold  $S = (V, E, L)$  obtained at the last step of the computation of  $\mathcal{A}$  on  $x^r$ . Then the edge of the top node  $n \in V$  corresponding to the non-terminal  $A \in \text{NT}$  will be present if and only if the corresponding parsing expression  $R(A)$  accepts  $x = x_1 \cdots x_n$ . When present, this edge will point to the position of  $x^r$  corresponding to the symbol after  $\text{Rec}_{\mathcal{G}}(R(A), x)$ . I.e., if  $|\text{Rec}_{\mathcal{G}}(R(A), x)| = \ell \geq 0$  is the number of consumed symbols, then  $e_n \in E$  has  $e_n(A) = n - \ell$ .

Having defined how we create the new top node, it suffices to explain how the new state  $q'$  is chosen. We will set  $q' = q_{\text{yes}}$  if the new edge  $e_t(S)$ , where  $t$  is the new top node, and  $e_t(S)$  is the edge corresponding to the starting non-terminal of  $\mathcal{G}$ , has been set to equal a node with empty label, i.e. if  $L(e_n(S)) = \emptyset$ . We set  $q' = q_{\text{no}}$  otherwise. Since only the base of the scaffold has an empty label, we will be in an accepting state if and only if  $S$  consumes the entire input seen thus far. By Claim 17 it follows that  $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{G})$ .  $\square$

*Proof of Theorem 16, sufficient direction.* Now let  $\mathcal{A} = \langle \Sigma, d, \Gamma, k, Q, \delta, q_0, F \rangle$  be a scaffolding automaton accepting the language  $L$ . Assume without loss of generality (by duplicating states) that  $\mathcal{A}$  is only in the initial state  $q_0$  at the very beginning of the computation, and never re-enters it after reading the first symbol.

---

<sup>8</sup>Incidentally, it is based on this observation that one may convert a total PEG  $\mathcal{G}$  into an equivalent well-formed PEG. See the discussion after Definition 4.

We construct a parsing expression grammar  $\mathcal{G} = \langle \text{NT}, \Sigma, R, S \rangle$  recognising  $L^r$ . The grammar  $\mathcal{G}$  will have the following non-terminals:

- For each  $q \in Q$ , we have a non-terminal  $\text{State}(q)$ .
- For each  $\gamma \in \Gamma$ , we have a non-terminal  $\text{Label}(\gamma)$ .
- For each  $N \in \mathcal{N}_k(d, \Gamma)$ , we have a non-terminal  $\text{Neighbourhood}(N)$ .
- For each  $p \in [d]^{\leq k}$ , we have a non-terminal  $\text{Path}(p)$ .
- The initial non-terminal of the grammar is  $\text{AutomatonAccepts}$ .

Now we will define various grammar rules, of the form

$$N \leftarrow N_1 / N_2 / N_3 / \dots,$$

where  $N$  is one of the non-terminals  $\text{State}(q)$ ,  $\text{Label}(\gamma)$ , *etcetera*, and  $N_1, N_2, \dots$  are parsing expressions.

Below, when we say that we “add an alternative  $N \leftarrow E$ ”, we mean that the rule corresponding to the non-terminal  $N$  should have the parsing expression  $E$  appearing as one of the parsing expressions  $N_i$  on the right-hand side. If no alternative was added in this process, for a given non-terminal  $N$ , then the rule corresponding to  $N$  is instead  $N \leftarrow \text{FAIL}$ .

So, for example, if during the proof we add the alternative  $N \leftarrow A$ , the alternative  $M \leftarrow B$ , then the alternative  $N \leftarrow C$ , and no other alternatives were added, then the resulting grammar will have the rules  $N \leftarrow A / C$  and  $M \leftarrow B$ , and for every non-terminal  $O$  other than  $N$  and  $M$ , we will have the rule  $O \leftarrow \text{FAIL}$ .

This allows us to specify how each transition of the scaffolding automaton affects the different rules appearing in the grammar. If we had to specify each rule of the grammar completely, then we would need to define the rules of the grammar in a fixed order with respect to the non-terminal appearing on the left side, which would obscure the idea behind the construction.

Let  $\Sigma = \{\sigma_1, \sigma_2, \dots\}$  give the (finitely-many) symbols of  $\Sigma$ . The rules of the grammar are defined as follows. We have the rule

$$\text{State}(q_0) \leftarrow ! (\sigma_1 / \sigma_2 / \dots)$$



and if  $N_0$  is the trivial neighbourhood containing a single unlabelled node with no edges (i.e. the neighbourhood of the top node of the initial scaffold), we also have the rule

$$\text{Neighbourhood}(N_0) \leftarrow ! (\sigma_1 / \sigma_2 / \dots)$$

This ensures that the end of the input of the grammar (which is the beginning of the input of the automaton) matches the initial state and neighbourhood.

Now for each possible  $q \in Q$ ,  $\sigma \in \Sigma$ , and  $N \in \mathcal{N}_k(d, \Gamma)$ , we have a transition

$$\delta(q, \sigma, N) = (q', \gamma, p_0, \dots, p_{d-1}).$$

Recall that this transition means “if the scaffolding automaton is in state  $q$ , reads input symbol  $\sigma$ , and the neighborhood of the current top node is  $N$ , then it will move to state  $q'$ , and create a new top node with label  $\gamma$ , with edges given by the paths  $p_0, \dots, p_{d-1} \in [d]^{\leq k} \cup \{\text{SELF}, \emptyset\}$ .”

Let us write  $\text{Transition}(q, \sigma, N)$  as an abbreviation for the parsing expression

$$\&(\sigma \text{ State}(q)) \&(\sigma \text{ Neighbourhood}(N)).$$

We then add the alternative

$$\text{State}(q') \leftarrow \text{Transition}(q, \sigma, N).$$

These alternatives will be added for every transition given by  $\delta$ . It will follow, by induction on the length of the input string, that  $\text{State}(q)$  will accept the string  $x_i \cdots x_1$  if and only if the computation  $\mathcal{A}(x_1 \cdots x_i)$  ends in state  $q$ ; even when it accepts,  $\text{State}(q)$  will never consume any input. Let  $F = \{f_1, f_2, \dots\}$  give the (finitely-many) accepting states. We then naturally have the rule

$$\text{AutomatonAccepts} \leftarrow (\text{State}(f_1) / \text{State}(f_2) / \dots) .*$$

Then let  $\lambda \in [d]^0$  be the sequence of length 0. We add the alternative

$\text{Path}(\lambda) \leftarrow \varepsilon$ , i.e.,  $\text{Path}(\lambda)$  is always accepted and consumes no input. Now take a sequence  $ip \in [d]^{1+\ell}$  of length  $1 + \ell \geq 1$ ; then if  $p_i \notin \{\emptyset, \text{SELF}\}$ , we add the alternative

$$\text{Path}(ip) \leftarrow \text{Transition}(q, \sigma, N) \sigma \text{ Path}(p_i) \text{ Path}(p)$$

If  $p_i = \emptyset$ , we instead add the alternative:

$$\text{Path}(ip) \leftarrow \text{Transition}(q, \sigma, N) \text{ FAIL}$$

And if  $p_i = \text{SELF}$ , we instead add the alternative:

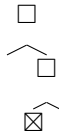
$$\text{Path}(ip) \leftarrow \text{Transition}(q, \sigma, N) \text{ Path}(p)$$

It will follow by induction that the non-terminal  $\text{Path}(p)$  will accept the string  $x_i \cdots x_1$  if and only if path  $p$  goes from the top of the scaffold in the computation  $\mathcal{A}(x_1 \cdots x_i)$ , i.e. from node  $i$  in that scaffold, to some node  $j \leq i$ . And, if the non-terminal  $\text{Path}(p)$  accepts  $x_i \cdots x_1$ , it will consume the input exactly up to (but not including) position  $j$ , i.e., it will consume the string  $x_i \cdots x_{j+1}$  (the entire string will be consumed if  $j = 0$ , i.e., if the edge points to the base of the scaffold). Finally, we add the alternative

$$\text{Label}(\gamma) \leftarrow \text{Transition}(q, \sigma, N)$$

The above alternatives may be added in any order, since the various conditions  $\text{Transition}(q, \sigma, N)$  are disjoint. The following observation is *crucial* to understand why the above definitions are well-founded: the expression  $\text{Transition}(q, \sigma, N)$  uses **State** and **Neighbourhood** non-terminals, but *only after consuming symbol  $\sigma$* ; so the accepting/consuming of the various non-terminals depends on the accepting/consuming of the same non-terminals, but in prior positions of the input, where this has already been determined.

All we are left to do is explain how each **Neighbourhood** is defined. But notice that knowing whether the top of a scaffold has a certain neighbourhood consists of checking that certain paths exist, and that the nodes under these paths have certain labels, and that certain other paths do not exist. For example, if we wish to check for the neighbourhood  $N \in \mathcal{N}_2(2, \{\square, \boxtimes\})$  where the top node is labelled  $\square$ , the second edge of the top node leads to a child labelled  $\square$  and that child has itself a child labelled  $\boxtimes$  on its first edge, i.e., if  $N$  is the neighbourhood:



we then have the rule:

$$\begin{aligned} \text{Neighbourhood}(N) \leftarrow & \\ & \&\text{Label}(\square) \\ & \&\text{Path}(0) \&\text{Path}(1) \\ & \&(\text{Path}(1) \text{Label}(\square)) \\ & \&\text{Path}(1,0) \&\text{Path}(1,1) \\ & \&(\text{Path}(1,0) \text{Label}(\boxtimes)) \end{aligned}$$

With this observation the proof is now complete.  $\square$

We would like to make the following remark. It may be observed in the grammar above, which simulates a given scaffolding automaton, that the different alternatives may all be added in any order, since they cover disjoint cases. The reader should now suspect that the prioritized choice operator  $/$  may, after all, be replaced by the usual disjunction operator  $|$  from context-free grammars. This is entirely correct, since  $A / B$  is equivalent to  $A | (!A)B$ , where  $!$  is the negation operator of PEGs. It is the  $!$  operator that we cannot do away with: our simulation of scaffolding automaton uses the  $!$  operator both for detecting the end of the input and for detecting the absence of a path in the scaffold. Interestingly, it is possible to modify the above construction to remove the second use case, by adding an extra family of non-terminal symbols  $\text{NoPath}(p)$ , that accepts the input exactly when  $p$  is not a valid path starting at that position. The result of this is that any parsing expression grammar may be replaced by a grammar where the operators appearing in parsing expressions are  $\&$ ,  $|$ , and the special symbol  $\text{EndOfInput}$ , which accepts only at the end of the input. Details are left to the reader.

## 5. Applications

In this section we will use Theorem 16 to prove all of the remaining results mentioned in the abstract.

### 5.1. “Universality”

**Theorem 18.** Let  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  be any computable function. Then there exists a computable function  $g : \{0, 1\}^* \rightarrow \mathbb{N}$  such that the language

$$L = \{f(x)\$^\ell x \mid x \in \{0, 1\}^*, \ell \geq g(x)\} \subseteq \{0, 1, \$\}^*$$

has a parsing expression grammar.

*Proof.* We describe a scaffolding automaton for the reverse language  $L^r$ , and then the result follows from Theorem 16. The basic idea is to use the *reverse and scan* trick. For this purpose, let  $M$  be a one-tape Turing machine computing  $f$ .

The automaton first reads the input  $x^r$ , copying the symbols of  $x^r$  to the labels of the corresponding nodes and adding an edge connecting each node to the previous one. It then finds the first  $\$$  symbol; at this point it continues reading  $\$$  symbols, while successively labelling the corresponding nodes of the scaffold with the successive configurations of the Turing machine  $M$  on input  $x$ . After this it checks that the input matches the output of  $M$  on input  $x$ . So, if  $c_i$  is the configuration of  $M$  on input  $x$  at time-step  $i$ , and  $M$  runs for  $t$  time steps on input  $x$ , then the labels, when seen from first to last, form the string:

$$\begin{array}{ll} \text{labels:} & x^r \# \# c_0 \# c_0^r \# \# c_1 \# c_1^r \# \# c_2 \# c_2^r \# \# \cdots c_t \# c_t^r \# \# \underbrace{\# \dots \#} \\ \text{input:} & x^r \$ \$ \$ \$ \$ \$ \$ \$ \$ \$ \$ \$ \$ \$ \$ \cdots \$ \$ \$ \$ \$ \$ f(x)^r \end{array}$$

Here  $\#$  is being used as a separator. Note that  $\$$  is also being used as a separator, but the symbol  $\$$  is part of the actual language being recognized, and the symbol  $\#$  is part of the alphabet being used to label the scaffold.

One may verify that the above labelling can be produced by a scaffolding automaton, provided we choose a reasonable encoding for Turing machine configurations (and for this purpose the working alphabet can be as large as desired). For example, we may encode a configuration by the sequence of symbols on the tape, and the position of the tape head will be additionally marked with some (finite) information containing the current state of the computation. With such an encoding, the scaffolding automaton can, for each  $i$ , produce the labels in the sequence  $c_{i+1}$ , provided that when reaching the first symbol of  $c_{i+1}$ , the top of the scaffold has an edge pointing to the last symbol of  $c_i^r$  (which is easy to ensure), and that each node in the scaffold has an edge to the previous node; then the labelling  $c_{i+1}$  is produced one symbol at a time by scanning  $c_i^r$  starting with its last symbol, and producing the symbols of  $c_{i+1}$  according to the transition function of  $M$ . Similarly, for each  $i$ , one may produce the labels in the sequence  $c_i^r$ , provided that when reaching the first symbol of  $c_i^r$ , the top of the scaffold has an edge pointing to the last symbol of  $c_i$ ; then the labelling  $c_i^r$  is produced by copying one symbol at a time.

The scaffolding automaton finally accepts if the last  $\$$  symbol corresponds exactly to the last position of the (reversal of) last configuration of the computation of  $M$  on  $x$ , and the last  $\$$  symbol is followed by the

string  $y$  which is the reverse of the output written on the tape, in that final configuration; i.e. if it is followed by  $f(x)^r$ .  $\square$

We may now show that the recognition procedure underlying parsing expression grammars is complete for polynomial time, under logspace reductions. This was previously unknown, and stands in contrast with context-free grammars. In the case of context-free grammars, we may define the complexity class **LOGCFL**, to be the class of languages which are reducible to context-free languages under logspace reductions. It may be proven that this is exactly the class of languages decidable by log-depth Boolean circuits where the OR gates have arbitrary fan-in, and the AND gates have fan-in 2 [see 37, p. 137]. In particular, **LOGCFL** is a sub-class of **NC<sub>2</sub>**, which is believed to be strictly contained in **P**.

In contrast, if we were to define an analogous complexity class **LOGPEG**, containing those languages that are reducible, via logspace reductions, to PEG-recognizable languages, it turns out that **LOGPEG** = **P**. It is easy to see that **LOGPEG**  $\subseteq$  **P**, since **PEG**  $\subseteq$  **P** and **P** is closed under logspace reductions. The other direction follows as a corollary of Theorem 18.

**Corollary 19.** There is a language  $L \in \text{PEG}$  which is complete for **P** under logspace reductions.

*Proof.* Notice in the proof of Theorem 18 that the resulting function  $g : \{0, 1\}^* \rightarrow \mathbb{N}$  grows quadratically in the running time of the Turing machine  $M$ . Now consider the function  $f$  such that  $f(x) = 1$  if  $x$  encodes a triple  $\langle N, 0^t, y \rangle$  where, in turn,  $N$  encodes a Turing machine which accepts input  $y$  in  $t$  or fewer steps, and  $t \geq |N| + |y|$ . And let  $f(x) = 0$  otherwise. Then, computing  $f(x)$  is a problem which is complete for polynomial time under logspace reductions. There are machines for computing  $f$  in time  $O(t^2)$ , and hence  $g(\langle N, 0^t, y \rangle) = O(t^4) \leq c \cdot t^4$  for some sufficiently large integer constant  $c$ . The language  $L$  of Theorem 18 is thus also complete for polynomial time under logspace reductions, since  $f(\langle N, 0^t, y \rangle) = 1$  if and only if  $1\$^{c \cdot t^4} \langle N, 0^t, y \rangle \in L$ , and the string  $1\$^{c \cdot t^4} \langle N, 0^t, y \rangle$  may be computed from  $\langle N, 0^t, y \rangle$  in logarithmic space.  $\square$

## 5.2. Impossibility of a Pumping Lemma

We may define a pumping lemma by the following:

**Definition 20.** A *pumping lemma for PEGs* is a total computable function  $A$  such that, for every total<sup>9</sup> PEG  $G$ , there exists a length  $n_0$  such that for every string  $x \in \mathcal{L}(G)$  of size  $|x| \geq n_0$ , the output  $y = A(G, x)$  is in  $\mathcal{L}(G)$  and has  $|y| > |x|$ .

Some explanation is required as to why this definition is the right one.

- The first observation we may make is that, to our knowledge, every pumping lemma proven thus far either already is of the above form (e.g. [25, 28, 29]) or can be made to work in the above form with few modifications (e.g. considering resource-bounded Kolmogorov complexity in [26]).
- The second observation is that if  $A$  is not required to be total, then the definition trivialises: there exists a pumping lemma for every recursively-enumerable language. Indeed given any Turing machine  $M$  and input  $x$ ,  $A$  can simply dovetail on all  $y$  larger than  $x$  until it finds a larger  $y$  accepted by  $M$  (if no such  $y$  is found,  $M$  decides a finite language, and so the requirement on  $A$  is trivially satisfied).
- We mention also that the definition is equivalent to one where  $A$  is required to produce an infinite sequence  $y^{(1)}, y^{(2)}, \dots$  of strings of increasing size, which is what one typically sees in pumping lemmas.

**Theorem 21.** There is no pumping lemma for PEGs.

We must show that any candidate computable function  $A$  must fail on some grammar. Intuitively one may quickly realise, by way of Theorem 18, that the size of “the next string” in the language decided by a parsing expression grammar may well grow as high as any computable function of our choice. Hence given any candidate procedure  $A$  meant to serve as a pumping lemma, we should be able to find a PEG language such that the gap between consecutive words grows faster than what the existence of  $A$  would allow. The only difficulty in making this argument precise is that we wish to run algorithm  $A$  on a PEG for the very same language we are trying to define. This is solved much the same way as in the proof of Kleene’s

---

<sup>9</sup>Although the totality of a given PEG is undecidable, the results of this section still hold if “total” is replaced by “well-formed”. (Recall that well-formedness of PEGs is a decidable syntactic restriction which ensures totality. See remarks after Definition 4.) It should be understood, hence, that the impossibility of a pumping lemma is not a hidden consequence of the undecidability of totality.

second recursion theorem (see [38], §6.1): one shows that it is possible to construct a scaffolding automaton which has access to its own encoding.

*Proof.* For any scaffolding automaton  $X$ , let  $\langle X \rangle$  be a binary encoding of  $X$ . Let  $S \in \mathbb{S}(d, \Gamma)$  be a scaffold and  $w \in \Gamma^n$ . We say that  $S$  *sees  $w$  written backwards* if, for every  $\ell \in [n]$ , following the first edge once and then the second edge  $\ell$  times, from the top of  $S$ , will place us in a node labelled by  $w_{n-\ell}$ . Suppose we have a scaffolding automaton  $C$ , which accepts an input of the form  $\$^s \langle X' \rangle$ , where  $\langle X' \rangle$  in turn is the encoding of some scaffolding automaton  $X'$ . Let  $\langle C \rangle$  be an encoding of  $C$ . We then define a scaffolding automaton  $X_{\langle C \rangle}$ , which recognises a language  $\mathcal{L}(X_{\langle C \rangle}) = \{y_1, y_2, \dots\}$ , via the following procedure:

- $X_{\langle C \rangle}$  begins by checking that the input begins with  $\langle C \rangle$ , in such a way that after this check, the resulting scaffold sees  $\langle C \rangle$  written backwards;
- $X_{\langle C \rangle}$  also maintains an edge from the current top node to the previous top node, at every step of the computation, and always copies the input into the labels of the scaffold, so it is not forgotten.
- Then  $X_{\langle C \rangle}$  simulates a run of  $C$  itself, which by assumption recognises a string of the form:

$$\$^s \langle X' \rangle$$

An edge to the last symbol of  $\langle X' \rangle$  is preserved by  $X_{\langle C \rangle}$  throughout the rest of the computation (on every top node henceforth);

- Then  $X_{\langle C \rangle}$  checks that the following input is the sequence  $\# \text{Start} \#$ , and enters an accepting state at this point.
- The scaffold now sees the string  $y_1 = \langle C \rangle \$^s \langle X' \rangle \# \text{Start} \#$  backwards.
- Then for each  $j = 1, 2, \dots$ , the automaton repeatedly:
  - Simulates the computation of  $A(G_{\langle X' \rangle}, y_j^r)$ , in order to recognise an input of the form  $\$^{aj} A(G_{\langle X' \rangle}, y_j^r) \#$ , where  $G_{\langle X' \rangle}$  is the grammar recognising the reverse of the language decided by  $X'$ . The grammar  $G_{\langle X' \rangle}$  is (constructively) given by Theorem 16, and the automaton can recognise an input of this form by way of Theorem 18. Here we require that  $A$  is total.
  - After scanning this input (while copying it into the labels of the scaffold), the automaton enters an accepting state.

– The scaffold now sees backwards:

$$y_{j+1} = y_j \$^{a_j} A(G_{\langle X' \rangle}, y_j^r) \#.$$

Let  $B$  be the scaffolding automaton which, under the assumption that the top of the scaffold sees an encoding  $\langle C \rangle$  written backwards, accepts a string of the form

$$\$^b \langle X_{\langle C \rangle} \rangle.$$

Such a scaffolding automaton  $B$  exists, by Theorem 18. Let  $\langle B \rangle$  be the code for the above scaffolding automaton. Then let us consider the scaffolding automaton  $X_{\langle B \rangle}$ , which accepts  $y_1, y_2, \dots$  — this sequence is infinite by our assumption that  $A$  is total. Note that setting  $C = B$  satisfies the assumption that  $X_{\langle C \rangle}$  makes on  $C$ . The string  $\langle X' \rangle$  recognised during execution of  $X_{\langle B \rangle}$  is exactly  $\langle X_{\langle B \rangle} \rangle$ . Hence  $G_{\langle X' \rangle} = G_{\langle X_{\langle B \rangle} \rangle}$  is a parsing expression grammar deciding the same language as  $X_{\langle B \rangle}$ , in reverse. i.e.  $G_{\langle X_{\langle B \rangle} \rangle}$  recognises the strings  $y_1^r, y_2^r, \dots$ . Now let  $n_0$  be an arbitrary natural number, and consider  $y_{n_0}$ ; clearly  $|y_{n_0}| \geq n_0$ ; and yet the smallest string larger than  $y_{n_0}$  which is accepted by  $X_{\langle B \rangle}$  is  $y_{n_0+1} = y_{n_0} \$^{a_{n_0}} A(G_{\langle X_{\langle B \rangle} \rangle}, y_{n_0}^r) \#$  — but its size is strictly greater than  $A(G_{\langle X_{\langle B \rangle} \rangle}, y_{n_0}^r)$ , and so is the size of  $y_{n_0+k}$  for any natural  $k > 1$ ; hence  $A$  must fail on the grammar  $G_{\langle X_{\langle B \rangle} \rangle}$ .  $\square$

### 5.3. PEGs vs. Online Turing Machines

Because scaffolding automata are machines which read a single input symbol at a time, and which do only a constant number of operations per symbol read, they can be thought of as a *real-time* computational model. This led us to conjecture that the reverse of any language in PEG could be recognised by a real-time Turing machine. However this conjecture turns out to be demonstrably false.

Let us begin by the following definition:

**Definition 22.** An *online Turing machine* is a Turing machine where the head of the input tape can only move in one direction. At the beginning of the computation, an input  $x \in \Sigma^*$  is written on the the input tape, and the head of the input tape sits over the leftmost symbol of  $x$ , and every time the tape head is moved to the right, we say that *another symbol from the input was read*. For convenience, an additional auxiliary tape is provided where



the input size  $|x|$  is given in binary.<sup>10</sup>

The class  $\text{Online}(t(n))$  is the class of languages  $X \subseteq \Sigma^*$  which can be decided by an online Turing machine  $M$ , in the following way. If  $x \in \Sigma^n$ , then  $M(x)$  accepts if  $x \in X$  and rejects otherwise, and furthermore, the computation  $M(x)$  does at most  $t(n)$  steps between each input symbol read.

This section is devoted to proving the following:

**Theorem 23.** There exists a language  $L \in \text{PEG}$  such that neither  $L$  nor  $L^r$  is in  $\text{Online}(t(n))$ , for any  $t(n) = o(n/(\log n)^2)$ .

The proof of this theorem uses the method of Rosenberg (see [39], §4.1), for proving lower-bounds against online Turing machines. We will explain it here for completeness.

**Definition 24.** Let  $L \subseteq \Sigma^*$  and  $\ell, m \in \mathbb{N}$ . We then say that two strings  $y_1, y_2 \in \Sigma^\ell$  are  $(L, \ell, m)$ -equivalent, which we write  $y_1 \equiv_L^{\ell, m} y_2$ , if

$$\forall x \in \Sigma^m (y_1 \cdot x \in L \iff y_2 \cdot x \in L)$$

We may then define the sets  $\mathcal{E}_L(\ell, m) = \Sigma^* / \equiv_L^{\ell, m}$  of  $(L, \ell, m)$ -equivalence classes. To each  $L \subseteq \Sigma^*$ , then, corresponds a function  $E_L : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  giving the number of  $(L, \ell, m)$ -equivalence classes:

$$E_L(\ell, m) = |\mathcal{E}_L(\ell, m)|$$

The framework of Rosenberg then rests on the following crucial observation:

**Theorem 25** ([31]). If  $L \in \text{Online}(t(n))$ , then  $E_L(\ell, m) \leq 2^{O(m \cdot t(\ell+m))}$ .

*Proof.* Let  $M$  be an online Turing machine that decides whether  $z \in L$  by making  $\leq t(|z|)$  computation steps per symbol. Let  $y \cdot x \in \Sigma^n$ , where  $y \in \Sigma^\ell$ ,  $x \in \Sigma^m$  and  $n = \ell + m$ . Consider the configuration  $C$  of the computation  $M(y \cdot x)$ , after  $M$  has read all the  $\ell$  symbols in  $y$  and done whichever computation it does on them, and precisely before it reads the

---

<sup>10</sup>So that one will not think that the lower-bounds we are about to prove result, somehow, from the fact that the machine does not know the input size. Indeed the reason why the lower-bound holds is more profound. We may even fill the auxiliary tape with any content we please (as a function of  $n$ ), i.e. the lower-bounds here proven will hold even in the presence of non-uniform advice.

first symbol of  $x$ . As  $M$  then proceeds to read the  $m$  symbols of  $x$ , it can only do  $t(n)$  steps per symbol; and thus if one would describe the configuration  $C$  partially, by giving only the state of the finite control, the position of the tape heads, and the contents of the tape heads at a distance of  $\leq m \cdot t(n)$  from the position of the tape heads, then one can simulate the entire computation to its very end.

But since there are only  $2^{O(m \cdot t(n))}$ -many such possible partial descriptions, then this behaviour can only proceed in so-many different ways.  $\square$

As a warm-up, we begin by showing the following easy result:

**Theorem 26.** There is a language  $K \subseteq \{0, 1, \#\}^*$  in PEG, such that  $E_K((D + 1) \cdot 2^D, D) \geq 2^{2^D}$  for all  $D \in \mathbb{N}$ . Hence  $K \notin \text{Online}(t(n))$ , for any  $t(n) = o(n/(\log n)^2)$ .

*Proof.* Consider the language:

$$K^r = \{x \# w_1 \# w_2 \# \dots \# w_N \mid x \in \{0, 1\}^*, \forall i \ w_i \in \{0, 1\}^*, \exists i \ x^r = w_i\}.$$

A scaffolding automaton can easily decide  $K^r$  by maintaining an edge pointing to the last symbol of  $x$ , and then for each  $w_i$  which it sees, scanning  $x$  in reverse and comparing it with  $w_i$ . Hence  $K \in \text{PEG}$  by Theorem 16.

But looking carefully at  $K = \{w_1 \# w_2 \# \dots \# w_N \# x \mid \exists i \ x^r = w_i\}$ , one sees that if we have  $N = 2^D$  strings  $w_i$  each of length  $D$ , then the suffixes  $x$  that cause acceptance are exactly those  $x$ 's in the set  $\{w_1, \dots, w_N\}$ , and there are  $2^N - 1 = 2^{2^D} - 1$  such sets. The empty set may be obtained by a malformed prefix, where none of the  $w_i$  has length  $D$ , but their concatenation, with  $\#$  as a separator, still has length  $(D + 1)2^D$ . Hence  $E_K((D + 1)2^D, D) = 2^{2^D}$ .

Now, if  $K$  were in  $\text{Online}(t(n))$  for  $t(n) = o(n/(\log n)^2)$ , by Theorem 25 we would have  $E_K((D + 1)2^D, D) \leq 2^{D \cdot t((D+1)2^D)} = 2^{o(2^D)}$ , a contradiction.  $\square$

Now we will show the following:

**Theorem 27.** There is a language  $H \subseteq \{0, 1, \#\}^*$ , decidable by a scaffolding automaton, such that  $E_H(O(D \cdot 2^D), D) \geq 2^{2^D}$ .

Hence  $H \notin \text{Online}(t(n))$ , for any  $t(n) = o(n/(\log n)^2)$ .

The proof of this theorem is significantly more involved, and uses the *reverse and scan* trick we have seen before. So let us first observe that from  $K$  and  $H$  we may obtain the language  $L$  promised by Theorem 23. Let  $L^r = \{0x0 \mid x \in K^r\} \cup \{1x1 \mid x \in H\}$ . It is easy to see that  $L^r$  has a scaffolding automaton, since  $K^r$  and  $H$  both do, and so  $L \in \text{PEG}$ . But an online Turing machine for deciding  $L^r$  can be easily converted into an online Turing machine for deciding  $H$ , and an online Turing machine for deciding  $L$  can be converted into an online Turing machine for deciding  $K$ . Hence neither  $L^r$  nor  $L$  are in  $\text{Online}(t(n))$  for any  $t(n) = o(n/(\log n)^2)$ .

*Proof of Theorem 27.* Given  $(d, \Sigma)$ -scaffold  $G = (V, E, L)$  (as per Definition 10) with  $d \geq 2$ , and a node  $v$  of  $G$ , let us define the map  $\text{Path}_{G,v} : \{0, 1\}^* \rightarrow V \cup \{\emptyset\}$ , so that  $\text{Path}_{G,v}(x_1 \cdots x_n) = v'$  if the sequence of bits  $x_1 \cdots x_n \in \{0, 1\}^n$  is a valid path from  $v$  to  $v'$  in  $G$  (as per Definition 10). If we repeat Definition 10 here, for explicitness, we get that  $\text{Path}_{G,v}$  is given inductively by

- $\text{Path}_{G,v}(\lambda) = v$ ; and
- if  $\text{Path}_{G,v}(x_1 \cdots x_n) = w \neq \emptyset$  and  $e_w$  is the edge list corresponding to node  $w$  of  $G$ , then  $\text{Path}_{G,v}(x_1 \cdots x_n x_{n+1}) = e_w(x_{n+1})$ ; and
- if  $\text{Path}_{G,v}(x_1 \cdots x_n) = \emptyset$ , then  $\text{Path}_{G,v}(x_1 \cdots x_n x_{n+1}) = \emptyset$  also.

Then let us define the *binary-depth* of  $G$  with respect to  $v$ ,  $\text{BinDepth}_G(v)$ , to be the largest  $D \in \mathbb{N}$  such that  $\text{Path}_{G,v}$  is “total” and injective on  $\{0, 1\}^{\leq D}$ , i.e.  $\emptyset \notin \text{Path}_{G,v}(\{0, 1\}^{\leq D})$ , and  $|\text{Path}_{G,v}(\{0, 1\}^{\leq D})| = 2^{D+1} - 1$ . Intuitively explained: when recursively following the first two edges of  $v$  and its descendants, we will find a complete binary tree of depth  $D$ . Note that, although in general, in a scaffold, we can have two distinct paths leading to the same node, our notion of binary-depth requires that all  $2^{D+1} - 1$  different paths in  $\{0, 1\}^{\leq D}$  lead to distinct nodes of  $G$ . If  $\text{BinDepth}_G(v) \geq D$ , we will write  $\text{BinTree}_G(v, D)$  to denote the complete binary tree of depth  $D$ , rooted at  $v$ , obtained by recursively following the first two edges until depth  $D$  is reached.

A scaffolding automaton constructs a scaffold as it processes each new input symbol. We will devise a scaffolding automaton  $\mathcal{A}$  as follows. When  $\mathcal{A}$  is given any binary string  $y \in \{0, 1\}^\ell$ , with

$$\ell = D + 1 + \sum_{n=0}^{2^{D+1}-1} (2|n|_2 + 2) = O(D \cdot 2^D) \quad (1)$$

where  $|n|_2$  is the size of the smallest binary representation of the number  $n$ , then the resulting scaffold will have binary-depth  $\geq D$ , with respect to the first child of the top node. Formally said, the computation  $\mathcal{A}(y) = ((q_0, S_0), \dots, (q_\ell, S_\ell))$  constructs the scaffold  $S_\ell = ([\ell], E_\ell, L_\ell)$  having  $\text{BinDepth}_{S_\ell}(e_\ell(0)) \geq D$ .

Before showing how this is done let us show why it is enough. The language  $H$  will be decided by a scaffolding automaton  $\mathcal{A}'$ , in the following way: as long as  $\mathcal{A}'$  only sees 0s and 1s, it will run the algorithm of the automaton  $\mathcal{A}$ . Besides the labels which  $\mathcal{A}$  places at each node, we also copy the corresponding input bit into that node, i.e. the working alphabet of  $\mathcal{A}'$  will be the product of the working alphabet of  $\mathcal{A}$  with  $\{0, 1\}$ . Then we see our first separator symbol  $\#$ , and we stop running  $\mathcal{A}$ . Let us call  $z$  to the part of the input which precedes the separator symbol. After the separator, we expect to see a string  $p \in \{0, 1\}^*$ , and we interpret  $p$  as if it were a path down the tree which is embedded in the scaffold. As we read the symbols of  $p$ , we thus maintain some edge following down this path. In this way we will traverse some bit positions of  $z$ , and we can see which bits of  $z$  appear in these positions, since we have copied the bits of  $z$  into the labels; then, whenever  $z$  has a 1 at such a position, we enter an accepting state, and whenever  $z$  has a 0, we enter a rejecting state.

When  $|z| = \ell$  as above, we have a full binary tree of depth  $D$ , and thus the strings  $p \in \{0, 1\}^D$  will point to  $2^D$  different positions of  $z$ . These positions are distinct (as required by the definition of binary-depth). Thus there are  $2^{2^D}$  ways of filling such positions with bits. Each such way of filling these positions will give a different  $(H, \ell, D)$ -equivalence class. Hence

$$E_H(\ell, D) \geq 2^{2^D}.$$

Now to construct  $\mathcal{A}$ . The base of the method is similar to how we built a scaffolding automaton for the counting language, in Section 4.2. The scaffold constructed by  $\mathcal{A}$  will be labelled by the sequence

$$(0)_2^r \circ (0)_2 \# (1)_2^r \circ (1)_2 \# \dots (n-1)_2^r \circ (n-1)_2 \# (n)_2^r \circ (n)_2 \# \dots$$

where for each natural number  $k$ ,  $(k)_2$  is its binary representation, and  $(k)_2^r$  is the reverse of its binary representation. The characters  $\#$  and  $\circ$  are being used as separators, so  $\#$  is called the *outer separator*, and  $\circ$  the *inner separator*. It may be worthwhile to actually write it down:

$$0 \circ 0 \# 1 \circ 1 \# 01 \circ 10 \# 11 \circ 11 \# 001 \circ 100 \# 101 \circ 101 \# 011 \circ 110 \# \dots$$

It is not hard to see that such a labelling can be obtained by a scaffolding automaton: the automaton can copy what is before each inner separator symbol  $\circ$  to appear after it in reverse, and then, after writing an outer separator symbol  $\#$ , it can scan the binary representation of the number  $n$ , appearing before the  $\#$ , from the lowest to highest-order bit, and apply the usual algorithm for incrementing a binary number by 1, thus writing down the binary representation of  $n + 1$  in reverse. The nodes of the scaffold are thus divided into blocks, and the  $n$ -th block is of the form  $(n)_2^r \circ (n)_2 \#$ .

We must now explain how the edges of the tree are added to the scaffold. The invariant we would like to preserve at the  $n$ -th block, is the following. Suppose  $x_k \cdots x_1 = (n)_2$  is the binary representation of  $n$ , so that the  $n$ -th block is labelled by

$$x_1 \cdots x_k \circ x_k \cdots x_1 \#$$

Let  $v_1 \cdots v_k$   $r$   $v'_k \cdots v'_1$   $s$  be the nodes of the scaffold that get the labels above, i.e., the nodes of the scaffold corresponding to the  $n$ -th block. Then we would like to maintain the following property:

**Invariant 1.** It will always hold, on every block:

- If  $x_i = 1$  for some  $i \in \{2, \dots, k\}$ , then we will have  $e_{v_i}(0) = e_{v'_i}(0)$ , and  $\text{BinDepth}(e_{v_i}(0)) \geq i - 1$ .
- Furthermore, for distinct  $i, j \in \{2, \dots, k\}$  with  $x_i = x_j = 1$ , the trees  $\text{BinTree}(e_{v_i}(0), i - 1)$  and  $\text{BinTree}(e_{v_j}(0), j - 1)$  are node-disjoint.

I.e., one should think that if  $x_i = 1$ , the first edge leaving  $v_i$  and  $v'_i$  points to the root of the same full binary tree of depth  $i - 1$ . And that the two trees corresponding to different  $v_i$  and  $v_j$  share no node.

For simplicity, let us momentarily ignore the “Furthermore” part of the invariant, and later argue that it will be upheld.

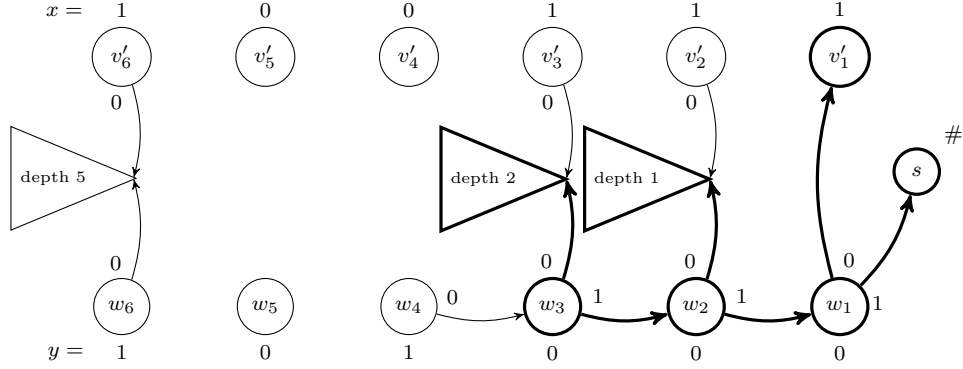
Now suppose that this invariant holds for the  $n$ -th block, let us show how the algorithm needs to behave in order to make it hold for the  $(n + 1)$ -th block. Suppose, for simplicity, that  $n$  and  $n + 1$  are both  $k$ -bit numbers (the case when  $n$  is  $k$ -bits and  $n + 1$  is  $k + 1$  bits is similar). Let  $x_k \cdots x_1 = (n)_2$  and  $y_k \cdots y_1 = (n + 1)_2$  be their binary representations. The algorithm constructs the first half of the  $(n + 1)$ -th block by scanning backwards the second half of the  $n$ -th block.

So, suppose that the second half of the  $n$ -th block has nodes  $v'_k \cdots v'_1$ , which are labelled  $x_k \cdots x_1$ , respectively. Let  $s$  be the node which is labeled by the outer separator  $\#$  between blocks  $n$  and  $n + 1$ . Suppose that the

algorithm is about to add the nodes  $w_1 \cdots w_k$  to the first half of the  $(n+1)$ -th block, and intends to write the labels  $y_1 \cdots y_k$  into them. This is done by reading  $x_k \cdots x_1$  backwards: when the algorithm writes the label  $y_1$  into  $w_1$ , he has an edge pointing to  $v'_1$  where he can read  $x_1$ , when he writes  $y_2$  into  $w_2$  he has an edge pointing to  $v'_2$ , which is labelled by  $x_2$ , and so on. Such “backwards scanning” is easy to do provided we maintain an edge at each node which points to the previous node. The algorithm will also maintain an edge pointing to  $s$ .

When incrementing  $x_1 = 0$ , then we will have  $y_1 = 1$ , and so we must make sure that  $e_{w_1}(0)$  has binary-depth  $\geq 0$ : this is easily ensured by letting  $e_{w_1}(0) = s$ ,  $e_{w_1}(1) = \emptyset$ .

When incrementing  $x_1 = 1$ , we will have  $y_1 = 0$ ; in this case we set  $e_{w_1}(0) = v'_1$ , and also set  $e_{w_1}(1) = s$ . It now holds that  $\text{BinDepth}(w_1) \geq 1$ , and we will use this as the base case of an induction on the length of the 1-prefix of  $x$ . This is illustrated in the figure below, for block number  $n = 39$ , so that  $(n)_2 = x_6 \cdots x_1 = 100111$ . So suppose that  $x_{i-1} = 1, \dots, x_1 = 1$  are the labels of  $v'_{i-1}, \dots, v'_1$ , and that we have written  $y_1 = 0, \dots, y_{i-1} = 0$  as the labels of  $w_1, \dots, w_{i-1}$ . We are about to add the node  $w_i$  to the first half of the  $(n+1)$ -th block, using our pointer to  $v'_i$  in the second half of the  $n$ -th block. Suppose by induction that  $\text{BinDepth}(w_{i-1}) \geq i-1$ . Now look at  $x_i$ . If we are not finished with the 1-prefix of  $x$ , i.e. if  $x_i = 1$ , then we must set  $y_i = 0$ . Our invariant for the previous block tells us that  $\text{BinDepth}(e_{v'_i}(0)) = i-1$ , and our induction hypothesis gives us  $\text{BinDepth}(w_{i-1}) = i-1$ . So we create the new top node  $w_i$  with  $e_{w_i}(0) = e_{v'_i}(0)$  and  $e_{w_i}(1) = w_{i-1}$ , so that  $\text{BinDepth}(w_i) = i$ . This satisfies our induction hypothesis. This case pertains to nodes  $w_2$  and  $w_3$  of the figure below. If we have reached the point where the carry stops, i.e., if  $x_i = 0$ , then we will set  $y_i = 1$ , and for this we create the new top  $w_i$  and set  $e_{w_i}(0) = w_{i-1}$ ,  $e_{w_i}(1) = \emptyset$ . This satisfies our invariant for the first half of  $(n+1)$ -th block (there is no carry in this case). This case pertains to node  $w_4$  of the figure below. Notice how  $\text{BinDepth}(w_3) = 3$ , i.e., we have a complete binary tree of depth 3 rooted at  $w_3$ , which we have drawn in thicker lines for emphasis.



Once we find the first  $x_i = 0$ , we proceed by copying the remaining nodes and their edges; i.e. we set  $y_i = x_i$ ,  $e_{w_i}(0) = e_{v'_i}(0)$ ,  $e_{w_i}(1) = e_{v'_i}(1)$ , until we find the inner separator  $\circ$ . After the inner separator  $\circ$ , we simply copy what we have done, i.e. we set  $y'_i = y_i$ ,  $e_{w'_i}(0) = e_{w_i}(0)$ ,  $e_{w'_i}(1) = e_{w_i}(1)$ , until we find the outer separator  $\#$ .

To keep things simple we have not considered the “Furthermore” part, so let us deal with it now. We have  $y_1 = 0, \dots, y_{i-1} = 0, y_i = 1$  for some  $i$ , which is the last point reached by the carry. Now notice that  $\text{BinTree}(w_{i-1}, i-1)$  (which is the tree under  $w_3$  in the figure above) is made from “fresh” nodes, which did not previously belong to a tree, namely  $w_1, \dots, w_{i-1}$ ,  $s$ , and  $v'_1$ , together with the sub-trees  $\text{BinTree}(e_{v'_j}(0), j-1)$ , for  $1 < j < i$ . These subtrees are, by the “furthermore” part of the invariant, disjoint from any sub-trees  $\text{BinTree}(e_{v'_j}(0), j-1)$  with  $j \geq i$ . Hence  $\text{BinTree}(w_{i-1}, i-1)$  will also be disjoint from  $\text{BinTree}(e_{w_j}(0), j-1)$ , for  $j \geq i$ .

The result of the above is that block number  $2^{D+1}$  will have the labels

$$0^D 1 \circ 10^D \#$$

and if we let  $v$  be the node which is labelled by the first 1 appearing in this block, then we will have  $\text{BinDepth}(v) = D$ . The expression (1) for  $\ell$  is simply the position of the input bit corresponding to the node  $v$ : we have  $2^{D+1} - 1$  many blocks before we reach the  $2^{D+1}$ -th block, and the  $n$ -th block has size  $2|n|_2 + 2$ ; then we have the  $D + 1$  symbols  $0^D 1$ , the last of which is at the position when the node  $v$  is the top of the scaffold.  $\square$

## Acknowledgement

Bruno Loff is the recipient of FCT postdoc grant number SFRH/BPD/116010/2016. This work is partially funded by the ERDF through the COM-

PETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the FCT as part of project UID/EEA/50014/2013. This work was partially supported by CMUP (UID/MAT/00144/2019), FCT (Portugal), FEDER and PT2020. The authors would like to thank Markus Holzer, Martin Kutrib, Leen Torenvliet and Jurgen Vinju for fruitful discussions on this subject.

## Bibliography

### References

- [1] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: ACM SIGPLAN Notices, Vol. 39, ACM, 2004, pp. 111–122.
- [2] A. Birman, J. D. Ullman, Parsing algorithms with backtrack, in: 11th Annual Symposium on Switching and Automata Theory, IEEE, 1970, pp. 153–174.
- [3] A. Birman, The TMG recognition schema, Ph.D. thesis, Princeton (1970).
- [4] A. V. Aho, J. D. Ullman, The theory of parsing, translation, and compiling, Vol. 1, Prentice-Hall, 1972.
- [5] N. Chida, K. Kuramitsu, Linear Parsing Expression Grammars, in: LATA, Vol. 10168 of LNCS, 2017, pp. 275–286.
- [6] T. Garnock-Jones, M. Eslamimehr, A. Warth, Recognising and generating terms using derivatives of Parsing Expression Grammars, CoRR abs/1801.10490 (2018).
- [7] F. Henglein, U. T. Rasmussen, PEG parsing in less space using progressive tabling and dynamic analysis, in: PEPM, ACM, 2017, pp. 35–46.
- [8] K. Mizushima, A. Maeda, Y. Yamaguchi, Packrat parsers can handle practical grammars in mostly constant space, in: PASTE, ACM, 2010, pp. 29–36.
- [9] A. Moss, Derivatives of Parsing Expression Grammars, AFL 252 (2017) 180–194.
- [10] R. R. Redziejowski, From EBNF to PEG, Fundam. Inform. 128f (1-2) (2013) 177–191.



- [11] R. R. Redziejowski, Cut Points in PEG, *Fundamenta Informaticae* 143 (1-2) (2016) 141–149.
- [12] R. R. Redziejowski, Trying to understand PEG, *Fundam. Inform.* 157 (4) (2018) 463–475.
- [13] R. Becket, Z. Somogyi, DCGs + Memoing = Packrat Parsing but Is It Worth It?, *PADL* 4902f (2008) 182–196.
- [14] R. Grimm, Better extensibility through modular syntax, *ACM SIGPLAN Notices* 41 (6) (2006) 38–51.
- [15] R. Ierusalimschy, A text pattern-matching tool based on Parsing Expression Grammars, *Softw. Pract. Exper.* 39 (3) (2009) 221–258.
- [16] A. Koprowski, H. Binsztok, TRX: A formally verified parser interpreter, *Logical Methods in Computer Science* 7f (2) (2011).
- [17] K. Kuramitsu, Fast, flexible, and declarative construction of abstract syntax trees with PEGs, *Journal of Information Processing* 24 (1) (2016) 123–131.
- [18] N. Laurent, K. Mens, Parsing expression grammars made practical, in: *SLE*, ACM, 2015, pp. 167–172.
- [19] A. M. Maidl, F. Mascarenhas, S. Medeiros, R. Ierusalimschy, Error reporting in Parsing Expression Grammars, *Sci. Comput. Program.* 132 (2016) 129–140.
- [20] T. Matsumura, K. Kuramitsu, A declarative extension of parsing expression grammars for recognizing most programming languages, *Journal of Information Processing* 24 (2) (2016) 256–264.
- [21] S. Medeiros, R. Ierusalimschy, A parsing machine for PEGs, *DLS* (2008) 2.
- [22] B. Ford, The packrat parsing and parsing expression grammars page, Online at <http://bford.info/packrat/>.
- [23] G. L. Steele, Growing a language, *Higher-Order and Symbolic Computation* 12 (3) (1999) 221–236.
- [24] C. Flood, Fortress: A next-generation programming language brought to you by Sun Labs, Java One developer conference (2008).

- [25] Y. Bar-Hillel, M. Perles, E. Shamir, On formal properties of simple phrase structure grammars, in: Y. Bar-Hillel (Ed.), *Language and Information: Selected Essays on their Theory and Application*, Series in Logic, Addison-Wesley, 1964, pp. 116–150.
- [26] M. Li, P. Vitányi, A new approach to formal language theory by Kolmogorov complexity, *SIAM Journal on Computing* 24 (2) (1995) 398–410.
- [27] T. Hayashi, On derivation trees of indexed grammars: an extension of the uvwxy-theorem, *Publications of the Research Institute for Mathematical Sciences, Kyoto University* 9 (1973) 61–92.
- [28] S. Yu, A pumping lemma for deterministic context-free languages, *Information Processing Letters* 31 (1) (1989) 47–51.
- [29] A. Amarilli, M. Jeanmougin, A proof of the pumping lemma for context-free languages through pushdown automata, *arXiv:1207.2819* (2012).
- [30] S. A. Greibach, The hardest context-free language, *SIAM Journal on Computing* 2 (4) (1973) 304–310.
- [31] J. Hartmanis, R. E. Stearns, On the computational complexity of algorithms, *Transactions of the American Mathematical Society* 117 (1965) 285–306.
- [32] B. Ford, Packrat parsing: simple, powerful, lazy, linear time, functional pearl, in: M. Wand, S. L. P. Jones (Eds.), *(ICFP '02)*, ACM, 2002, pp. 36–47.
- [33] B. Ford, *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking*, Massachusetts Institute of Technology (2002).
- [34] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to automata theory, languages, and computation, *ACM Sigact News* 32 (1) (2001) 60–65.
- [35] J. L. Balcázar, J. Díaz, J. Gabarró, *Structural Complexity I*, Springer, 1988.
- [36] S. Arora, B. Barak, *Computational complexity: a modern approach*, Cambridge University Press, 2009.
- [37] D. S. Johnson, A catalog of complexity classes, in: *Algorithms and complexity*, Elsevier, 1990, pp. 67–161.

- [38] M. Sipser, Introduction to the Theory of Computation, 3rd Edition, Cengage Learning, 2012.
- [39] A. Rosenberg, Real-time definable languages, Journal of the ACM 14 (4) (1967) 645–662.