Regular Expressions with Backreferences and Lookaheads Capture NLOG

Yuya Uezato
CyberAgent Inc., Japan
National Institute of Informatics, Japan

Abstract

Backreferences and lookaheads are vital features to make classical regular expressions (REGEX) practical. Although these features have been widely used, understanding of the unrestricted combination of them has been limited. Practically, most likely, no implementation fully supports them. Theoretically, while some studies have addressed these features separately, few have dared to combine them. Those few studies showed that the amalgamation of these features significantly enhances the expressiveness of REGEX. However, no acceptable expressivity bound for REWBLk—REGEX with backreferences and lookaheads—has been established. We elucidate this by establishing that REWBLk coincides with **NLOG**, the class of languages accepted by log-space nondeterministic Turing machines (NTMs). In translating REWBLk to log-space NTMs, negative lookaheads are the most challenging part since it essentially requires complementing log-space NTMs in nondeterministic log-space. To address this problem, we revisit Immerman–Szelepcsényi theorem. In addition, we employ log-space nested-oracles NTMs to naturally handle nested lookaheads of REWBLk. Utilizing such oracle machines, we also present the new result that the membership problem of REWBLk is **PSPACE**-complete.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Complexity classes

Keywords and phrases Regular Expression, Automata Theory, Nondeterministic Log-Space

Digital Object Identifier 10.4230/LIPIcs.ICALP.2024.150

Category Track B: Automata, Logic, Semantics, and Theory of Programming

Funding This work was supported by JST, CREST Grant Number JPMJCR21M3.

Acknowledgements I thank anonymous reviewers for their detailed comments on a previous version of this paper. I also sincerely thank anonymous reviewers for their careful reading and invaluable comments on the current version. All comments helped me to significantly improve the presentation of this paper and the clarity of proofs and constructions.

1 Introduction

Backreferences and lookaheads are practical extensions for classical regular expressions (REGEX). REGEX with backreferences—*REWB*—can represent the non context-free language $L = \{w\#w : w \in \{a, b\}^*\}$ with the REWB expression $E = ((a + b)^*)_x \ \# \ \backslash x$. Roughly telling about this expression, we save a substring matched with $(a + b)^*$ into the variable x, and later, we refer back to the matched string using $\backslash x$; therefore, E represents L. REWB is a classical calculus [2], and there are some results:

1. Schmid showed that **REWB**, the class of languages accepted by REWB, is contained in **NLOG**, the class of languages accepted by log-space nondeterministic Turing machines (NTMs) [24, Lemma 18].
2. Recently, Nogami and Terauchi showed that **REWB** is contained in the class of indexed languages, **IL** (an extension of context-free languages [1]) [21].

3. The membership problem of REWB—for an input REWB expression E and an input word w, deciding if E accepts w—is **NP**-complete [2].

On REGEX with lookaheads, there are two kinds of lookaheads. A *positive* lookahead $?(E)$ checks if the rest of the input can be matched by E *without* consuming the input. A *negative* lookahead $!(E)$ checks if the rest of the input *cannot* be matched by E without consuming the input. For example, the expression $?(E)F + !(E)G$ runs an expression F if E goes well with the rest of the input and runs an expression G otherwise. Thus, it can be read as `if` E `then` F `else` G and is helpful in writing practical applications. Although lookaheads are useful, they do not alter the REGEX expressiveness. This fact immediately follows from the result of alternating Turing machines that the language class of alternating finite automata corresponds to that of usual finite automata [6].

Now, it is a natural question: *How expressive are REWB with lookaheads?*

The class REWB with lookaheads—REWBLk—was studied by Chida and Terauchi [7, 8]. They have shown (a) **REWBLk**, the class of languages accepted by REWBLk, is closed under intersection and complement (it is immediately shown using lookaheads); (b) REWB is a proper subclass of REWB(+)—REWB with positive lookaheads—and REWB(−)—REWB with negative lookaheads; (c) the language emptiness problem of REWB(−) is undecidable. They also developed a new class of automata called *positive lookaheads memory automata (PLMFA)* and proved that REWB(+) equals PLMFA.

However, the following two key questions remain unresolved:

1. Which known language classes are related to **REWBLk**?
2. What is the computational complexity of the membership problem of REWBLk—a problem deciding if E accepts w for an input expression E and an input string w?

We solve these questions by presenting the following tight results:

(I) **REWBLk** = **NLOG**. Besides, **REWB** already contains an **NLOG**-complete language.
(II) The membership problem of REWBLk is **PSPACE**-complete.

Together with existing results, our results are summarized in the following table:

	Language Class	Membership Problem
REGEX + lookaheads	= **Regular** [6]	**P**-c [18]
REWB	$\subseteq$ **IL** [21], incomparable to **CFL** [4, 5], $\subseteq$ **NLOG** [24], $\ni$ **NLOG**-c language (I)	**NP**-c [2]
REWBLk	= **NLOG** (I)	**PSPACE**-c (II)

where **NLOG**-c is short for **NLOG**-complete, and the same applies to the others.

1.1 Difficulty in Translating REWBLk to Log-space NTMs

To investigate a hard part of translation from REWBLk to log-space NTMs, let us consider the following language, which is a well-known **NLOG**-complete language:

$$L_{\text{reach}} = \{s\#x_1 \to y_1\# \cdots \#x_n \to y_n\#t : s, t, x_i, y_i \in V^*, \text{and there is a path from } s \text{ to } t\}$$

where the part $\#x_1 \to y_1\# \cdots \#x_n \to y_n\#$ means the directed graph with direct edges $x_1 \to y_1$, $x_2 \to y_2$, and so on. The following REWBLk expression E_{reach} recognizes L_{reach}:

$$E_{\text{reach}} = (V^*)_{\text{CUR}}\,\#\,\left(?(\Sigma^*\,\#\,\backslash\text{CUR} \to (V^*)_{\text{CUR}}\,\#)\right)^*\,\Sigma^*\,\#\,\backslash\text{CUR}$$

where $\Sigma = V \cup \{\#, \rightarrow\}$. It first captures s into the variable Cur and walks on graphs while repeatedly evaluating the part $?(\cdots)$. Each evaluation makes a nondeterministic one-step move on the graph. The part $\Sigma^* \# \setminus \textsc{Cur}$ checks if we reach the goal t.

It is not difficult to structurally translate E_{reach} to a log-space NTM M such that $L(M) = L(E_{\text{reach}})$. Now, using M, let us translate the if-then-else expression $?(E_{\text{reach}})F + !(E_{\text{reach}})G$ for some expressions F and G to a log-space NTM N. Let w be an input word $s\#edges\#t$. For the part $?(E_{\text{reach}})F$, we run M in N; if M accepts w, we proceed to simulate F.

However, for the other part $!(E_{\text{reach}})G$, we encounter problems:

(A) We need to check if all possible walks of M starting from s do not reach t.

(B) Walking paths starting from s and aiming for t become infinitely long, and thus there are infinitely many walks (branching) to be checked.

Therefore, we cannot run M directly in N to handle the negative lookahead $!(E_{\text{reach}})$.

Our Idea: Immerman–Szelepcsényi Theorem & Log-space Nested-oracles NTMs

To address the above problems, we leverage Immerman–Szelepcsényi theorem [16, 28]. This theorem states that the class **NLOG** is closed under complement; i.e., there exists a log-space NTM $\overline{M}$ such that $L(\overline{M}) = \Sigma^* \setminus L(M)$. Therefore, in our machine N, we run $\overline{M}$ for the part $!(E_{\text{reach}})G$: If $\overline{M}$ eventually accepts w, then we proceed to simulate G.

On the other hand, REWBLk permits nested lookaheads, such as $?(\cdots!(\cdots?(\cdots)\cdots)\cdots)$. To handle them naturally, we employ log-space NTMs with *nested oracles* [16, 26, 19]. These machines can easily simulate REWBLk and are translated to log-space NTMs by Immerman–Szelepcsényi theorem; thus, **REWBLk** = **NLOG** holds. Moreover, oracle machines are crucial to showing that the membership problem of REWBLk is **PSPACE**-complete. To this end, we also give the new result that the membership problem of such machines is in **PSPACE**.

Structure of Paper

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 reviews REWBLk and demonstrates that **REWB** already contains an **NLOG**-complete language. Section 4 illustrates the expressiveness of REWBLk: (1) **NLOG** $\subseteq$ **REWBLk**; (2) the membership problem of REWBLk is **PSPACE**-hard; (3) REWB(+) and REWB(−) represent languages $\notin$ **IL**; and (4) the emptiness problems of REWB(+) and REWB(−) are undecidable even if $\Sigma = \{a\}$. Section 5 reviews log-space nested-oracles NTMs and their language class (= **NLOG**), and shows our new result: their membership problem is in **PSPACE**. Section 6 establishes **REWBLk** $\subseteq$ **NLOG** and that the membership problem of REWBLk is in **PSPACE**. Section 7 concludes this paper by giving open problems.

2 Related Work

As discussed in the Introduction, Chida and Terauchi have formalized REWBLk and its semantics [7, 8]. To our knowledge, their study is the first theoretical exploration into the simultaneous treatment of backreferences and lookaheads. Surprisingly, there has been no prior theoretical research on the topic despite their longstanding and widespread use. They introduced PLMFA (positive lookahead MFA) by expanding MFA (memory finite automata), which Schmid presented for studying REWB in [25]. One of their main results is the equivalence of PLMFA to REWB(+), established through translations between PLMFA and REWBLk. Nevertheless, they did not address: (i) a relationship between REWBLk

and existing known language classes; (ii) the complexity of the membership problem of REWBLk. In contrast, we show that REWBLk captures **NLOG** and the membership problem of REWBLk is **PSPACE**-complete.

As highlighted in the Introduction, for REWB, Schmid showed **REWB** $\subseteq$ **NLOG** [24], and Nogami and Terauchi showed **REWB** $\subseteq$ **IL** [21]. Schmid also introduced MFA and showed that MFA corresponds to REWB [25]. About the relationship between **NLOG** and **IL**, it is worth noting that:

- **NLOG** $\not\subseteq$ **IL**. It is shown as follows. The language $L_{2^{2^n}} = \{a^{2^{2^n}} : n \in \mathbb{N}\}$ clearly belongs to **NLOG**. However, $L_{2^{2^n}} \notin$ **IL** by pumping lemmas for indexed languages [14, 11].
- **IL** $\not\subseteq$ **NLOG** unless **NLOG** = **NP**. It is shown as follows. **IL** can represent the language $L_{3\mathrm{SAT}}$, whose words are true 3-SAT formulas [23]. However, $L_{3\mathrm{SAT}} \notin$ **NLOG** unless **NLOG** = **NP**.

In the present paper, we take their result further and show that **REWBLk** = **NLOG** and that **REWB** already contains an **NLOG**-complete language.

We also refer to modern REGEX engines that (partially) support both backreferences and lookaheads. Several programming languages (for example, Perl, Python, PHP, Ruby, and JavaScript) and .NET framework support these features. However, their support is limited in both syntax and semantics. First, expressions like $(\backslash x \backslash x)_x$ and $(\backslash x + \backslash x)_x$ are rejected by most implementations because the variable x appears more than once in single captures. Next, in most implementations, the expression $F = (?((\backslash x a)_x))^* \backslash x$, which represents $\{\epsilon, a, a^2, \ldots\}$, does not match with a^2 and a^3, so on. It is due to conservative loop-detecting semantics. Such a semantics is standardized in ECMAScript [9]. This semantics works for F as follows. First, it unfolds the Kleene-* of $(?((\backslash x a)_x))^*$ as $(\epsilon + \underline{?((\backslash x a)_x)}(?((\backslash x a)_x))^*)$. Next, it enters the underline part and updates the variable x without consuming any input characters. Then, it tries to evaluate $(?((\backslash x a)_x))^*$ *again* at the same input position. At this point, many REGEX engines think that we enter an infinite loop and so stop unfolding the Kleene-*. Consequently, F only matches with ϵ (without loop unfolding) and a (with a single loop unfolding).

We can rephrase this situation as follows: (1) the amalgamation of lookaheads with variables induces side effects without consuming any characters; (2) however, the loop-detecting semantics overlooks such side effects and changes behaviors from the naive semantics. On the other hand, such conservative semantics work well for REGEX, REGEX with lookaheads, and REWB since they do not induce such side effects.

This paper presents a translation between REWBLks and log-space NTMs, enabling the development of REGEX engines that fully support backreferences and lookaheads. Notably, such engines run in *polynomial time* (for a fixed expression) since **NLOG** $\subseteq$ **P**.

3 Preliminaries: REWBLk

We review the syntax and semantics of REWBLk [7, 8] step-by-step below.

3.1 Regular Expressions with Backreferences and Lookaheads

We first give the syntax of REWBLk over an alphabet Σ and variables $\mathcal{V}$:

$$\begin{aligned} E \;::=\; & \sigma \mid \epsilon \mid E + E \mid E\,E \mid E^* \\ & \mid \underbrace{(E)_v}_{\text{capture}} \mid \underbrace{\backslash v}_{\text{backreference}} \mid \underbrace{?(E)}_{\text{positive lookahead}} \mid \underbrace{!(E)}_{\text{negative lookahead}} \end{aligned}$$

where $\sigma \in \Sigma$ and $v \in \mathcal{V}$ is a variable. The first line defines classical regular expressions, REGEX. We consider the following subclasses in this paper: *REWB* (REGEX with captures and backreferences), *REWB(+)* (REWB with positive lookaheads $?(E)$), *REWB(−)* (REWB with negative lookaheads $!(E)$).

Semantics of REGEX

We first give a semantics for REGEX. To accommodate variables and lookaheads, configurations for REGEX are 4-tuples $\langle E, w, p, \Lambda\rangle$ where

- E is an expression to be executed;
- w is an input string and will not change throughout computation;
- p is a 0-origin position on w ($0 \le p \le |w|$). We write $w[p]$ for the symbol on the position p. It should be noted that $p = |w|$ is allowed to represent that we consume all the input.
- $\Lambda : \mathcal{V} \to \Sigma^*$ is an assignment from variables $\mathcal{V}$ to substrings of w.

We write $\mathcal{C}$ for the set of configurations. To denote all the results obtained by computing, we use a semantic function $[\![\cdot]\!] : \mathcal{C} \to \mathcal{P}(\mathbb{N} \times (\mathcal{V} \to \Sigma^*))$ where $\mathcal{P}(X)$ is the power set of X. On $[\![\langle E, w, p, \Lambda\rangle]\!] = \{\langle p_1, \Lambda_1\rangle, \ldots, \langle p_n, \Lambda_n\rangle\}$, each pair $\langle p_i, \Lambda_i\rangle$ means that, after executing E on w from p under Λ, we move to the position p_i and obtain an assignment Λ_i. On the basis of this idea, we define a semantics for each rule of the REGEX part:

$$[\![\langle \sigma, w, p, \Lambda\rangle]\!] = \begin{cases} \{\langle p+1, \Lambda\rangle\} & \text{if } p < |w| \text{ and } w[p] = \sigma, \\ \emptyset & \text{otherwise,} \end{cases} \qquad [\![\langle \epsilon, w, p, \Lambda\rangle]\!] = \{\langle p, \Lambda\rangle\},$$

$$\begin{aligned} [\![\langle E_1 + E_2, w, p, \Lambda\rangle]\!] &= [\![\langle E_1, w, p, \Lambda\rangle]\!] \cup [\![\langle E_2, w, p, \Lambda\rangle]\!], \\ [\![\langle E_1 E_2, w, p, \Lambda\rangle]\!] &= \bigcup_{\langle p', \Lambda'\rangle \in [\![\langle E_1, w, p, \Lambda\rangle]\!]} [\![\langle E_2, w, p', \Lambda'\rangle]\!], \\ [\![\langle E^*, w, p, \Lambda\rangle]\!] &= \bigcup_{i=0}^{\infty} [\![\langle E^i, w, p, \Lambda\rangle]\!] \quad \text{where } E^0 = \epsilon,\ E^i = \overbrace{EE\cdots E}^{i}. \end{aligned}$$

We note that our semantic function $[\![\langle E, w, p, \Lambda\rangle]\!]$ is inductively defined on the lexicographic ordering over the star height of E and the expression size of E. The start height and expression size of REWBLK is defined in the usual way.

We also note that each $[\![\langle E, w, p, \Lambda\rangle]\!]$ forms a finite set because the value of each variable x must be a substring of w, and also p is bounded as $0 \le p \le |w|$.

Semantics of REWB

A capture expression $(E)_x$ attempts to match the input string with E. If it succeeds, the matched substring is stored in the variable x.

$$[\![\langle\, (E)_x, w, p, \Lambda\,\rangle]\!] = \{\langle\ p', \Lambda'[\, x \mapsto w[p..p')\,]\ \rangle : \langle p', \Lambda'\rangle \in [\![\langle E, w, p, \Lambda\rangle]\!]\}$$

where $w[p..q)$ is the string $w[p]\, w[p+1] \cdots w[q-1]$.

A backreference $\backslash x$ refers to the substring stored previously by evaluating some $(E)_x$.

$$[\![\langle \backslash x, w, p, \Lambda\rangle]\!] = [\![\langle \Lambda(x), w, p, \Lambda\rangle]\!].$$

Semantics of Lookaheads

Positive lookaheads $?(E)$ run E from the current input without consuming any input. Although the change in head position is undone after running E, the modification to variables in E is not. So, we can also call it destructive lookahead.

$$[\![\langle ?(E), w, p, \Lambda\rangle]\!] = \{\langle p, \Lambda'\rangle : \langle p', \Lambda'\rangle \in [\![\langle E, w, p, \Lambda\rangle]\!]\}.$$

Negative lookaheads $!(E)$ also run E without consuming any input. If E does not match anything, we invoke a continuation. Compared with positive lookaheads $?(E)$, both the previous head position and the previous values of variables are recovered.

$$[\![\langle !(E), w, p, \Lambda\rangle]\!] = \begin{cases} \{\langle p, \Lambda\rangle\} & \text{if } [\![\langle E, w, p, \Lambda\rangle]\!] = \emptyset, \\ \emptyset & \text{otherwise.} \end{cases}$$

Readers may wonder why positive lookaheads modify variables while negative ones do not. This asymmetry arises because, in negative lookaheads, all computations uniformly fail, leaving no suitable configuration for altering variables. On the other hand, using the non-destructive property of the negative lookaheads, we can also define non-destructive positive lookaheads by $!(!(E))$. This expression executes E from the current position without any variable modifications.

Remark: Special character \$ Using a negative lookahead, we define $\$ = !(\Sigma)$ to check the end of the input. \$ is one of the most important applications of negative lookaheads.

▶ **Definition 1.** *The language of a REWBLK E, $L(E)$, is defined as follows:*

$$L(E) = \{w \in \Sigma^* : \langle p, \Lambda\rangle \in [\![\langle E, w, 0, \iota\rangle]\!],\ p = |w|\} \quad \textit{where } \forall x \in \mathcal{V}.\, \iota(x) = \epsilon.$$

We can also consider another definition using $\gamma(x) = \perp$ instead of ι, where γ indicates that all variables are initially undefined. Although some real-world regular expression engines adopt that definition, we adopt the above ι-definition since it is tedious to initialize all variables x using $(\epsilon)_x$. The results discussed in this paper will not change regardless of which one is used. There is another formalization that excludes labels that appear multiple times in a single group, for instance $(\backslash x \backslash x)_x$. We discuss such a restriction, which we call reference restriction, in the immediately following section.

3.2 Reference Restriction and Normalization

Our formalization of REWBLK allows that variable references appear inside their definitions; for example, we allow expressions like $(\backslash x \backslash x)_x$. On the other hand, many studies on REWB do not allow them; i.e., expressions like $(\backslash x \backslash x)_x$ are prohibited [5, 24, 10, 21].

To the best of the author's knowledge, it remains unclear whether the restriction alters the expressive power of REWB. However, on REWBLK, the restriction does not change the language class of REWBLK. Here, we formalize the restriction and then present our normalization, which converts REWBLK expressions to language-equivalent restricted forms.

We use the function VAR that receives an expressions E and returns the set of all the backreference variables inside E. For example,

$$\textsc{Var}(\ \backslash x \backslash x\ ) = \{x\},\ \textsc{Var}(\ (\backslash x \backslash y)_z\ ) = \{x, y\},\ \textsc{Var}(\ a\, b\, c\, ?(\ (\backslash w (\backslash x)_z)_y\ )\ ) = \{w, x\}.$$

It can be easily defined inductively on the expression size of E.

We also define a restriction, *reference restriction*. An REWBLK expression E satisfies the reference restriction condition if, for all the capture subexpressions $(F)_x$ of E, $x \notin \textsc{Var}(F)$ holds. For example, the expression $a\,b\,c?((\backslash w(\backslash x)_z)_y)$ satisfies the condition. On the other hand, the expressions $(\backslash x\backslash x)_x$ and $a\,b\,c?((\backslash w(\backslash y)_z)_y)$ do not.

▶ **Theorem 2.** *For any REWBLK expression E, there is an expression E' that satisfies the reference restriction condition and $L(E) = L(E')$.*

Proof. It suffices to perform variable renaming like alpha-conversion in lambda calculus to remove patterns $(\cdots x \cdots)_x$. Formally, we apply the following renaming function $\mathcal{R}$ from the innermost of E to the outermost:

$$\mathcal{R}(F) = \begin{cases} ?((F')_y)(\backslash y)_x & \text{where } y \text{ is a fresh variable} \quad \text{if } F = (F')_x, \\ F & \text{otherwise.} \end{cases}$$

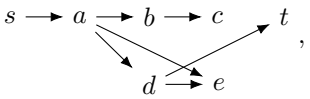
The expression $?((F')_y)(\backslash y)_x$ is equivalent to $(F')_x$ because it first stores the result of running F' to a fresh variable y and then puts it to the original variable x. Thus, the innermost-to-outermost applying $\mathcal{R}$ changes expressions to ones, which satisfy the restriction condition, without change their languages.

For example, an expression $(aa)_x(\backslash x\backslash x)_x$ is translated to $(aa)_x?((\backslash x\backslash x)_y)(\backslash y)_x$ by introducing a variable y. Another expression $(a)_x(b)_y(?[(\backslash x\backslash x)_x]aa\backslash y)_x$ is first translated to $(a)_x(b)_y(?[?((\backslash x\backslash x)_\alpha)(\backslash\alpha)_x]aa\backslash y)_x$ by introducing α. Then, introducing β, it is translated to $(a)_x(b)_y?((?[?((\backslash x\backslash x)_\alpha)(\backslash\alpha)_x]aa\backslash y)_\beta)(\backslash\beta)_x$. ◀

We will use this theorem in Section 6 to translate REWBLK to nested-oracles machines.

3.3 NLOG-complete Language Accepted by REWB

Hartmanis and Mahaney proposed a decision problem called *TAGAP*, which is the topological sorted version of the reachability problem of directed *acyclic* graphs (DAG) [13]. Let us consider a word $w = \ulcorner x_1 \to y_1 \urcorner\ulcorner x_2 \to y_2 \urcorner \cdots \ulcorner x_n \to y_n \urcorner$, which represents a DAG. We call w *topologically sorted* if: for all pairs of $a \to b$ and $b \to c$ of w, $a \to b$ appears before $b \to c$ in w. The following example represents a DAG and one of its topologically sorted representation:



, $\ulcorner s \to a \urcorner\ulcorner a \to d \urcorner\ulcorner a \to e \urcorner\ulcorner d \to t \urcorner\ulcorner a \to b \urcorner\ulcorner d \to e \urcorner\ulcorner b \to c \urcorner$.

We define the language for TAGAP as follows:

$$L_{\text{TAGAP}} = \{s \# R \# t : R \text{ is a topologically sorted repr. of } G,\ t \text{ is reachable from } s \text{ in } G\}.$$

Hartmanis and Mahaney showed that this language is **NLOG**-complete [13, Theorem 3]. Since we only consider the topologically sorted representation, there is no longer a need to explore the entire input many times. In the above example, we can reach t from s by nondeterministically finding edges $s \to a$, $a \to d$, and $d \to t$ in this order by one-way scanning. Indeed, we can show the following theorem.

▶ **Theorem 3.** $L_{TAGAP} \in \boldsymbol{REWB}$.

Proof. The following expression E_{TAGAP} clearly recognizes L_{TAGAP}:

$$E_{\text{TAGAP}} = (V^*)_{\textsc{Cur}} \,\#\, \left(\Sigma^* \ulcorner \backslash\textsc{Cur} \to (V^*)_{\textsc{Cur}} \urcorner\right)^* \Sigma^* \,\#\, \backslash\textsc{Cur},$$

where V is an alphabet for vertices. ◀

4 Expressiveness of REWBLk

In this section, we present some theorems about the expressiveness of REWBLk.

4.1 Unary Non-Indexed Language

We consider the single exponential numerical language $L_{1exp} = \{a^{2^k} : k \in \mathbb{N}\}$ over the unary alphabet $\Sigma = \{a\}$. The language L_{1exp} is represented by the REWB(+) expression $E_{1exp} = ?(a)_x\,(?(\backslash x \backslash x)_x)^*\,\backslash x$. The part $?(a)_x$ initializes $x = a$ (i.e., $x = a^0$), and the Kleene-$*$ part iteratively doubles x.

Furthermore, we can represent the doubly exponential language $L_{2exp} = \{a^{2^{2^k}} : k \in \mathbb{N}\}$ by the following REWBLk expression E_{2exp}:

$$\begin{aligned} E(\alpha,\beta) &= \big(?[(\backslash\alpha\, a)_\alpha]\ ?[(\backslash\beta\backslash\beta)_\beta]\big)^*, \qquad \text{(it adds } a \text{ to } \alpha \text{ and doubles } \beta\text{)} \\ E_{2exp} &= ?((a)_m)\ E(n,m)\ ?((a)_x)\ E(y,x)\ ?(a^*\,?(\backslash m\$)\,?(\backslash y\$))\ \backslash x. \end{aligned}$$

It searches the numbers n, m, x, y that satisfy $2^n = m$, $2^y = x$, and $m = y$; so, $x = 2^{2^n}$. While unfolding the Kleene-* of $E(n,m)$ and $E(y,x)$, $2^n = m$ and $2^y = x$ hold. The part $?(a^*\,?(\backslash m\$)\,?(\backslash y\$))$ checks if $m =_? y$ by utilizing the negative lookahead expression $\$ = !(a)$.

We emphasize the known result $L_{2exp} \notin \mathbf{IL}$, which is shown by the pumping or shrinking lemma for indexed languages [14, 11]. Since we can carry out a similar construction of E_{2exp} in REWB(+) and REWB(−), we have the following result.

▶ **Theorem 4.** *REWB(+) and REWB(−) can represent unary non-indexed languages.*

Proof (Sketch). Due to the page limitation, we provide a proof sketch for the REWB(−) part and put the complete proof in the Appendix. Let us consider the following REWB(−) expression:

$$\begin{aligned} E'(\alpha,\beta) &= \big((\backslash\alpha\, a)_\alpha\ (\backslash\beta\backslash\beta)_\beta\big)^*, \\ E'_{2exp} &= (a)_m\ E'(n,m)\ (a)_x\ E'(y,x)\ !(!(\backslash m\$))\ !(!(\backslash y\$))\ a^*. \end{aligned}$$

While the expression E'_{2exp} resembles E_{2exp}, it lacks positive lookaheads. Let us explain E'_{2exp} step-by-step:

1. The expressions $(a)_m$ and $(a)_x$ initialize m and x by a as with E_{2exp}.
2. The subexpression $E'(n,m)$ repeatedly expands variables n and m as with E_{2exp}. So, executing $E'(n,m)$ *actually* consumes inputs as follows without positive lookaheads:

$$\underline{a^1}_n\ \underline{a^{2^1}}_m\ \underline{a^2}_n\ \underline{a^{2^2}}_m\ \underline{a^3}_n\ \underline{a^{2^3}}_m \cdots \underline{a^i}_n\ \underline{a^{2^i}}_m \cdots$$

3. The same holds for the expression $E'(y,x)$.

The part $!(!(\backslash m\$))$ is a non-destructive positive lookahead by $\backslash m\$$ that is simulated by double negative lookaheads; therefore, the part $!(!(\backslash m\$))\ !(!(\backslash y\$))$ requires $m = y$. If we pass the assertion, we consume the rest input by a^*. By replacing positive lookaheads with actual consuming, the language $L(E'_{2exp})$ *grows faster* (the notion *fast growth* is formalized in the Appendix) than L_{2exp}. This property implies that $L(E'_{2exp})$ is not an indexed language. ◀

It states that, even if restricted to unary languages, positive or negative lookaheads make REWB expressive and incomparable to **IL**. We can also show the undecidability of the emptiness problem, which is checking if $L(E) = \emptyset$ for a given expression E, of REWB(+) and REWB(−) over $\Sigma = \{a\}$.

▶ **Theorem 5.** *The emptiness problems of REWB(+) over a unary alphabet and REWB(−) over a unary alphabet are undecidable.*

The two undecidability results can be shown by translating the Post Correspondence Problem to the emptiness problems. Due to the page limitation, we put the proof of Theorem 5 in the Appendix.

4.2 Simulating Two-way Multihead Automata by REWBLk

We show that REWBLk can simulate *two-way multihead automata*, which are a classical extension of automata and capture **NLOG** [12].

Simulating Two-way One-head Automata

We start from two-way *one*-head automata. Let $\mathcal{A} = (Q, q_{\text{init}}, q_{\text{acc}}, \Sigma, \Delta)$ be a two-way automata where $Q = \{q_0, q_1, \ldots, q_{|Q|-1}\}$ is a finite set of states, $q_{\text{init}} \in Q$ is the initial state, $q_{\text{acc}} \in Q$ is the accepting state, Σ is an input alphabet, and Δ is a set of transition rules. Each transition rule is of the form $p \xrightarrow{\tau/\theta} q$ where $p, q \in Q$, $\tau \in (\Sigma \cup \{\vdash, \dashv\})$, and $\theta \in \{-1, 0, 1\}$. The component θ indicates a head moving direction: (1) if $\theta = -1$ (resp. $\theta = 1$), we move the scanning head left (resp. right); (2) if $\theta = 0$, we *do not* move the scanning head.

For an input $w \in \Sigma^*$, we run $\mathcal{A}$ on the extended string $\vdash w \dashv$, which are surrounded by the left and right end markers. A configuration of $\mathcal{A}$ for $\vdash w \dashv$ is a tuple (q, i) where $q \in Q$ and $0 \leq i < 2 + |w|$, and thus the current scanning symbol is $(\vdash w \dashv)[i]$.

A transition rule $\delta = p \xrightarrow{\tau/\theta} q \in \Delta$ gives a labelled transition relation $\overset{\delta}{\Rightarrow}$ as follows:

$$(p, i) \overset{\delta}{\Rightarrow} (q, i + \theta) \quad \text{if } (\vdash w \dashv)[i] = \tau \text{ and } 0 \leq i + \theta < |w| + 2$$

The word w is accepted by $\mathcal{A}$ if the initial configuration $(q_{\text{init}}, 0)$, reading $\vdash$, has a computation path to a configuration with the accepting state q_{acc}. We now define the language of $\mathcal{A}$ as follows:

$$L(\mathcal{A}) = \{w : (q_{\text{init}}, 0) \overset{\delta_1}{\Rightarrow} (q_1, i_1) \overset{\delta_2}{\Rightarrow} \cdots \overset{\delta_n}{\Rightarrow} (q_{\text{acc}}, i_n)\}.$$

To simulate $\mathcal{A}$ by REWBLk, we use some variables $L, R, S \in \Sigma^*$. Intuitively, each variable means the following:

- If $L = w$ (resp. $R = w$), $\mathcal{A}$ is located on $\dashv$ (resp. $\vdash$). Otherwise, $\exists \sigma \in \Sigma.\, w = L\sigma R$. So, L (resp. R) means the left (resp. right) part of w.
- The length of S denotes the index i of the current state q_i of $\mathcal{A}$.

We formalize the above intuition as the following simulation $\sim$ between (q, i) and $\langle L, R, S\rangle$:

$$\begin{array}{ll} (q_j, 0) \sim \langle \epsilon, w, S\rangle & \text{if } |S| = j, \\ (q_j, |w| + 1) \sim \langle w, \epsilon, S\rangle & \text{if } |S| = j, \\ (q_j, i) \sim \langle L, R, S\rangle & \text{if } 1 \leq i \leq |w|,\ \exists \sigma.\ w = L\,\sigma\,R,\ |L| = i - 1, \text{ and } |S| = j. \end{array}$$

To represent all states, we need $|w| \geq |Q| - 1$. So, we mainly consider to represent $L(\mathcal{A}) \setminus L'$ where $L' = \{w \in L(\mathcal{A}) : |w| < |Q| - 1\}$. For instance, our simulation proceeds as follows:

(1) $\vdash^{q_0} ab \dashv \sim \langle L = \epsilon,\ R = ab,\ S = \epsilon\rangle$, (if $R = w$, then $L = \epsilon$ and $\mathcal{A}$ is on $\vdash$)
(2) $\vdash a^{q_1} b \dashv \sim \langle L = \epsilon,\ R = b,\ S = a\rangle$, (move right from (1))
(3) $\vdash ab^{q_1} \dashv \sim \langle L = a,\ R = \epsilon,\ S = a\rangle$, (move right from (2))
(4) $\vdash ab \dashv^{q_1} \sim \langle L = ab,\ R = \epsilon,\ S = a\rangle$, (if $L = w$, then $R = \epsilon$ and $\mathcal{A}$ is on $\dashv$)
(5) $\vdash ab^{q_2} \dashv \sim \langle L = a,\ R = \epsilon,\ S = ab\rangle$, (move left from (4) and change q_1 to q_2)
(6) $\vdash a^{q_2} b \dashv \sim \langle L = \epsilon,\ R = b,\ S = ab\rangle$.

As initialization, we use the expression $E_{\text{init}} = (\epsilon)_L\ ?((\Sigma^*)_R\,\$)\ (\epsilon)_S$, which sets $L = S = \epsilon$ and $R = w$ for the input w. To move the head right, we use the following expression:

$$E_{+1} = ?((\backslash L\Sigma)_L\ \Sigma\,(\Sigma^*)_R\,\$) + ?((\backslash L\Sigma)_L\,(\epsilon)_R\,\$).$$

The part $?((\backslash L\Sigma)_L\,\Sigma\,(\Sigma^*)_R\,\$)$ first attempts to expand L toward the right and then updates R. Similarly, we define the expression E_{-1} to move the head left as follows:

$$E_{-1} = ?((\Sigma^*)_L\,\Sigma\,(\Sigma\backslash R)_R\,\$) + ?((\epsilon)_L\,(\Sigma\backslash R)_R\,\$).$$

To check if the current scanning symbol is $\dashv$, $\vdash$, or $\sigma \in \Sigma$, we use the following expressions:

$$E_{\dashv} = ?(\backslash L\$),\ E_{\vdash} = ?(\backslash R\$),\ E_{\sigma} = ?(\backslash L\,\sigma\,\backslash R\$).$$

To check if the current state is q_i, we use the expression $E_{q_i} = ?\big(\Sigma^*\ ?(\backslash S)\$\ ?(\Sigma^i)\$\big)$. To change the current state q_i to q_j, we use the expression $E_{i\text{-to-}j} = E_{q_i}\ ?(\Sigma^*\,(\Sigma^j)_S\,\$)$.

Now, each transition rule δ is simulated by the following expression $E(\delta)$ defined as:

$$E(q_i \xrightarrow{\tau/0} q_j) = E_{i\text{-to-}j}\ E_\tau, \quad E(q_i \xrightarrow{\tau/\theta} q_j) = E_{i\text{-to-}j}\ E_\tau\ E_\theta\ \ (\theta \in \{-1, +1\}).$$

Finally, the following expression $E_{\mathcal{A}}$ simulates $\mathcal{A}$, and $L(E_{\mathcal{A}}) = L(\mathcal{A})$ holds:

$$E_{\mathcal{A}} = E_{L'} + ?(E_{q_{\text{init}}}\,(E(\delta_1) + E(\delta_2) + \cdots + E(\delta_n))^*\,E_{q_{\text{acc}}})\,\Sigma^*$$

where $E_{L'}$ is a regular expression for the finite language L', and $\Delta = \{\delta_1, \delta_2, \ldots, \delta_n\}$.

We summarize the our translation as follows.

▶ **Lemma 6.** *For a two-way one-head automata $\mathcal{A}$, there is an expression $E_{\mathcal{A}}$ with 3 variables (L, R, and S) such that $L(E_{\mathcal{A}}) = L(\mathcal{A})$.* Especially*, the expression uses negative lookaheads only in the form of $\$$.*

Simulating Two-way Multihead Automata

We extend the above argument to two-way *multihead* automata $\mathcal{M}$ [12, 15]. Compared to two-way one-head automata $\mathcal{A}$, $\mathcal{M}$ has multiple-heads on input strings. We write K for the number of heads. The difference between $\mathcal{A}$ and $\mathcal{M}$ are the following:

- Each configuration of $\mathcal{M}$ is a tuple $(q, i_1, i_2, \ldots, i_K)$ where q is the current state and i_j is the j-th head position.
- Each transition rule is $p \xrightarrow{(\tau_1,\ldots,\tau_K)/(\theta_1,\ldots,\theta_K)} q$ where $p, q \in Q$, $\tau_j \in \Sigma \cup \{\vdash, \dashv\}$ is used for inspecting the scanned symbol by j-th head, and θ_j denotes the head moving direction for the j-th head.

We define a transition relation $\Rightarrow$ in the same way as for $\mathcal{A}$. Let $\delta = p \xrightarrow{(\tau_1,\ldots,\tau_K)/(\theta_1,\ldots,\theta_K)} q$ be a rule and $C = (p, i_1, i_2, \ldots, i_K)$ be a valid configuration. If $\forall 1 \le j \le K.\, (\vdash w \dashv)[i_j] = \tau_j$, then we have $C \overset{\delta}{\Rightarrow} (q, i_1 + \theta_1, i_2 + \theta_2, \ldots, i_K + \theta_K)$. We define the language as follows:

$$L(\mathcal{M}) = \{w : (q_{\text{init}}, 0, 0, \ldots, 0) \overset{\delta_1}{\Rightarrow}\overset{\delta_2}{\Rightarrow} \cdots \overset{\delta_n}{\Rightarrow} (q_{\text{acc}}, i_1, i_2, \ldots, i_K)\}.$$

We can show the following lemma by simply extending our above construction for two-way one-head automata.

▶ **Lemma 7.** *For a given two-way K-head automata $\mathcal{M}$, we have an expression $E_{\mathcal{M}}$ with $3K$ variables such that $L(E_{\mathcal{M}}) = L(\mathcal{M})$. $E_{\mathcal{M}}$ uses negative lookaheads only in the form of $\$$.*

Proof. As with the two-way one-head automata, for each i-th head, we prepare variables L_i, R_i, and S_i. For each i-th head, by using the variables for i, we give expressions E^i_π where $\pi \in \{+1, -1, \sigma, \vdash, \dashv\}$. Employing the same E_{q_i} and $E_{i\text{-to-}j}$, we can give $E(\delta)$ for each transition rule δ of $\mathcal{M}$ and so $E_{\mathcal{M}}$. ◀

It is well-known that the class of languages accepted by two-way multihead automata corresponds to **NLOG** [12]; so, we have the following theorem.

▶ **Theorem 8.** **NLOG** $\subseteq$ **REWBLk**.

4.3 True Quantified Boolean Formula

We translate the **PSPACE**-complete problem **TQBF**, checking if a *quantified boolean formula* (QBF) is true, into the membership problem of REWBLK. Here we only consider QBFs in CNF since TQBF restricted to CNF is **PSPACE**-complete [3]. For instance, let us consider the following QBF Q and translate it to the equivalent form Q' by replacing $\forall$ with $\neg\exists\neg$:

$$Q : \forall a.\, \exists b.\, \forall c. \forall d.\, (a \vee b \vee c) \wedge (\overline{b} \vee c \vee d) \Rightarrow Q' : \neg\exists a.(\neg\exists b.(\neg\exists c.(\exists d.(\neg((a \vee b \vee c) \wedge (\overline{b} \vee c \vee d)))))).$$

In order to check if Q is true, we first structurally translate Q' into the following $E_{Q'}$:

$$\begin{aligned} E(v) &= \big((T)_v(F)_{\overline{v}} \;+\; (T)_{\overline{v}}(F)_v\big) \quad \text{where } v \text{ is a propositional variable,} \\ E_{Q'} &= !(E(a)\; !(E(b)\; !(E(c)\; E(d)\; !((\backslash a + \backslash b + \backslash c)(\backslash\overline{b} + \backslash c + \backslash d))))), \end{aligned}$$

where we replace $\neg$ with $!$, $\exists x$ with $E(x)$, x with $\backslash x$, $\overline{x}$ with $\backslash\overline{x}$, and $\vee$ with $+$.

We then check $w = TF\,TF\,TF\,TF\,TT \in_? L(E_{Q'})$. We explain the string w using the annotated version $T_1F_2T_3F_4T_5F_6T_7F_8T_9T_{10}$: (1) the first two characters T_1F_2 makes the two cases where $(a = T, \overline{a} = F)$ or $(a = F, \overline{a} = T)$; (2) similarly, T_3F_4 (resp. T_5F_6 and T_7F_8) works for b and $\overline{b}$ (resp. $c, \overline{c}$ and $d, \overline{d}$) ; (3) by T_9, we check if the expression $(a \vee b \vee c)$ holds (in the negative context); (4) by T_{10}, we also check if $(\overline{b} \vee c \vee d)$ holds. Thus, Q is true iff $w \in L(E_Q)$.

On the basis of the above translation using $E(v)$, we can translate every CNF-QBF Q to the corresponding expression E_Q and give the membership problem $TFTF \ldots TF\,TT \ldots T \in_? L(E_Q)$ in polynomial time for the size of Q. It implies the following result.

▶ **Theorem 9.** *The membership problem of REWBLK is* ***PSPACE****-hard.*

5 Log-space Nested-Oracles Nondeterministic Turing Machines

As we have stated in the Introduction, we utilize log-space nested-oracles NTMs. We will translate REWBLK to them in the next section.

We first review log-space NTMs. Here we especially consider *c-bounded k-working-tapes* log-space NTM $M = (k, c, Q, q_{\text{init}}, Q_F, \Sigma, \Gamma, \square, \Delta)$. Each component of M means:

- k is the number of working tapes $T_1, T_2, \ldots, T_k$.
- c is used to bound the size of working tapes. It will be defined precisely below.
- Q is a finite set of states, q_{init} is the initial state, and $Q_F \subseteq Q$ is a set of accepting states.
- Σ is an input alphabet.
- Γ is a working tape alphabet. $\square \in \Gamma$ is the blank symbol for working tapes.
- Δ is a set of transition rules. Each rule is either $p \xrightarrow{\tau \mid \theta} q$ or $p \xrightarrow[T_i]{\kappa \mapsto \kappa' \mid \theta} q$ where $p, q \in Q$, $\tau \in \Sigma \cup \{\vdash, \dashv\}$, $\kappa, \kappa' \in \Gamma \cup \{\vdash, \dashv\}$, and $\theta \in \{-1, +1, 0\}$.

Let $w \in (\vdash \Sigma^* \dashv)$ be a string surrounded by the left and right end markers. Valid configurations of M for $\vdash w \dashv$ are tuples $\langle q, i, (T_1, i_1), \ldots, (T_k, i_k)\rangle$ where

- $q \in Q$ is the current state. $i \in \mathbb{N}$ $(0 \le i < |w| + 2)$ is the current head position on $\vdash w \dashv$.
- $T_x \in (\vdash \Gamma^C \dashv)$ where $C = c \cdot \lceil \log |w| \rceil$ is the x-th working tape surrounded by the end markers. $\lceil \cdot \rceil$ is the ceiling function to integers; for example, $\lceil \log 3 \rceil = \lceil 1.584 \ldots \rceil = 2$.
  Remark: The tape capacity C is determined by the parameter c and the input w.
- i_x is the x-th tape head position on T_x $(0 \le i_x < C + 2)$.

We write $\mathbf{Valid}_M(w)$ (or, simply $\mathbf{Valid}(w)$) for the set of valid configurations for the input w. It is clear that $|\mathbf{Valid}(w)| = |Q| \times (|w|+2) \times (|\Gamma|^C \times (C+2))^k$ where $C = c \cdot \lceil \log |w| \rceil$.

For an input string w, we write $\mathcal{I}(w)$ to denote the initial configuration on $\vdash w \dashv$:

$$\mathcal{I}(w) = \langle q_{\text{init}},\ 0,\ (\vdash \square^C \dashv, 0), \ldots, (\vdash \square^C \dashv, 0)\rangle \text{ where } C = c \cdot \lceil \log |w| \rceil.$$

Let $\xi = \langle p, i, (T_1, i_1) \ldots, (T_x, i_x), \ldots, (T_k, i_k)\rangle$ be a valid configuration on $\vdash w \dashv$. For each transition rule δ, we define a labelled transition relation $\overset{\delta}{\Rightarrow}$ on *valid* configurations as follows:

$$\frac{\delta = p \xrightarrow{\tau \mid \theta} q \qquad (\vdash w \dashv)[i] = \tau}{\xi \overset{\delta}{\Rightarrow} \langle q,\ i + \theta,\ (T_1, i_1), \ldots, (T_k, i_k)\rangle} \qquad \frac{\delta = p \xrightarrow[T_x]{\kappa \mapsto \kappa' \ \mid\ \theta} q \qquad \kappa = T_x[i_x]}{\xi \overset{\delta}{\Rightarrow} \langle q, i, (T_1, i_1), \ldots, (T_x[i_x] := \kappa', i_x + \theta), \ldots, (T_k, i_k)\rangle}$$

where $T_x[i_x] := \kappa'$ is the new working tape obtained by writing κ' to the position i_x.

We also simply write $\xi \Rightarrow \xi'$ if there is a transition rule $\delta \in \Delta$ such that $\xi \overset{\delta}{\Rightarrow} \xi'$.

We write $\mathbf{NLOG}(c, k)$ for the set of c-bounded k-working-tapes log-space NTMs. If c and k is not important, by abusing notation, we simply write $\mathbf{NLOG}$. For $M \in \mathbf{NLOG}$ and an input string w, we write $M(w, \xi)$ to denote the set of valid and acceptable configurations that are reachable from a valid configuration ξ on $\vdash w \dashv$:

$$M(w, \xi) = \{\xi' \ :\ \xi \Rightarrow^* \xi',\ \xi' = \langle q_{\text{acc}}, i, \mathcal{T}\rangle,\ q_{\text{acc}} \in Q_F\},$$

where $\mathcal{T} = (T_1, i_1) \ldots (T_k, i_k)$ is a sequence of pairs of a working tape and an index. Now the language $L(M)$ is defined as:

$$L(M) = \{w : M(w, \mathcal{I}(w)) \neq \emptyset\}.$$

Here we state a useful proposition, which will be used below sometimes.

▶ **Proposition 10.** *Let $M \in \mathbf{NLOG}(c, k)$. For any input w, to represent a single valid configuration or store $|\mathbf{Valid}(w)|$, we need an extra $O(c \cdot k)$-bounded working tape.*

Proof. Since $|\mathbf{Valid}(w)| = |Q| \times (|w| + 2) \times (|\Gamma|^C \times (C + 2))^k$ where $C = c \cdot \lceil \log |w| \rceil$, $\log |\mathbf{Valid}(w)| = (k \cdot C) \log |\Gamma| + \cdots = O(k \cdot c) \log |w|$. So, we need an $O(c \cdot k)$-bounded tape. ◀

On log-space NTMs, we can solve the problem-***(B)*** in Section 1.1.

▶ **Proposition 11.** *Let $M \in \mathbf{NLOG}(c, k)$. There exists $N \in \mathbf{NLOG}(O(c \cdot k), k + 1)$ such that $L(M) = L(N)$ and, for any input w, all computations of N starting from w eventually halt.*

Proof. The number of reachable configurations is bounded by $\mathcal{B} = |\mathbf{Valid}_M(w)|$. So, we can safely ignore all paths P whose length $> \mathcal{B}$ without changing the accepting language. To check if the current path length $> \mathcal{B}$, we need an $O(c \cdot k)$-bounded tape by Proposition 10. ◀

We note that properties, like Proposition 11, are insufficient to show that a language class is closed under complement.[1] Thus, the problem-($\boldsymbol{A}$) in Section 1.1 is essentially hard; indeed, it is the interesting part of Immerman–Szelepcsényi theorem [16, 28]. In the following subsection, we revisit their theorem along with introducing log-space nested-oracles NTM.

5.1 Augmenting NTM with Nested Oracles

We extend log-space NTM with finitely nested oracles (or subroutines) to naturally handle nested lookaheads of REWBLk. Similar to our definition of log-space NTMs, we consider c-bounded k-tapes log-space nested oracles NTMs.

To allow nested oracle calling, we inductively define our machines. First, as the base case, we write $\mathbf{OLOG}^0(c,k) = \mathbf{NLOG}(c,k)$ to denote machines without oracles. Next, as the induction step, we define $\mathbf{OLOG}^{x+1}(c,k)$ using $\mathbf{OLOG}^x(c,k)$ as follows:

- Each $M \in \mathbf{OLOG}^{x+1}(c,k)$ is a tuple $(k, c, Q, q_{\text{init}}, Q_F, \Sigma, \Gamma, \square, \Delta)$.
- There are new transition rules, *oracle transition rules*, of the form $p \xrightarrow{\in N} q$ and $p \xrightarrow{\notin N} q$ where $N \in \mathbf{OLOG}^y(c,k)$ and $y \leq x$. Their semantics will be defined immediately later.

We write $\mathbf{OLOG}^\omega(c,k)$ for $\bigcup_{i=0}^{\infty} \mathbf{OLOG}^i(c,k)$. If c and k are not important, we simply write as $\mathbf{OLOG}^n$ and $\mathbf{OLOG}^\omega$. To denote the nesting level of machines $M \in \mathbf{OLOG}^\omega(c,k)$, we inductively define the function *depth* as follows:

$$\mathit{depth}(M) = \begin{cases} 0 & \text{if } M \in \mathbf{OLOG}^0(c,k), \\ 1 + \max\{\mathit{depth}(N) : p \xrightarrow{\in N} q,\ p' \xrightarrow{\notin N} q \in \Delta(M)\} & \text{otherwise.} \end{cases}$$

Now, we define the semantics of oracle transition rules as follows:

$$\frac{p \xrightarrow{\in N} q \qquad N(w, \langle q^N_{\text{init}}, i, \mathcal{T}\rangle) \ni \langle r,\ j,\ \mathcal{U}\rangle}{\langle p, i, \mathcal{T}\rangle \Rightarrow \langle q, i, \mathcal{U}\rangle} \qquad \frac{p \xrightarrow{\notin N} q \qquad N(w, \langle q^N_{\text{init}}, i, \mathcal{T}\rangle) = \emptyset}{\langle p, i, \mathcal{T}\rangle \Rightarrow \langle q, i, \mathcal{T}\rangle}$$

where q^N_{init} is the initial state of N, $\mathcal{T}$ and $\mathcal{U}$ is a sequence of pairs of a working tape and an index $(T_1, i_1) \dots (T_k, i_k)$, and the function

$$N(w, \xi) = \{\xi' : \xi \Rightarrow^* \xi', \xi' = \langle q, i, \mathcal{T}\rangle, q \in Q_F(N)\}$$

is defined inductively on the depth of machines.

The semantics of $p \xrightarrow{\in N} q$ means that: (1) we call an oracle (subroutine) N for $\vdash w \dashv$ with the current position i and the current working tapes $\mathcal{T}$ as its initial working tapes; and (2) if N accepts w, then we enter a state q with the original position i and working tapes $\mathcal{U}$ of N's accepting configuration. The semantics of $p \xrightarrow{\notin N} q$ means that, if N does not accept w, we enter a state r with the original position and working tapes $\mathcal{T}$.[2]

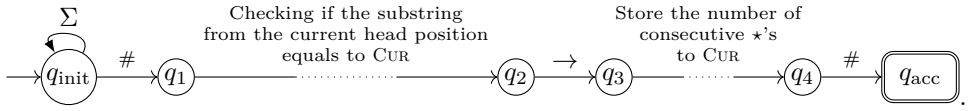
▶ **Example 12.** Using log-space nested-oracles NTMs, we simulate the following REWBLk:

$$E_{\text{reach}} = (V^*)_{\text{Cur}}\,\#\,\left(?(\Sigma^*\,\#\setminus\text{Cur} \to (V^*)_{\text{Cur}}\,\#)\right)^*\,\Sigma^*\,\#\setminus\text{Cur}.$$

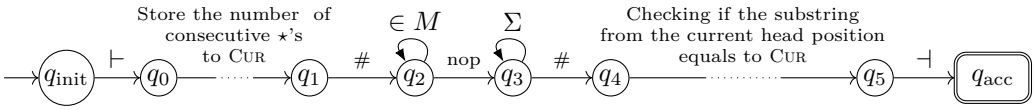
[1] For example, we can translate any nondeterministic pushdown automata to real-time ones, which do not have ϵ-transitions. However, the class of context free languages is not closed under complement.

[2] For simplicity, our oracle formalization differs from traditional treatments [17, 22, 27, 3] in some points: (1) we omit the use of oracle tapes, and (2) we allow inheriting configurations from called oracles. Despite these differences, our definition is adequate for Theorem 13 and for REWBLk in Section 6.

For the sake of simplicity, we assume that $V = \{\star\}$ is a unary alphabet. The subexpression $(\Sigma^* \# \backslash \textsc{Cur} \to (V^*)_{\textsc{Cur}} \#)$ is simulated by the following log-space NTM $M \in \mathbf{NLOG}$:



Using M, we give the following machine $N \in \mathbf{OLOG}^1$, clearly accepting $L(E_{\text{reach}})$:

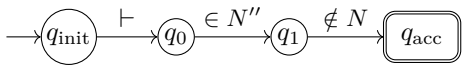


where edges labelled with "nop", $\xrightarrow{\text{nop}}$, mean transitions that only change states and do not depend scanning symbol. It just a syntax sugar because we can define $p \xrightarrow{\text{nop}} q$ by a set of transition rules $\{p \xrightarrow{\tau \mid 0} q : \tau \in \Sigma \cup \{\vdash, \dashv\}\}$.

▶ **Example 13.** We can also accept the language of non-reachability problems:

$$L_{\text{non-reach}} = \{s \# x_1 \to y_1 \# \cdots \# x_n \to y_n \# t : \text{there is no path from } s \text{ to } t\}.$$

To recognize this language, we use the following $M_{\text{non-reach}} \in \mathbf{OLOG}^2$:



where $N'' \in \mathbf{NLOG}$ recognizes the language defined by $V^* \# (V^* \to V^* \#)^* V^*$.

5.2 Collapsing $\mathbf{OLOG}^\omega$ by Immerman–Szelepcsényi theorem

Thanks to Proposition 11, we can give a decision procedure to check $w \in_? L(M_{\text{non-reach}})$ for our above examples. However, it is not clear that $\mathbf{OLOG}^2 =_? \mathbf{NLOG}$ and more generally $\mathbf{OLOG}^\omega =_? \mathbf{NLOG}$. For example, is there a log-space NTM N that recognizes $L_{\text{non-reach}}$?

Fortunately, the class **NLOG** is closed under complement, $\mathbf{NLOG} = \mathbf{co\text{-}NLOG}$. This result is known as Immerman–Szelepcsényi theorem [16, 28]. We employ their proof to collapse $\mathbf{OLOG}^x$ for some x to log-space NTMs $\mathbf{OLOG}^0 = \mathbf{NLOG}$.

▶ **Lemma 14.** *Let $M \in \mathbf{NLOG}(c, k)$. There is a machine $\overline{M} \in \mathbf{NLOG}(O(c \cdot k), k + \partial)$ where $L(\overline{M}) = \Sigma^* \backslash L(M)$ and ∂ is independent of M, c, and k.*

Proof. We review Immerman's original construction in [16]. For the reader who would like to know more detailed explanation about his construction, we recommend some literature [27]. His construction consists of the following two parts:

- Let Start be a configuration of M. First, we compute the number C, the total number of configurations reachable from Start.
- Next, using C, we check if there is a path from Start to an acceptable configuration.

The first part is accomplished by the following pseudocode [16, Lemma 2].

```
global w; // input string

// For configurations x and x′, we check if x ⇒ x′
def one_step_M(x, x′):
```

```
    foreach δ ∈ Δ(M): // Δ(M) is the set of transition rules of M
        if x ⇒^δ x': return True;
    return False;

// calculate the total number of configurations reachable from START
def counting_M(START):
    cur ← 1; // the number of reachable configurations within ≤ dist steps
    for(dist ← 0, next ← 0; dist < |Valid_M(w)|; dist += 1, cur ← next, next ← 0):
        foreach x ∈ Valid_M(w):
            count ← 0; found_x ← false;
            foreach y ∈ Valid_M(w):
                z ← START; // search a path from START to y
                for(i ← 0; z ≠ y & i < dist; i += 1):
                    z' ← Nondeterministically generated configuration;
                    if one-step(z, z'): z ← z';
                    else: break;
                if z = y:
                    count += 1;
                    if one-step(y, x): { next += 1; found_x ← true; break; }
            if ¬found_x & count ≠ cur: Halt and reject;
    return cur;
```

These functions require extra working tapes at least for variables of δ, *cur*, *dist*, *next*, x, *count*, y, i, z, and z'. By Proposition 10, for each variable, we need an $O(c \cdot k)$-bound working tape.

The second part is accomplished by the following pseudocode [16, Lemma 1].

```
// Judge whether an acceptable configuration can be reached from START.
// If so, it returns such a configuration y by SOME(y).
// Otherwise, we return the nothing by NONE.
def judge_M(START):
    C ← counting_M(START);
    count ← 0;
    foreach x ∈ Valid_M(w):
        y ← START;
        for(i ← 0; i ≤ C; i += 1):
            y' ← Nondeterministically generated configuration;
            if one-step(y, y'):
                y ← y';
                if y is an accepting configuration: return SOME(y);
                if y = x: { count += 1; break; }
            else: break;
    if count = C: return NONE;
    else: Halt and reject;
```

This function also requires extra $O(c \cdot k)$-bound working tapes.

Now we can build $\overline{M} \in \mathbf{NLOG}(O(c \cdot k), k + \partial)$ as a log-space NTM that simulates the function `judge` and then accepts inputs if $\texttt{judge}_M(\textsc{Start})$ is NONE. ◀

Repeatedly applying Immerman's construction collapses nested oracle machines to ma-

chines without oracles [16, Corollary 2].

▶ **Theorem 15.** *Let $M \in \boldsymbol{OLOG}^n(c,k)$ be a log-space n-nested-oracles NTM. There exists a log-space NTM $N \in \mathbf{NLOG}(O(c \cdot k^n), O(k \cdot n))$ such that $L(M) = L(N)$.*

Proof. We eliminate oracle transitions from the innermost to the outermost for M as follows. We replace $p \xrightarrow{\in N} q$ with $N \in \mathbf{OLOG}^0$ with multiple transition rules that perform: (1) save the head position H to an extra tape; (2) run N; (3) if we reach an accepting configuration $\langle q_f, \mathcal{T}\rangle$, then we continue $\langle q, H, \mathcal{T}\rangle$. Similarly, we replace $p \xrightarrow{\notin N} q$ with $N \in \mathbf{OLOG}^0$ with multiple transition rules using $\overline{N}$ obtained by Immerman's construction. We emphasize that each generation of $\overline{N}$ increases c and k in the order of the statement of Lemma 14. ◀

5.3 Membership Problem of Log-space Nested-oracles NTMs

We now show that the membership problem of log-space nested-oracles machines is in **PSPACE**. We first formally state our membership problem.

▶ **Definition 16** (Membership problem of $\mathbf{OLOG}^\omega$)**.** *The membership problem of $\boldsymbol{OLOG}^\omega$ is a decision problem of the following form:*
Inputs *Binary encoded integers c and k. A machine $M \in \boldsymbol{OLOG}^\omega(c,k)$. A word $w \in \Sigma^*$.*
Output *If $w \in_? L(M)$, return Yes. Otherwise, No.*

To show that the problem belongs to **PSPACE**, we would like to employ Theorem 15. However, it is not feasible because the theorem gives a log-space $O(c \cdot k^{|M|})$-bounded tapes machine N in general; i.e., N demands $O(c \cdot k^{|M|}) \cdot \log |w|$ space. Thus, we cannot simulate N in polynomial size in c, k, $|M|$, and $|w|$. To address this problem, we adopt Immerman's construction for $\mathbf{OLOG}^\omega$ in an interpreter style.

▶ **Theorem 17** (Membership problem of $\mathbf{OLOG}^\omega$ belongs to **PSPACE**)**.** *Let w be an input word and $M \in \boldsymbol{OLOG}^\omega(c,k)$ be an input machine where c and k are binary encoded. We can decide if $w \in_? L(M)$ in polynomial space in c, k, $|w|$, and $|M|$.*

Proof. First, we extend the function `one-step` for oracle transitions as follows:

```
def one-step_{M_i}(x, x'): // return True if x ⇒ x'. Otherwise, False.
    foreach δ ∈ Δ(M_i):
        if δ is a non-oracle transition & x ⇒^δ x': return True;
        else: // δ is an oracle transition
            (p, i, T) ← x; // extract state, position, and tape contents from x
            if δ = p -∈N-> q:
                match judge_N(⟨q^N_init, i, T⟩): // pattern matching
                case Some(⟨p', i', U⟩) -> return (x' =? ⟨q, i, U⟩);
                case None -> return False;
            if δ = p -∉N-> q:
                match judge_N(⟨q^N_init, i, T⟩):
                case Some(⟨p', i', U⟩) -> return False;
                case None -> return (x' =? ⟨q, i, T⟩);
    return False;
```

Next, we generate the codes of $\texttt{one-step}_{M_i}$, $\texttt{counting}_{M_i}$, and $\texttt{judge}_{M_i}$ for all oracle machines M_i that appears in M. Such generation is carried out in polynomial-time for $|M|$. The total size of generated code is also polynomial in $|M|$.

We can also provide an interpreter for the generated code in polynomial time for $|M|$. While this interpreter needs a call stack for function calls, its depth is bounded by $\mathit{depth}(M) \leq |M|$. Additionally, the size of each stack frame is bounded by $O((c{\cdot}k)\log|w|)$ by Proposition 10. From the above argument, we can check $w \in_? L(M)$ using (nondeterministic) polynomial space with respect to c, k, $|w|$, and $|M|$. ◀

We will use this theorem to show that the membership problem of REWBLk belongs to **PSPACE** in the following section. To this end, we put remarks about this theorem.

Remark (Drop k from Theorem 17): We can decide $w \in_? L(M)$ in polynomial space for c, $|w|$, and $|M|$. Compared with Theorem 17, this refined version does not depend on k. This is because $k \leq |M|$ holds, even when k is binary encoded.

Of course, to establish this property, we assume a natural restriction that all tapes $T_1, \ldots, T_k$ are used in some transition rules. Indeed, if $M \in \mathbf{OLOG}^\omega(c,k)$ has a working tape T_i with $1 \leq i \leq k$ that is not engaged by M, a better parameter k' (where $k' < k$) should be employed, ensuring $M \in \mathbf{OLOG}^\omega(c,k')$. Given such constraints, $|M| \geq k \cdot \log(k) \geq k$ since k tapes necessitate $\log(k)$-space for tape identification.

Remark (Cannot drop c from Theorem 17): On the other hand, we cannot drop c from Theorem 17. Namely, we cannot decide $w \in_? L(M)$ in polynomial space for $|w|$ and $|M|$ because c is not bounded by $\mathrm{Poly}(|M|)$ in general. (Please recall that $k \leq |M|$.) It can be understood from the following argument, which establishes $c = \Omega(2^{|M|})$. Let us consider a machine M that runs as follows:

- First, M fills a tape T_1 with N 1's by using states $q_1, q_2, \ldots, q_N$.
- Next, M interprets T_1 as a binary number t_1 and fills a tape T_2 with t_1 1's.
- Finally, M writes the contents of T_2 N times to T_3.
- For example, if $N = 3$, then M makes $T_1 = 111$ and then makes $T_2 = 1111111 = 1^{2^3-1}$ and then $T_3 = 3\,T_2 = 1^{3\cdot(2^3-1)} = 1^{N\cdot(2^N-1)}$.

The size of M satisfies $|M| = O((\log N)N)$ since it contains binary-encoded N states; thus, the length of T_3, $|T_3|$, satisfies $|T_3| = \Omega(2^{|M|})$. By appending the content of T_3 onto another tape $\log|w|$ times (where w is the input word), we establish that $c = \Omega(2^{|M|})$.

However, fortunately we can assume $c = 1$ when simulating REWBLk by $\mathbf{OLOG}^\omega$ as we will see below. It is crucial for showing that the membership problem of REWBLk belongs to **PSPACE**.

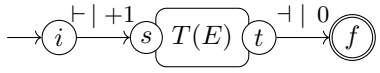
6 From REWBLk to Log-Space Nested Oracles NTM

We finally show **REWBLk** $\subseteq$ **NLOG** by translating REWBLks to $\mathbf{OLOG}^\omega$.

▶ **Theorem 18.** *Given a REWBLk expression E, we can translate it to $M \in \boldsymbol{OLOG}^\omega(1, O(|E|))$ in polynomial time in the size of $|E|$ where $L(E) = L(M)$.*

Proof. We inductively translate a given REWBLk E to a $\mathbf{OLOG}^\omega$ $T(E)$.

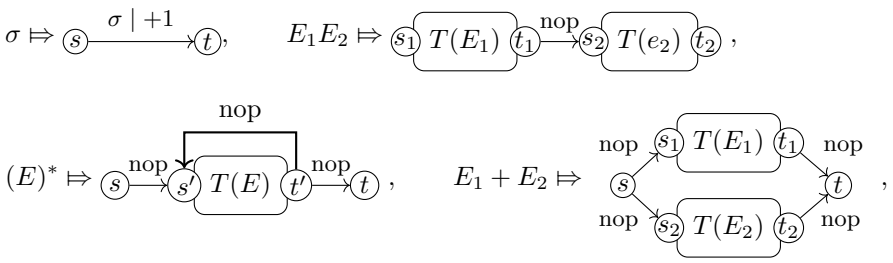
As we will see below, the generated $T(E)$ has a unique source state s, which does not have incoming edges, and a unique sink state t, which does not have outgoing edges. Please recall that E accepts an input w if it consumes all the input, i.e., if we have $\langle p, \Lambda\rangle \in [\![\langle E, w, 0, \iota\rangle]\!]$ with $p = |w|$. For M, we add transition rules to treat the endmarkers $\vdash$ and $\dashv$:

$M =$ 

where i and f are initial and accepting states of M.

Translating REGEX Part

First, we give a translation for the REGEX as follows:

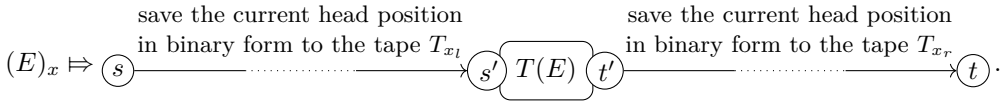


where the edges labelled with "nop" are the same ones used in Example 12, which just change states. This translation is identical to the McNaughton–Yamada–Thompson algorithm, which is well-known and found in textbooks of automata theory.

Translating Backreference and Capturing Expressions

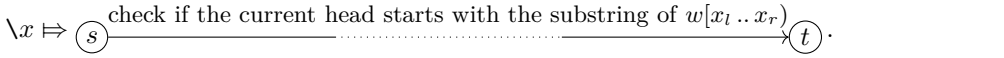
Using Theorem 2, we can assume that every variable x does not appear inside an expression capturing x; i.e., we avoid patterns such that $(\cdots x \cdots)_x$. The following translation heavily depends on the theorem.

We now focus on the part $(E)_x$ of REWB:



In order to keep the start position of the variable x, we first copy the current head position to the special working tape T_{x_l} in binary form. Then, we execute the expression E by running from the state s to t. Finally, we record the new head position into the working tape T_{x_r}. Now, $w[x_l \mathinner{..} x_r) = w[x_l]w[x_l + 1] \cdots w[x_r - 1]$ is a substring matched with the expression E where x_l and x_r are the numbers corresponding to the contents of T_{x_l} and T_{x_r}.

Next, we focus on the part $\backslash x$ of REWB:



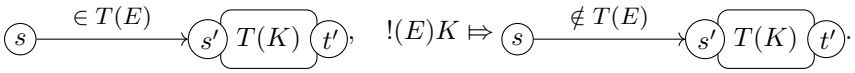
As we have seen above, the substring $w[x_l \mathinner{..} x_r)$ denotes the value of the variable x. This checking task is accomplished using an extra tape T_{tmp} without changing T_{x_l} and T_{x_r}.

Remark: Why do we need Theorem 2. Let us consider an expression $E = (a)_x\ (\backslash x \backslash x)_x$ and run it for an input string a. We underline $a \notin L(E)$. By interpreting $(a)_x$, we set the position 0 to T_{x_l} and 1 to T_{x_r}. On the translation of $(\backslash x \backslash x)_x$, we first save the current head position 1 to the tape T_{x_l} and then proceed to the part $\backslash x \backslash x$. It should be noted that, at this point, T_{x_l} denotes 1; so, we cannot correctly recover the captured content by $(a)_x$. When meeting $\backslash x$, we check if the substring between $T_{x_l}(= 1)$ and $T_{x_r}(= 1)$ starts from the current head position. Since the substring is the empty string ϵ, we go through the part $\backslash x \backslash x$ and accepts a incorrectly.

To prevent us from this situation, we use Theorem 2. It rewrites E to $(a)_x\ ?((\backslash x \backslash x)_y)(\backslash y)_x$, and the rewritten expression is safely interpreted thanks to the variable y.

Translating Positive and Negative Lookaheads

For our construction, we need that lookahead are augmented with a continuation K, for instance $?(E)K$ and $!(E)K$. If a given expression does not satisfy this property, we add the expression ϵ, which always succeeds. For example, $(E_1E_2!(E_3))^*E_4 \Rightarrow (E_1E_2(!(E_3)\epsilon))^*E_4$. Now, we can easily translate $?(E)K$ and $!(E)K$ using oracle transition rules as follows:



By the semantics of our nested oracle machines, it is clear that $L(E) = L(M)$. ◀

As corollaries of Theorem 18, we have the following main results.

▶ **Corollary 19.** **REWBLk** = **NLOG**.

Proof. We already have **NLOG** ⊆ **REWBLk** by Theorem 8. It is an immediate result from Theorem 15 and 18 that **REWBLk** ⊆ **NLOG**. ◀

▶ **Corollary 20.** *The membership problem of* REWBLk *is* ***PSPACE****-complete.*

Proof. By Theorem 9, the membership problem of REWBLk is **PSPACE**-hard.

The result that the problem belongs to **PSPACE** is shown as follows. Let E be an input REWBLk expression and w be an input string. By Theorem 17, we translate $M \in \mathbf{OLOG}^{\omega}(1, O(|E|))$ such that $L(E) = L(M)$. This translation is carried out in polynomial time in $|E|$; so, $|M| = \mathrm{Poly}(|E|)$. By Theorem 17 and its remarks, we can check if $w \in_? L(M)$ in polynomial space in $|w|$ and $|M|$. This also derives that we can check if $w \in_? L(E)$ in polynomial space in $|w|$ and $|E|$. ◀

7 Future Work and Conclusion

We have shown that the language class REWBLk—regular expressions plus backreferences and lookaheads (without any restrictions)—captures the class **NLOG**. Our result closes the expressiveness about REWBLk. Furthermore, we have shown that the membership problem of REWBLk is **PSPACE**-complete. On the other hand, it remains unclear whether REWB(+) and REWB(−) are proper subclasses of REWBLk. For example, we conjecture that REWB(+) cannot recognize the language $L_{\text{prime}} = \{a^n : n \text{ is a prime number}\}$.[3] It is known that the language can be represented by REWB(−). Also, we conjecture that some **NLOG**-languages cannot be recognized by REWB(−).

References

1 A. V. Aho. Indexed grammars – an extension of context free grammars. *J. ACM*, 15(4):647–671, 1968.

2 A. V. Aho. Algorithms for finding patterns in strings. In J. V. Leeuwen, editor, *Algorithms and Complexity*, Handbook of Theoretical Computer Science, pages 255–300. Elsevier, 1990.

3 S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[3] REWB(−) recognizes L_{prime} as follows: (1) we define $E_{\text{composite}} = (aa\,a^*)_w\backslash w\backslash w^*\$$, which recognizes composite numbers; (2) then, we define $E_{\text{non-prime}} = a\$ + E_{\text{composite}}$, which recognizes non-prime numbers; (3) finally, we define $E_{\text{prime}} = !(E_{\text{non-prime}})\ a^*\ \$$. Readers interested in conducting a prime test with E_{prime} can try the following expression in C#: `@"^(?!((a$)|((?<W>(aa+))(\k<W>+)$))).*$"`.

4 M. Berglund and B. van der Merwe. Re-examining regular expressions with backreferences. *Theoretical Computer Science*, 940:66–80, 2023.
5 C. Câmpaanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14(06):1007–1018, 2003.
6 A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.
7 N. Chida and T. Terauchi. On lookaheads in regular expressions with backreferences. In *FSCD 2022*, volume 228, pages 15:1–15:18. Schloss Dagstuhl, 2022.
8 N. Chida and T. Terauchi. On lookaheads in regular expressions with backreferences. *IEICE Transactions on Information and Systems*, E106.D(5):959–975, 2023.
9 ECMAScript community. Ecmascript 2023 language specification. `https://262.ecma-international.org/14.0/#sec-runtime-semantics-repeatmatcher-abstract-operation`.
10 D. D. Freydenberger and M. L. Schmid. Deterministic regular expressions with back-references. *Journal of Computer and System Sciences*, 105:1–39, 2019.
11 R. H. Gilman. A shrinking lemma for indexed languages. *Theoretical Computer Science*, 163(1):277–281, 1996.
12 J. Hartmanis. On non-determinancy in simple computing devices. *Acta Informatica*, 1(4):336–344, 1972.
13 J. Hartmanis and S. Mahaney. Languages simultaneously complete for One-way and Two-way log-tape automata. *SIAM Journal on Computing*, 10(2):383–390, 1981.
14 T. Hayashi. On derivation trees of indexed grammars—an extension of the uvwxy-theorem—. *Publications of the Research Institute for Mathematical Sciences*, 9(1):61–92, 1973.
15 M. Holzer, M. Kutrib, and A. Malcher. Complexity of multi-head finite automata: Origins and directions. *Theoretical Computer Science*, 412(1):83–96, 2011.
16 N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.
17 N. Immerman. *Descriptive Complexity*. Springer Verlag, 1998.
18 T. Jiang and B. Ravikumar. A note on the space complexity of some decision problems for finite automata. *Information Processing Letters*, 40(1):25–31, 1991.
19 K.-J. Lange, B. Jenner, and B. Kirsig. The logarithmic alternation hierarchy collapses: $A\Sigma_2^{\mathcal{L}} = A\Pi_2^{\mathcal{L}}$. In *ICALP 87*, pages 531–541. Springer, 1987.
20 Y. V. Matiyasevich. *Hilbert's Tenth Problem*. MIT Press, 1993.
21 T. Nogami and T. Terauchi. On the expressive power of regular expressions with backreferences. In *MFCS 2023*, volume 272, pages 71:1–71:15. Schloss Dagstuhl, 2023.
22 C. H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.
23 W. C. Rounds. Complexity of recognition in intermediate level languages. In *SWAT'73*, pages 145–158, 1973.
24 M. L. Schmid. Inside the class of regex languages. *International Journal of Foundations of Computer Science*, 24(07):1117–1134, 2013.
25 M. L. Schmid. Characterising regex languages by regular languages equipped with factor-referencing. *Information and Computation*, 249:1–17, 2016.
26 U. Schöning and K. W. Wagner. Collapsing oracle hierarchies, census functions and logarithmically many queries. In *STACS 88*, pages 91–97. Springer, 1988.
27 M. Sipser. *Introduction to the theory of computation*. Cengage Learning, third edition, 2013.
28 R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

A Proof of Theorem 4: Non Indexed Languages Accepted by REWB(+) and REWB(−)

We said that the language $L(E)$ grows faster than the language $L_{2exp} = \{a^{2^{2^n}} : n \in \mathbb{N}\}$ in Section 4.1. To complete the proof sketch for the REWB(−) part, we introduce the notion

of faster growing. To formally state the notion, we first introduce the (growth) order function $\mathcal{G}_L : \mathbb{N} \to \mathbb{N}$ for unary languages L as follows:

$\mathcal{G}_L(n) =$ the length of n-th shortest word in L.

For example, $\mathcal{G}_{L_{1exp}}(n) = 2^n$ where $L_{1exp} = \{a^{2^n} : n \in \mathbb{N}\}$, $\mathcal{G}_{L_{2exp}}(n) = 2^{2^n}$, and $\mathcal{G}_{L_{2\text{tower}}}(n) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{n}$. Using $\mathcal{G}$, for two unary languages L_1 and L_2, we say L_1 *grows faster* than L_2 if $\mathcal{G}_{L_1}(n) = \Omega(\mathcal{G}_{L_2}(n))$.

We recall an important result about indexed languages: i.e., for any indexed language I, its growth order is $O(2^n)$. In other words, $\mathcal{G}_I(n) = O(2^n)$. This result was shown by pumping or shrinking arguments on indexed languages [14, 11]. Since the growth rate of $L(E'_{2exp})$ surpasses the order 2^n clearly, REWB$(-)$ can represent a non-indexed language.

▶ **Lemma A.1.** *REWB$(-)$ recognizes a language that grows faster than L_{2tower}.*

Next, let us consider the REWB$(+)$ part. Combining the construction the example of Section 4.1 and a technique, which we call *halving*, we can give a REWB$(+)$ expression representing L_{2exp}.

▶ **Lemma A.2.** *REWB$(+)$ can recognize the doubly exponential language $L_{2exp} = \{a^{2^{2^n}} : n \in \mathbb{N}\}$.*

Proof. We proceed as the example of Section 4.1; therefore,

- By $E_1 = ?(a)_m?((\backslash na)_n(\backslash m\backslash m)_m)^*$, we make n and $m = 2^n$ nondeterministically.
- Also, by $E_2 = ?(a)_y?((\backslash xa)_x(\backslash y\backslash y)_y)^*$, we make x and $y = 2^x$ nondeterministically.

Then, we need to check $x = m$; however, due to the absence of negative lookaheads, especially \$, we cannot do $?(\Sigma^*?(\backslash m\$)?(\backslash x\$))$ to check it.

Instead of using \$, we use another technique, which heavily depends on the acceptance condition of REWBLK—we need to consume all the inputs.

As idea, we build $m_{1/2}$ (resp. $x_{1/2}$) that contains the half a's of m (resp. x).

Then, $E = E_1E_2\backslash y?(\backslash m)?(\backslash x)\backslash m_{1/2}\backslash x_{1/2}$ accepts w iff $m = x$; therefore, $L(E) = \{a^{2^{2^n}}a^{2^n} : n \in \mathbb{N}\}$.

To show that E accepts w iff $m = x$, we show (1) $m = x \implies w \in L(E)$; and (2) $m \neq x \implies w \notin L(E)$.

In the case $m = x$, it suffices to showing $\backslash m = (\backslash m_{1/2}\backslash x_{1/2})$. It is clear.

In the case $m < x$, $(\backslash m_{1/2}\backslash x_{1/2}) < \backslash x$.

Therefore, if we reach the point $\backslash m_{1/2}\backslash x_{1/2}$, we cannot consume the rest part since $\backslash y\backslash x$ is a prefix of w ($\backslash y\backslash m_{1/2}\backslash x_{1/2} \preceq \backslash y\backslash x \preceq w$).

In the case $m > x$, we cannot consume all the input w; hence, $w \notin L(E)$. ◀

Combining these lemmas, we obtain the following theorem.

▶ **Theorem 3.** *REWB$(+)$ and REWB$(-)$ can represent non indexed languages.*

B Proof of Theorem 5: Undecidability of Emptiness Problems of Unary REWB$(+)$ and Unary REWB$(-)$

Here we consider unary REWB$(+)$ and unary REWB$(-)$ whose input alphabet is a single set $\Sigma = \{a\}$.

▶ **Lemma B.1.** *The emptiness problem of unary REWB(−) is undecidable.*

Proof. We use the undecidability of checking if a given Diophantine equation has a solution of *natural* numbers [20].

To illustrate our idea, let us consider a Diophantine equation and transform it to one where each coefficient is positive as follows:

$$D : 2x^3 - (x-1)y = -2 \quad \Rightarrow \quad 2x^3 + y + 2 = xy.$$

This has a solution e.g., $(x, y) = (2, 18)$.

We nondeterministically build x, y, x^2, x^3, and xy in this order in inputs using the expression

$$E_{\mathrm{GEN}} = (a^*)_x(a^*)_y((\backslash x^2 \backslash x)_{x^2}(\backslash w_x a)_{w_x})^*((\backslash x^3 \backslash x^2)_{x^3}(\backslash w'_x a)_{w'_x})^*((\backslash xy \backslash y)_{xy}(\backslash w''_x a)_{w''_x})^*$$

as follows:

$$\underbrace{\cdots}_{x}\underbrace{\cdots}_{y} \mid \backslash x\, a \quad 2\backslash x\; a^2 \cdots \underbrace{n\backslash x}_{x^2}\underbrace{a^n}_{w_x} \mid \backslash x^2\, a \cdots \underbrace{m\backslash x^2}_{x^3}\underbrace{a^m}_{w'_x} \mid \qquad \backslash y\, a \cdots \underbrace{l\backslash y}_{xy}\underbrace{a^l}_{w''_x}$$

If we can ensure $\backslash x = \backslash w_x = \backslash w'_x = \backslash w''_x$, then $\backslash x^2 = a^{x^2}$, $\backslash x^3 = a^{x^3}$, and $\backslash xy = a^{xy}$ are derived. Then, it suffices to performing $!!(\backslash x^3 \backslash x^3 \backslash yaa\$)!!(\backslash xy\$)$ to check $2x^3 + y + 2 = xy$. Here $!!(E)$ is a shortened version of $!(!(E))$, which means quasi positive lookaheads where we cannot update variables in such lookaheads.

To ensure $\backslash x = \backslash w_x = \backslash w'_x = \backslash w''_x$, we extend the above expression as follows:

$$E_{\mathrm{check}} = !!(\backslash x^3 \backslash x^3 \backslash yaa\$)!!(\backslash w_{xy}\$)\ a^*\ !!(\backslash v_x\$)!!(\backslash w_x\$)!!(\backslash w'_x\$)!!(\backslash w''_x\$)\ a^*.$$

The entire expression is $E = E_{\mathrm{GEN}} E_{\mathrm{check}}$, and then $L(E) \neq \emptyset$ iff D has a solution. ◀

Next, we focus on unary REWB(+).

▶ **Lemma B.2.** *The emptiness problem of unary REWB(+) is undecidable.*

Proof. Let us consider the following (binary) PCP instance:

	R_1	R_2	R_3
α	a	ab	bba
β	baa	aa	bb

$\Longrightarrow$

	R_1	R_2	R_3
α	2	24	442
β	422	22	44

We replace the characters by 2 and 4, here $a \mapsto 2$ and $b \mapsto 4$. It has a solution $R_3R_2R_3R_1$; $442\,24\,442\,2 = 44\,22\,44\,422$.

Starting from the number 2, we can simulate applying each rule. For example, apply $R_3 = 442$ of α, we first multiply the current value by $10^{|442|=3}$ and then add 442. Repeatedly applying this, we obtain

$$\alpha : 2 \xrightarrow{R_3} 2 \cdot 10^3 + 442 \xrightarrow{R_2} 2442 * 10^2 + 24 \xrightarrow{R_3} 244224442 \xrightarrow{R_1} 2442244422,$$

where we drop the leftmost 2, then we obtain the correct value 442244422. To build an expression E that satisfies $L(E) \neq \emptyset$ iff there is a solution of the PCP instance, we take the following idea:

1. Besides calculating α and β, we also compute their halves $\alpha^1\!/_2$ and $\beta^1\!/_2$.
2. If $\alpha = \beta$, $\alpha^1\!/_2 + \beta^1\!/_2 = \alpha = \beta$. Otherwise, $\alpha^1\!/_2 + \beta^1\!/_2 < \max \alpha, \beta$.
3. Therefore, it holds that $?(\alpha)?(\beta)(\alpha^1\!/_2\beta^1\!/_2)$ matches a^n iff the PCP instance has a solution, whose result is a^n.

4. Please recall that REWBLk determines acceptance based on if it can consume the entire input.

To initialize variables, we consider $E_{\text{init}} = ?(aa)_{\alpha}?(a)_{\alpha_{1/2}}?(aa)_{\beta}?(a)_{\beta_{1/2}}$. We use $E_{R_1} = \begin{pmatrix} ?((10 \times \backslash\alpha)a^2)_{\alpha}?((10 \times \backslash\alpha_{1/2})a^1)_{\alpha_{1/2}} \\ ?((10^3 \times \backslash\beta)a^{422})_{\beta}?((10^3 \times \backslash\beta_{1/2})a^{211})_{\beta_{1/2}} \end{pmatrix}$ to reflect the rule R_1. Also defining E_{R_2} and E_{R_3}, then we just consider $E = E_{\text{init}}(E_{R_1} + E_{R_2} + E_{R_3})^* \; ?(\backslash\alpha) \; ?(\backslash\beta)\backslash\alpha_{1/2} \; \backslash\beta_{1/2}$. Now, $L(E) \neq \emptyset$ iff the PCP has a solution. ◀

Combining these lemmas, we obtain the following theorem.

▶ **Theorem 4.** *The emptiness problems of REWB(+) over a unary alphabet and REWB(−) over a unary alphabet are undecidable.*