Finitely Presented Group Construction from Turing Machine

Maksym Shamrai
Saint Petersburg State University
St. Petersburg, Russia
vortmanmax@gmail.com

Semyon Grigorev
Saint Petersburg State University
JetBrains Research
St. Petersburg, Russia
s.v.grigoriev@spbu.ru
semyon.grigorev@jetbrains.com

***Abstract*—A study of formal languages showed there are open problems that rather difficult to solve involving traditional combinatorial methods. In this paper, we try to bring group theory methods into play by developing an algorithm, which had been described by Sapir, Birget, and Rips, and constructs a presentation of a finitely presented group by a given Turing machine. Also, we obtained an injective mapping from an alphabet of the Turing machine language into a word problem of the corresponding group. The algorithm can be used to further study the classification of formal languages in terms of well-known classes of groups, highlight signs of belonging to a particular class, and apply the algebraic methods to formal languages. Generally, we try to bring new approach to analyze formal languages by group theory.**

***Index Terms*—Turing machine, symmetric Turing machine, S-machine, group presentation, group theory, word problem, formal languages, Haskell**

I. Introduction

Formal languages are the theoretical foundation of system programming, namely the development of translators and static analyzers, through lexical, syntactic, and semantic analysis. The formal language analysis approach has begun new development in the 50s of the last century [1, 2]. During this time, it found application not only in areas directly related to programming but, among other things, in the analysis of graph data models [3, 4], which are used, for example, in bioinformatics [5] and social networks [6].

Development entails new theoretical problems and challenges. Several of them are associated with the advent of conjunctive [7] and Boolean language [8] classes. The first and most important issue concerns the bounds of Boolean grammars. The limitations of the expressive power of conventional context-free grammars are fairly well known. There are direct methods for proving the unrepresentability of certain languages, such as the pumping lemmas and its variants [9], which show that some languages cannot be generated by a context-free and regular grammars. On the contrary, there are no methods for proving the unrepresentability of languages by Boolean grammars, and this is the main gap in the knowledge of these grammars. Similarly, such methods are not known for conjunctive grammars [10].

Recently, there has been a tendency to shift the direction of methods of studying formal languages. Researchers are moving away from combinatorial techniques and try to find new methods for analysis formal languages. One of the possible directions is the study of formal languages with the help of a group-theoretical methods.

Well known that formal languages too close to group theory. It is seen if we pay attention to a group constructed using an alphabet of formal language like generators set and concatenation operation. A formal language can be naturally defined from a description of a group presentation, however, in the opposite direction, the task is not trivial. Unfortunately, there is no simple way to construct a group presentation using a formal language. For this, complex mathematical transformations are carried out for a Turing machine that recognizes a language, and it is not advisable to produce them with a pen and paper, but studying the group properties of languages seems to be a rather promising area of research [11, 12].

Thus, the problem of the development of the algorithm for constructing a group using a Turing machine is relevant. Since constructing a Turing machine that accepts certain language is no big deal, we focused on the construction of group presentation by Turing machine, but as an example, in experiments, we construct Turing machines that accept context-free languages.

In total, the main contribution of this work is the development of an algorithm proposed by Sapir, Birget, and Rips [11], which constructs a finite presentation of a group by any Turing machine. We also implemented an algorithm for constructing a Turing machine based on a context-free grammar, which can be used with the developed algorithm of constructing a group presentation for the study of formal languages.

II. Background

In this section, the definitions and theorems used in this paper are given, the connection between formal languages and groups is considered, and the formal algorithm for constructing a presentation of a group by a Turing machine proposed in the article [11] is considered.

A. Formal languages

In this work, we use a standard definition for the formal languages such can be seen for example in Chomsky's works (e.g. see [1, 2]). In follows, we present the base ones we use in this work.

Definition II.1. In the classic formalization a formal grammar G can be presented as four-tuple $G = (\Sigma, N, R, S)$ and consists of the following components:

- A finite set Σ of terminal symbols.
- A finite set N of nonterminal symbols that is disjoint from Σ.
- A finite set R of production rules, each rule of the form $(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$
- A letter $S \in N$ is the start nonterminal.

Definition II.2. A context-free grammar is a special case of a formal grammar (type 2 according to the Chomsky hierarchy), in which the left part of each product is a single nonterminal.

We define a pushdown automaton, which is useful to us in the construction a Turing machine by a context-free grammar. The following theorem is also given, thanks to which we can argue that this algorithm builds Turing machines that recognize exactly context-free languages.

Definition II.3. A pushdown automaton is a finite state machine that has additional stack storage. The transitions performed by the machine are based not only on input and current state, but also on the stack. The formal definition is a seven-tuple: $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ where

- Q is a finite set of states.
- Σ is a finite set which is called the input alphabet.
- Γ is a finite set which is called the stack alphabet.
- δ is a finite subset of $(Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$, the transition relation.
- $q_0 \in Q$ is the start state.
- $Z \in \Gamma$ is the initial stack symbol.
- $F \subseteq Q$ is the set of accepting states.

Theorem II.1. *Languages that are allowed by pushdown automaton coincide with context-free languages [13].*

B. Group theory

In this subsection, we going to define a free group, a group presentation, a word problem, and several basic group theory definitions and theorems leading to it. All the definitions and theorems described in this section can be found in the book by A. Yu. Ol'shanskii [14].

Definition II.4. A set G with an operation $(\cdot)$ given on it is called group if the following is true:

- $\forall a, b, c \in G, (a \cdot b) \cdot c = a \cdot (b \cdot c)$
- $\exists e \in G : \forall a, a \cdot e = e \cdot a = a$
- $\forall a \in G \ \exists a^{-1} \in G : a \cdot a^{-1} = a^{-1} \cdot a = e$

In group theory, cosets are divided according to the position of a group element into right and left cosets. For example, left coset can be defined as $aH = \{ah : h \in H\}$ where $a \in G$, H is subgroup of G. A subgroup N of a group G is called normal if for all elements a of G the corresponding left and right coset are equal, that is, $aN = Na$. Furthermore, the cosets of N in G form a group called the quotient group or factor group, which denotes as G/N.

$$G/N = \{aN : a \in G\}$$

Normal closure S^G of a subset S of a group G is the smallest normal subgroup of a group G that contains the subset S.

$$S^G = \{gsg^{-1} : g \in G, \, s \in S\}$$

The following theorem is one of the central results in group theory, called the fundamental homomorphism theorem.

Theorem II.2. *For any group homomorphism $\phi : G \to G_1$, where G and G_1 are groups, follows that*

$$Im(\phi) \cong G/Ker(\phi)$$

.

Let A be an arbitrary set of symbols, then

$$A^{-1} = \{a^{-1} : a \in A\}$$

and $A \cup A^{-1}$ is a group alphabet. ω is word in alphabet $A \cup A^{-1}$ if $\omega = a_1 \dots a_n$, where $a_i \in A \cup A^{-1} \ \forall i$ from 1 to n. The word is reducible if there is i and letters x_i and x_{i+1} are mutually inverse.

Definition II.5. The set $G(A)$ of all irreducible words in the alphabet $A \cup A^{-1}$ is a group for the concatenation operation followed by reduction and is called free group.

Since in any group one can choose some system of generators $\{g_i\}$ then there is a surjective homomorphism

$$\phi : G(A) \to G$$

such that $\phi(a_i) = g_i$, where $\{a_i\} = A$. And by applying the fundamental homomorphism theorem (see Theorem II.2) we obtain an isomorphic factor group of a free group with respect to the kernel of the homomorphism:

$$G \cong G(A)/Ker(\phi)$$

We turn to the possibility of defining every group as a quotient of a free group by some normal subgroup. Since it is possible to determine a factor group from any normal subgroup, it means that the possibility of defining each group as a factor of a free group by some normal subgroup is interesting. In this regard, methods of defining a normal subgroup in a free group are interesting, that is, methods of defining a kernel of a homomorphism.

To define the equivalence relation in the words of a free group $G(A)$ let describe elementary transformations of words that do not change a coset of a word in a subgroup $R^{G(A)}$ where R is a subset of $G(A)$ as follows:

1) Reduction.
2) Replacing the word $\omega = \omega_1 r^{\pm} \omega_2$ where $r \in R$ with the word $\omega_1 \omega_2$ and vice versa.

Then the any words ω_1 and ω_2 are equivalent if we can bring one to the other with the help of these transformations.

Definition II.6. Set of relations $\{r = 1 : r \in R\}$ is called determining for the group $G = \langle g_i \rangle$ if any other relation between g_i follows from the system $\{r = 1 : r \in R\}$. If relations $\{r = 1 : r \in R\}$ is determining for the group G then

$$G \cong G(A)/R^{G(A)}$$

and then we can say that G has a presentation $\langle A \mid R \rangle$.

C. *Relation between formal languages and groups*

Consider the relation between formal languages and groups. Note that we obviously assume the fulfillment of group axioms. Let Σ be the finite alphabet of letters, $(\cdot)$ be the concatenation of two words, and ε is an empty word, then

- $(\Sigma^+, \cdot)$ — free semigroup.
- $(\Sigma^*, \cdot, \varepsilon)$ — free monoid.
- $((\Sigma \cup \Sigma^{-1})^*, \cdot, \varepsilon)$ — free group.

In addition to this obvious connection, there is the concept of the word problem of a finitely generated group, which can be formulated as follows.

Definition II.7. Word problem for a finitely generated group G is an algorithmic deciding problem, whether two words consisting of generators represent the same element. Often it is reduced to the equality of words to identity element, which is equivalent.

More precisely, if A is a finite set of generators for G, and A^{-1} is the set of its inversions, then the word problem is the problem of belonging to the formal language of all words from $\Sigma = A \cup A^{-1}$ to the group unit under the mapping by the natural homomorphism $\phi : \Sigma^* \to G$ (see Fig. 1). Set of these words can be written as follows:

$$W(G) = \phi^{-1}(1) = \{\omega \in \Sigma^* : \phi(\omega) = 1\}$$

Moreover, there is a word problem with several statements that have been already proved to show the algebraic properties of groups correspond to the properties of formal languages. For example, for a finitely generated group, the following is true:

- Word problem for G is decidable $\iff$ $W(G)$ is a recursive language.
- $W(G)$ is a regular $\iff$ G is a finite [15].
- $W(G)$ is a context-free $\iff$ exists H: a free subgroup of finite index of G [16].

D. *Group presentation construction algorithm*

The work described here is based on one study of connections between asymptotic functions of groups and computational complexity [11]. In particular, the authors show how to construct a finitely presented group with an NP-complete word problem. To do that they are proof several theorems that we present in this subsection.

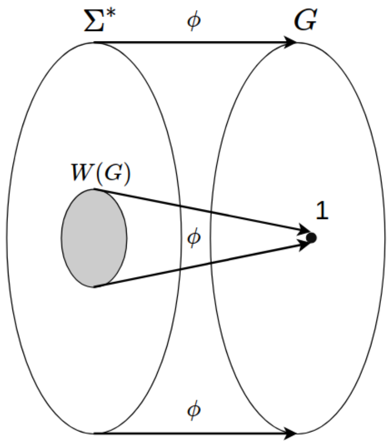


Fig. 1. Word problem of group.

1) Turing machine: Sapir, Birget, and Rips use the following notation for Turing machines [11].

Definition II.8. A Turing machine has k tapes and k heads. It can be described as six-tuple: $M = \langle X, \Gamma, Q, \Theta, \overline{s_1}, \overline{s_0} \rangle$ where

- X is the input alphabet.
- Γ is the tape alphabet.
- $Q = \cup_{i=1}^{k} Q_i$ is the set of states of the heads of the machine.
- Θ is a set of commands.
- $\overline{s_1}$ is the k-vector of start states.
- $\overline{s_0}$ is the k-vector of accept states.

Also, authors assume that the leftmost (rightmost) square on every tape is always marked by α (ω), but they don't actually mean that it and state of the head belong to the word written on tape. At every moment the head observes two squares on each tape to the right and left of it.

A configuration of tape is the word $\alpha u q v \omega$ where q is the current state letter of the head, u (v) is the word to the left (right) of the head. The start configuration is the configuration in which the word recorded on the first tape from X^+, all other tapes are empty, the head sees the right marker ω, and states form starting vector $\overline{s_1}$. The receiving configuration is any configuration in which a state vector $\overline{s_0}$.

A command of a Turing machine is determined by the states of the heads and some of the $2k$ letters observed by the heads. As a result of a command, we can replace some of these $2k$ letters by other letters, insert new squares in some of the tapes, delete some squares in some tapes and move the head one square to the left (right) concerning some of the tapes. The only constraints are that a machine can insert or delete squares on the tape but just before the ω-sign and no letter can be inserted to the left (right) of α (ω) and the end markers cannot be deleted.

In the following, we present one-tape machine commands that as result replace (1), insert (2), move to left (**??**), move to right (4) and delete (5) squares.

$$uqv \to u'q'v' \tag{1}$$

$$q\omega \to uq'\omega \tag{2}$$

$$uq \to q'u \tag{3}$$

$$qu \to uq' \tag{4}$$

$$uq\omega \to q'\omega \tag{5}$$

where u, v, $u^{'}$, $v^{'}$ are letters, q, q' are states.

2) Symmetric Turing machine: In further theorems, a class of Turing machines called symmetric Turing machines will be used. We give definitions that can be seen, for example, in [17].

Definition II.9. Turing machine is symmetric if for every command in the set of commands there is its inverted command.

For example, for command

$$uqv \to u'q'v'$$

the inverted command is

$$u'q'v' \to uqv$$

Definition II.10. The symmetrization of a Turing machine M is the symmetric Turing machine M^{sym} obtained from M by adding the inverse τ^{-1} command into a set of commands for each τ command from M.

Definition II.11. Turing machine M is symmetrizable if and only if M and M^{sym} are equivalent (i.e. they accept the same language).

The following theorem is fundamental in the study of the word problem and was obtained E. Post and A. A. Markov independently of each other in 1947.

Theorem II.3. *Every deterministic Turing machine is symmetrizable. On the other hand, not all nondeterministic Turing machines are symmetrizable.*

This means that to obtain an equivalent symmetric machine from a deterministic one, it is enough to simply add inverse commands to the set of commands, but from a nondeterministic one it is not so easy; for this, complex transformations are performed, which will be considered below.

3) S-machine: In addition, the authors introduce a new rewriting system, which they call an S-machine, and define it as follows.

Definition II.12. Let n be a natural number. A hardware of an S-machine is a pair (Y, Q) where

- Y is a n-vector of sets Y_i which elements are called tape letters.
- Q is a $(n+1)$-vector of disjoint sets Q_i which elements are called state letters.

The sets of elements of Q and Y are also disjoint.

With every hardware

$$S = (Y, Q)$$

can be associate the language of admissible words

$$L(S) = Q_1F(Y_1)Q_2 \dots F(Y_n)Q_{n+1}$$

where $F(Y_j)$ is the language of all reduced group words in the alphabet $Y_j \cup Y_j^{-1}$.

The rewriting rules, or S-rules, have the following form:

$$[U_1 \to V_1, \dots, U_m \to V_m]$$

where the following conditions hold:

1) Each U_i is a subword of an admissible word starting with a Q_l-letter and ending with a Q_r-letter.
2) If $i < j$ then $r(i) < l(j)$.
3) Each V_i is also a subword of an admissible word whose Q-letters belong to $Q_{l(i)} \cup \dots \cup Q_{r(i)}$ and which contains a letter from Q_l and from Q_r.
4) Tape letters are not inserted to the left of Q_1-letters and to the right of Q_{n+1}-letter.

To apply an S-rule to a word W means to replace simultaneously subwords U_i by subwords V_i, $i = 1, \dots, m$. After every application of a rewriting rule, the word is automatically reduced.

For example, if a word is

$$q_1aaq_2bq_3$$

and the S-rule is

$$[q_1 \to p_1a^{-1}, q_2bq_3 \to a^{-1}p_2b'q_3]$$

where $q_i \in Q_i$, $p_i \in Q_i$, $a \in Y_1$, $b, b' \in Y_2$. Then the result of the application of this rule is

$$p_1p_2b'q_3$$

Besides, with every S-rule τ we need to associate the inverse S-rule τ^{-1} in the following way: if

$$\tau = [U_1 \to x_1V_1y_1, \dots, U_m \to x_mV_my_m]$$

then

$$\tau^{-1} = [V_1 \to x_1^{-1}U_1y_1^{-1}, \dots, V_m \to x_m^{-1}U_my_m^{-1}]$$

It is worth noting that throughout their work, the authors assume that an S-machine is symmetric; i. e. if an S-machine contains a rewriting rule τ, it also contains the rule τ^{-1}.

4) The theorems used in the construction: Earlier all the necessary definitions were given, and further theorems related to them will be presented, on the evidence of which the algorithm for constructing a presentation of a group by a Turing machine is based.

So, for example, the following theorem states that for any Turing machine M (including non-deterministic) there is an equivalent symmetric Turing machine, but which is not its symmetrization M^{sym}. That is, to construct an equivalent symmetric Turing machine for any Turing machine, more complex constructions are performed than the simple symmetrization in the Theorem II.3. Indeed, Harry R. Lewis and Christos H. Papadimitriou were first who proved a similar theorem [18]. The authors modify the construction of such a machine from [17].

Theorem II.4. *For any Turing machine M that recognizes the language L, there is Turing machine M' with the following properties:*

1) *The language that M' recognizes is L.*
2) *M' is symmetric.*
3) *A machine accepts a word only when all tapes are empty.*
4) *Any command M' or its inverse has one of the following forms for some i:*

$$\begin{aligned}&\{q_1\omega \to q_1'\omega, ..., q_{i1}\omega \to q_{i1}\omega,\\ &aq_i\omega \to q_i\omega, q_{i+1}\omega \to q_{i+1}\omega, ...\}, \qquad (6)\\ &\{q_1\omega \to q_1'\omega, ..., q_{i1}\omega \to q_{i1}\omega,\\ &\alpha q_i\omega \to \alpha q_i\omega, q_{i+1}\omega \to q_{i+1}\omega, ...\} \qquad (7)\end{aligned}$$

where a belongs to the tape alphabet of the tape i, and q_j, q_j' are the states of the tape j.

5) *The letters used on different tapes, including states, belong to non-overlapping alphabets.*

We note that symmetric Turing machines are more similar than usual to a group because of the possibility of inverse computations that are possible in a group. The proof of this theorem allows us to construct a symmetric Turing machine which preserving the recognizable language of the original machine.

In the next step, the authors propose to build an S-machine from a symmetric Turing machine to achieve even greater similarity with groups. There is a "natural" way of transforming a Turing machine into an S-machine. For this, we take a Turing machine M satisfying all the conditions of Theorem II.4, combine all the tapes of the machine M and replace each command

$$aq\omega \to q'\omega$$

on

$$q\omega \to a^{-1}q'\omega$$

, so the M commands become S-rules. Unfortunately, an S-machine constructed in this way does not inherit most of the properties of the original machine. This is because the S-machine alphabet contains inverse characters. Therefore, it turns out that on the one hand, S-machines are much better suited to imitate their work in group relations, than ordinary Turing machines, since the S-machine can work with reverse alphabet characters (and, moreover, S-machines themselves are considered in [19] as group), and on the other hand, when symmetric Turing machine begins to work as an S-machine, the number of accepted words can increase uncontrollably, since it is possible to recognize negative words. From this it becomes obvious that in order to preserve a recognizable language, it is necessary to make additional constructions.

Before starting to build an S-machine, the authors make the following renames. For every $q \in Q$ they denote the word $q\omega$ by F_q and the left marker on tape $\#i$ by E_i. After renaming a symmetric Turing machine rule or their inverses have one of the following forms for some i:

$$\{F_{q_1} \to F_{q_1'}, \dots, aF_{q_i} \to F_{q_i'}, \dots, F_{q_k} \to F_{q_k'}\} \qquad (8)$$

, where $a \in Y$, or

$$\{F_{q_1} \to F_{q_1'}, \dots, E_iF_{q_i} \to E_iF_{q_i'}, \dots, F_{q_k} \to F_{q_k'}\} \qquad (9)$$

Besides, for any configuration

$$C = (E_1u_1F_{q_1}, ..., E_ku_kF_{q_k})$$

of the machine M, they introduce the value of $\sigma(C)$, which is the admissible word for $S(M)$:

$$\begin{aligned}&E(0)\alpha^n x(0)F(0)\\ &E(1)u_1x(1)F_{q_1}(1)E'(1)p(1)\delta^{||u_1||}q(1)r(1)s(1)t(1)\\ &\overline{p}(1)\overline{q}(1)\overline{r}(1)\overline{s}(1)\overline{t}(1)F'_{q_1}(1)\\ &...\\ &E(k)u_kx(k)F_{q_k}(k)E'(k)p(k)\delta^{||u_k||}q(k)r(k)s(k)t(k)\\ &\overline{p}(k)\overline{q}(k)\overline{r}(k)\overline{s}(k)\overline{t}(k)F'_{q_k}(k)\\ &E'(k+1)x(k+1)\omega^nF'(k+1)\end{aligned}$$

where $n = |u_1| + ... + |u_k|$.

To construct an S-machine equivalent to a symmetric Turing machine, the authors additionally introduce eleven small S-machines (hence the new symbols on tape in the configuration above), which are templates for acceptance management of negative characters. Then, to simulate every rule of a symmetric Turing machine of the form (8) start these small S-machines, linking them together with separate S-rules. And all commands of the form (9) are matched with one special S-rule. Thus, the number of rules in the S-machine is much larger than in the equivalent symmetric Turing machine. Here we did not give specific rules for S-machines used in the construction, and additional S-machines, since this takes up a significant part of the article [11] and it makes no sense to rewrite it. Then the following theorem states that for any Turing machine satisfying Theorem II.4, there is an S-machine simulating it.

Theorem II.5. *Let $M = \langle X, Y, Q, \Theta, s_1, s_0\rangle$ be a Turing machine, which satisfies the conditions of Theorem II.4. Let*

$W_0 = \sigma(C_0)$, where C_0 is start configuration of the Turing machine M. Then the configuration C of the machine M is admissible for M if and only if $S(M)$ can transfer $\sigma(C)$ to W_0 by rewriting rules.

At the last step of transforming, the authors finally build the group. Note that they call one of each pair of mutually inverse rules from Θ positive and the other negative. The set of all positive rules is denoted by Θ^+, and the set of all negative rules is denoted by Θ^-.

Theorem II.6. *Let $S(M)$ be the S-machine described in Theorem II.5, then for any positive integer N, there exists a representation of the group $G_N(S)$ with the generated set A and with the set of relations $P_N(S)$, where*

$$A = \bigcup_{i=1}^{17k+6} Q_i \cup \{\alpha, \omega, \delta\} \cup \bigcup_{i=1}^{k} Y_i \cup \{k_j | j = 1, \ldots, 2N\} \cup \Theta^+$$

and the set of relationships consists of the following:

1) *Transition relations.*
   These relations correspond to elements of Θ^+. Let
   $$\tau \in \Theta^+, \tau = [U_1 \to V_1, ..., U_p \to V_p]$$
   Then we include the relations
   $$U_1^\tau = V_1, ..., U_p^\tau = V_p$$
   If for some j from 1 to $17k+6$, letters from Q_j do not appear in any of the U_i, then also include the relations
   $$q_j^\tau = q_j$$
   for each $q_j \in Q_j$.
2) *Auxiliary relations.*
   This is all kinds of relations of the form
   $$\tau x = x\tau$$
   , where $x \in \{\alpha, \omega, \delta\} \cup_{i=1}^{k} Y_i, \tau \in \Theta^+$ and all relations of the form
   $$\tau k_i = k_i \tau$$
   , where $i = 1, ..., 2N, \tau \in \Theta^+$.
3) *The hub relation.*
   Let for each word u denote the word (10).
   $$K(u) = (u^{-1}k_1uk_2u^{-1}k_3uk_4...u^{-1}k_{2N-1}uk_{2N}) \cdot (k_{2N}u^{-1}k_{2N-1}u...k_2u^{-1}k_1u)^{-1} \quad (10)$$
   Then the hub relation is $K(W_0) = 1$.

Using this information, we are ready to introduce the main theorem of the considered article.

Theorem II.7. *Let $L \subseteq \Sigma^+$ be the language accepted by the Turing machine M, then there exists a finitely presented group $G(M) = \langle A \mid R \rangle$ and the injective mapping*

$$K : \Sigma^+ \to (A \cup A^{-1})^+$$

such that:

$$u \in L \iff K(u) \in W(G)$$

The authors described and proved all the constructions in the theorems above, but no one has automated these constructions at the moment, which we are trying to do in this work.

III. Implementation

This section provides a brief description of the algorithm for constructing a group presentation by a formal context-free grammar.

In this work was used the Haskell functional programming language[1], this choice was due to its rich and convenient type system, which we apply to represent algorithmic types such as formal grammar, Turing machine, S-machine or group presentation.

The algorithm was divided into modules, each of which contains the functionality of a specific construction step. In Fig. 2 presents the architecture of the algorithm, where the order of transformation of context-free grammar into the group presentation is shown. In the picture, types are shown as rectangles with text in the middle, modules are shown as rounded rectangles. The difference in color is due to the belonging of the modules to different groups. Brown unites modules that are responsible for converting types, construction from one type to another. By sequentially applying the functions from these modules, the task of constructing a group presentation by context-free grammar is performed. Purple indicates modules that implement for types the class of printing in LaTeX — *ShowLaTeX*. They generate the LaTeX code from the types for ease of perception to understand the construction process. Mint denotes interpreter modules that are needed to check the equivalence of transformations in the context of preservation the recognized language during transitions from machine to machine. Green marks additional modules that output the result of computations for further processing using other means. The types and modules of the algorithm will be discussed in detail below.

A. Implemented data types

In this work, data types of Haskell language were used. Listing 1 is a code listing with data types that represent grammars, Turing machines, S-machines, and group presentations, respectively. It is worth noting that data types were intentionally described in the same way as in the definitions of the corresponding concepts. This was done to maintain the maximum similarity to the article.

B. Construction a Turing machine from context-free grammar

The first part of the algorithm does not connect to the article discussed in the previous section and faced to bridge a gap between context-free grammar and the Turing machine. To achieve that we implement pushdown automation in terms of the Turing machine we introduced in the background (see Def. II.8). The algorithm accepts a formal grammar in Chomsky normal form as input, it is necessary for ease implementation of pushdown automation. As mentioned earlier in Theorem II.1, it allows us to recognize context-free languages and,

[1]https://github.com/YaccConstructor/LangToGroup

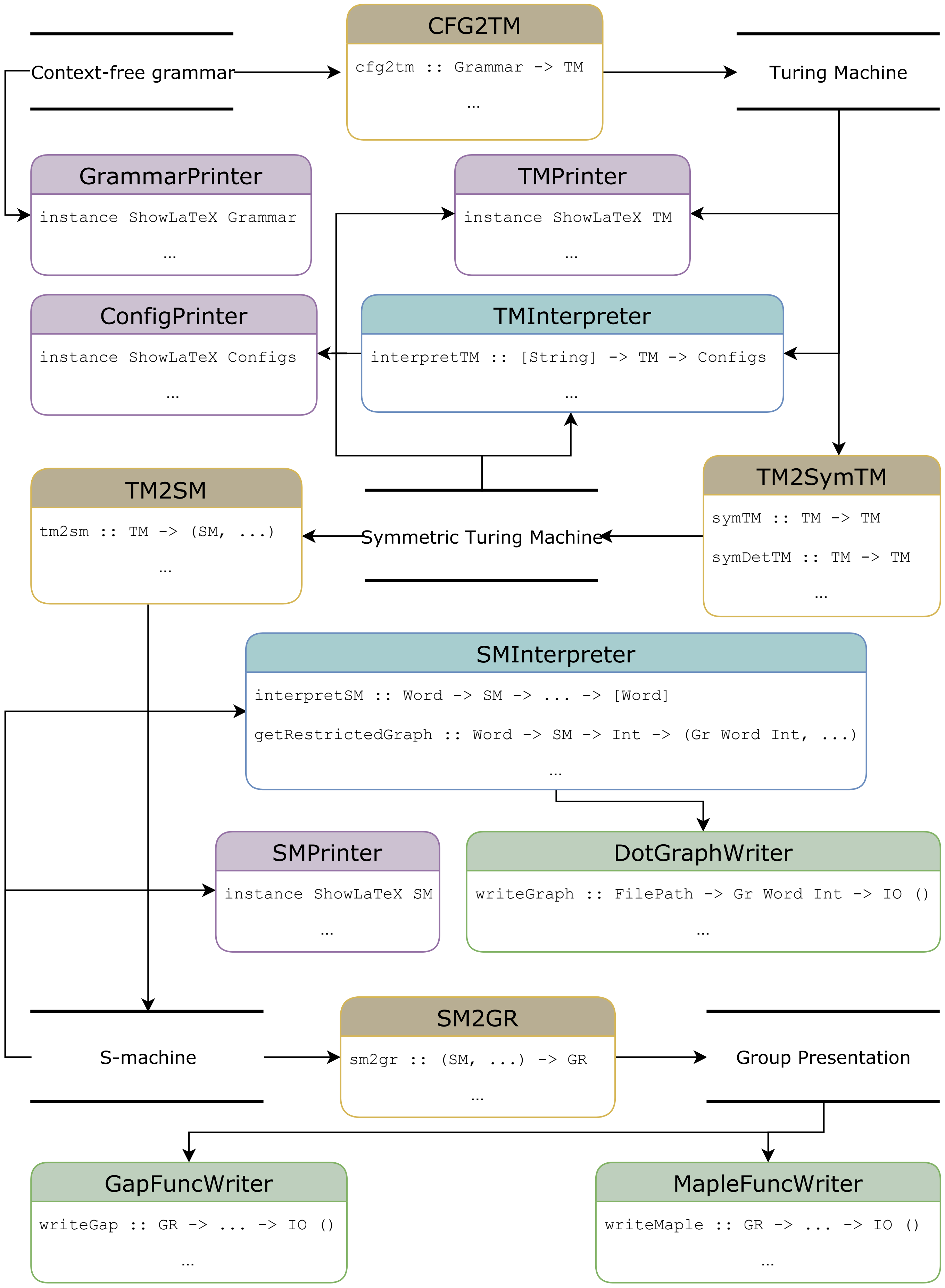
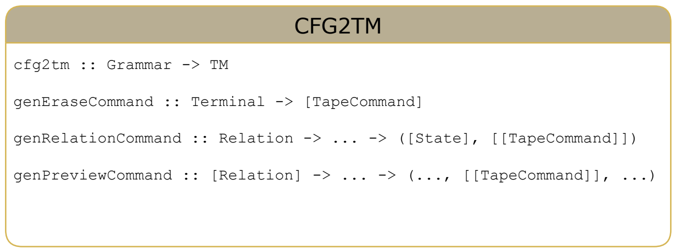


Fig. 2. Architecture of the solution

finally, with a Turing machine recognizing context-free languages, it is possible to construct a group presentation for context-free languages (see Theorem II.7). Thus, within the framework of this work, recognizers for more broad classes of languages that in general a Turing machine can recognize (e.g., recursively enumerable, Boolean, conjunctive) are not provided, so it is not currently possible for them to build a group presentation. However, for language classes that are subclassed of context-free, it's possible.

The transformation of a context-free grammar into a Turing machine is implemented in the module **CFG2TM** of the algorithm (see Fig. 3) and can be started using the *cfg2tm* function.


CFG2TM

```
cfg2tm :: Grammar -> TM

genEraseCommand :: Terminal -> [TapeCommand]

genRelationCommand :: Relation -> ... -> ([State], [[TapeCommand]])

genPreviewCommand :: [Relation] -> ... -> (..., [[TapeCommand]], ...)
```

Fig. 3. Scheme of the module for converting context-free grammar into a Turing machine

So, the Turing machine recognizing context-free languages consists of two tapes: the first tape for the input alphabet, the second for simulating a stack of a pushdown automation. $\overline{s_1} = (q_0^1, q_0^2)$, $\overline{s_0} = (q_1^1, q_2^2)$ — vectors of initial and final states respectively.

The first command of the Turing machine places the starting nonterminal on the second tape:

$$q_0^1\omega \to q_0^1\omega$$
$$q_0^2\omega \to Sq_1^2\omega$$

An accepting command is also introduced, which converts the current state vector of the Turing machine into a vector of final states, if the tapes are empty, that is, if the word was successfully accepted:

$$\alpha q_0^1\omega \to \alpha q_1^1\omega$$
$$\alpha q_1^2\omega \to \alpha q_2^2\omega$$

The following commands are grammar dependent and are divided into preview commands, commands generated from grammar rules, and commands received from grammar terminals. Let's consider them in order.

Further, it will be shown that it is advantageous for us to recognize the language if possible by a deterministic Turing machine. There are grammars whose language can be recognized by a deterministic Turing machine, but by which it is impossible to unambiguously immediately determine which rule should be used. As an example, consider the grammar:

$$S \to aSb$$
$$S \to ab$$

Seeing only the letter a on the tape, it is impossible to determine by one rule which of these two rules should be applied, since the first letter of the right side of the rules is the same. Therefore, rules are generated in the *genPreviewCommand* function that perform previewing one character forward to determine which rule should be applied next. The

```
-- formal grammar
newtype Grammar = Grammar
                        (Set Nonterminal, -- set of nonterminals
                        Set Terminal, -- set of terminals
                        Set Relation, -- set of relations
                        StartSymbol) -- start nonterminal
-- Turing machine
newtype TM = TM
                (InputAlphabet, -- input alphabet
                [TapeAlphabet], -- tapes alphabets
                MultiTapeStates, -- sets of tapes states
                Commands, -- set of commands
                StartStates, -- vector of start states
                AccessStates) -- vector of access states
-- S-machine
data SM =  SM
            {yn :: [[Y]], -- tapes alphabets
            qn :: [Set State], -- sets of tapes states
            srs :: [SRule]} -- list of rules
-- group presentation
newtype GR =  GR
                (Set A, -- set of generators
                Set GrRelation) -- set of relations
```

Listing 1: Base data types using in the work

genPreviewCommand function in our example generates the following Turing machine commands:

$$\begin{matrix} aq_0^1 \to q_2^1 a \\ Sq_1^2\omega \to Sq_1^2\omega \end{matrix},$$

$$\begin{matrix} aq_2^1 a \to aq_5^1 a \\ Sq_1^2\omega \to Sq_1^2\omega \end{matrix}, \quad \begin{matrix} bq_2^1 a \to bq_3^1 a \\ Sq_1^2\omega \to Sq_1^2\omega \end{matrix},$$

$$\begin{matrix} q_5^1 a \to aq_5^1 \\ Sq_1^2\omega \to Sq_1^2\omega \end{matrix}, \quad \begin{matrix} q_3^1 a \to aq_3^1 \\ Sq_1^2\omega \to Sq_1^2\omega \end{matrix},$$

$$\begin{matrix} aq_5^1\omega \to aq_6^1\omega \\ Sq_1^2\omega \to Sq_1^2\omega \end{matrix}, \quad \begin{matrix} aq_3^1\omega \to aq_4^1\omega \\ Sq_1^2\omega \to Sq_1^2\omega \end{matrix}.$$

Thus, if after the letter a there is again the letter a, then the first tape of the Turing machine will go to the state q_6^1, and if there is the letter b, then to the state q_4^1, and this is how determinism of the Turing machine is obtained by encoding information about the next letter in state.

Next, let's consider what commands of the Turing machine the *genRelationCommand* function generates. Let the grammar have a rule $A \to aA'$, then, to simulate this rule, it is necessary to replace A on the second tape with symbols from the right side of the rule. Thus, the rule $A \to aA'$ yields the following Turing machine commands:

$$\begin{matrix} aq_0^1\omega \to aq_0^1\omega \\ Aq_1^2\omega \to A'q_3^2\omega \end{matrix}, \quad \begin{matrix} q_0^1\omega \to q_0^1\omega \\ q_3^2\omega \to aq_4^2\omega \end{matrix}, \quad \begin{matrix} q_0^1\omega \to q_0^1\omega \\ q_4^2\omega \to q_1^2\omega \end{matrix}.$$

If the right side of the rule consists of one character, for example, $A \to a$, then there is no need to generate a new state for writing to the stack, and the following Turing machine command can be generated:

$$\begin{matrix} aq_0^1\omega \to aq_0^1\omega \\ Aq_1^2\omega \to aq_1^2\omega \end{matrix}.$$

In the case when the grammar contains a rule in which the right-hand side is an empty word, it is necessary to generate Turing machine commands differently. For example, consider the grammar

$$\begin{gathered} S \to \varepsilon \\ S \to aSb \end{gathered}$$

, then for the rule $S \to \varepsilon$ the *genRelationCommand* function will generate the following Turing machine commands:

$$\begin{matrix} \alpha q_0^1\omega \to \alpha q_0^1\omega \\ q_0^2\omega \to Sq_1^2\omega \end{matrix}, \quad \begin{matrix} \alpha q_0^1\omega \to \alpha q_0^1\omega \\ Sq_1^2\omega \to q_1^2\omega \end{matrix}, \quad \begin{matrix} bq_0^1\omega \to bq_0^1\omega \\ Sq_1^2\omega \to q_1^2\omega \end{matrix}.$$

The first one pushes S onto the stack at the start states and an empty input tape, the second applies the rule if the input tape is empty, and the third, if the next character on the tape is b.

Now consider the Turing machine commands that correspond to the grammar terminals and execute input word acceptance function. For some character of the input alphabet to be recognized by a Turing machine, it is necessary that it be observed on the input and stack tapes. It is believed that the Turing machine accepts a certain word if, after the machine is running, its tapes are empty and it is in final state. Thus, the process of recognizing a character is to erase that character. from two tapes simultaneously. So, the following command corresponds to any terminal a:

$$\begin{matrix} aq_0^1\omega \to q_0^1\omega \\ aq_1^2\omega \to q_1^2\omega \end{matrix}.$$

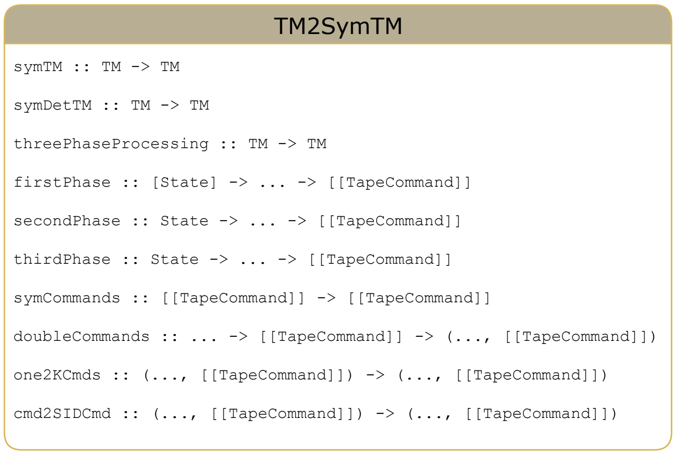
The function that maps each terminal to a rule of this kind is called *genEraseCommand*.

Thus, using the functions considered here, the context-free grammar is transformed into a Turing machine that recognizes the language of this grammar. In addition, the Appendix A contains an example of building a Turing machine using this algorithm.

C. Symmetrization of the Turing machine

The next step after constructing a Turing machine is its symmetrization. An equivalent symmetric Turing machine is constructed in the **TM2SymTM** module (see Fig. 4).

In the course of this work, two algorithms for symmetrization of the Turing machine were implemented. One of them is based on the Theorem II.4, which was proved in the article [11], and the other on the Theorem II.3. The first algorithm can construct an equivalent symmetric Turing machine for both deterministic and nondeterministic Turing machines. Note that this algorithm in this work is called the symmetrization algorithm for a nondeterministic Turing machine, since deterministic Turing machines are a subclass of nondeterministic ones. The second algorithm can output an equivalent symmetric Turing machine only if a deterministic Turing machine was fed to it, but the size and complexity of the resulting machine is much smaller, what will be shown below. Let's consider these two algorithms.



```
symTM :: TM -> TM

symDetTM :: TM -> TM

threePhaseProcessing :: TM -> TM

firstPhase :: [State] -> ... -> [[TapeCommand]]

secondPhase :: State -> ... -> [[TapeCommand]]

thirdPhase :: State -> ... -> [[TapeCommand]]

symCommands :: [[TapeCommand]] -> [[TapeCommand]]

doubleCommands :: ... -> [[TapeCommand]] -> (..., [[TapeCommand]])

one2KCmds :: (..., [[TapeCommand]]) -> (..., [[TapeCommand]])

cmd2SIDCmd :: (..., [[TapeCommand]]) -> (..., [[TapeCommand]])
```

Fig. 4. Scheme of the symmetrization module of the Turing machine

1) Symmetrization Algorithm for a Nondeterministic Turing Machine: The entire algorithm can be run with the *symTM* function, which integrates the functions that will be discussed later.

So, the first phase of the algorithm, as in the article [11], implies the construction of a new Turing machine M', which is equal to the original one except for the added tape $k+1$, where k is the number of tapes at the initial machine such that its alphabet consists of the commands of the initial Turing machine. After that, the commands of the Turing machine are generated, which fill the $k+1$ tape non-deterministically, the only restriction here is that the first added command must complete the computation of the original machine. Thus, after the first phase on the $k+1$ tape of the Turing machine M'

there should be a sequence of commands of the original Turing machine, where the first command completes its computation (transforms the state vector into the vector of final states). After that, the transition to the second phase occurs.

The second phase is that M' tries to execute commands in the order in which they appear on the $k+1$ tape. To do this, from the command of the original Turing machine:

$$\tau = \{U_1 \to V_1, ..., U_k \to V_k\}$$

following command of M' is being built:

$$\tau' = \{U_1 \to V_1, ..., U_k \to V_k, \tau q \to q\tau\}$$

where q is the state of M' on the tape $k+1$.

The authors claim that after such calculations, the first tape M' will be empty, and $k+1$ on the right will observe α only if, during the first phase, a sequence of commands of the original Turing machine was written on the $k+1$ tape, which brings it to its final states.

After the second phase, M' returns the head to ω on the $k+1$ tape and the third phase begins, which consists in clearing all the tapes and transitioning to the vector of final states.

These three phases are implemented in functions with the corresponding names *firstPhase*, *secondPhase*, *thirdPhase*, and the *threePhaseProcessing* function integrates them and builds M' after these three phases, at the end of this paper in the Appendix B, you can see the result of this function on a Turing machine from one rule.

As stated in [17], it is now possible to construct an equivalent symmetric Turing machine from M' by adding its inverse to the set of commands for each command. In the algorithm, this is done by the *symCommands* function. Thus, after these steps M' satisfies the first three properties from the II.4 theorem, and it remains to transform it so that it satisfies the last two properties.

To achieve the fourth property, the tapes are doubled. So for any tape i there appears a tape $i + 1/2$, such that if the initial configuration of the tape i is $\alpha uqv\omega$, then after doubling the configuration of the tapes i and $i + 1/2$: $\alpha uq\omega$ and $\alpha\bar{v}q\omega$, respectively, where u, v are words, $\bar{v}$ is v, written from right to left, q state, which is further renamed to achieve property 5. In this regard, it is necessary to replace every command:

$$\{u_1q_1v_1 \to u_1'q_1'v_1', ..., u_kq_kv_k \to u_k'q_k'v_k'\}$$

to the command:

$$\{u_1q_1\omega \to u_1'q_1'\omega, \bar{v}_1q_1\omega \to \bar{v}_1'q_1'\omega, ..., \\ u_kq_k\omega \to u_k'q_k'\omega, \bar{v}_kq_k\omega \to \bar{v}_k'q_k'\omega\}$$

If v in any of the commands is equal to ω, then $\bar{v} = \alpha$. Obviously, a Turing machine that is transformed in this way will recognize the same language as the original Turing machine. These conversions are implemented in the *doubleCommands* function.

The next step is to split each command into $2k$ commands so that each new command affects only one tape. This operation is performed in the *one2KCmds* function. To this end, many unique states are generated for each command to maintain the order of computation. Thus, the following commands are obtained:

$$\{q_1\omega \to q_1'\omega, ..., u_iq_i\omega \to u_i'q_i'\omega, ...\}$$

Note that if $u_i = \alpha$, then the command already has the form (7), since u_i' is also equal to α, otherwise you need to double the command again so that they look like (6) and this is done by the *cmd2SIDCmd* function:

$$\{q_1\omega \to q_1''\omega, ..., u_iq_i\omega \to q_i''\omega, ...\}$$

$$\{q_1''\omega \to q_1'\omega, ..., q_i''\omega \to u_i'q_i'\omega, ...\}$$

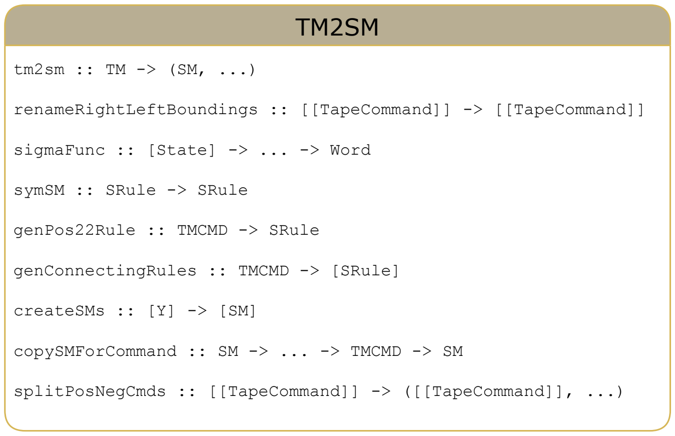
Thus, a Turing machine is constructed that satisfies 4 property of the Theorem II.4. To satisfy property 5, it is enough to generate unique names for states and letters, which happens in the algorithm during the construction process.

2) Symmetrization Algorithm for a Deterministic Turing Machine: Note that the first three phases of the symmetrization algorithm of a nondeterministic Turing machine are aimed at building an equivalent symmetric Turing machine and, if we have an alternative version of its construction, these actions can be omitted.

As it was written above, according to the Theorem II.3 it is possible to get an equivalent symmetric Turing machine from a deterministic Turing machine by simply adding its inverse to the set of commands for each command. But properties 4 and 5 of Theorem II.4 still need to be provided in a symmetric machine in order to proceed to the next step of constructing a group presentation. Therefore, to successfully build an equivalent symmetric Turing machine based on a deterministic Turing machine, it is sufficient to execute the *symDetTM* function, which integrates the *doubleCommands*, *one2KCmds*, *cmd2SIDCmd* and *symCommands* functions, discussed above and guaranteeing the execution of properties 4 and 5. In addition, in the Appendix B, you can see an example of how the algorithm works.

D. Construction of an S-machine from a symmetric Turing machine

In this subsection, we will consider an algorithm for constructing an S-machine that simulates a symmetric Turing machine and, in a sense, preserves the language recognized by this Turing machine. The module, which implements the functions necessary for construction, is called **TM2SM**, and its diagram is shown in Fig. 5. The function that constructs the S-machine is called *tm2sm*, further we consider the functions, the superposition of which it is.



```
tm2sm :: TM -> (SM, ...)

renameRightLeftBoundings :: [[TapeCommand]] -> [[TapeCommand]]

sigmaFunc :: [State] -> ... -> Word

symSM :: SRule -> SRule

genPos22Rule :: TMCMD -> SRule

genConnectingRules :: TMCMD -> [SRule]

createSMs :: [Y] -> [SM]

copySMForCommand :: SM -> ... -> TMCMD -> SM

splitPosNegCmds :: [[TapeCommand]] -> ([[TapeCommand]], ...)
```

Fig. 5. Diagram of a module of constructing an S-machine that simulates a Turing machine

As it was already mentioned in the background section, before immediately building the S-machine, the instructions of the Turing machine are renamed so that they have one of the forms (8), (9). This action in this work is performed by the *renameRightLeftBoundings* function. Also, 11 auxiliary S-machines are built in the *createSMs* function: S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9 , S_α, S_ω, and S-rules are generated in the *genConnectingRules* function, which are transitional between auxiliary machines.

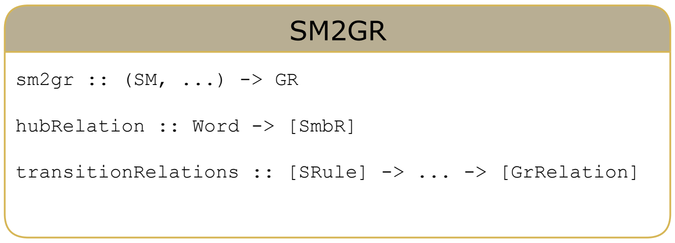
After that, each command of a Turing machine of the form (8) is matched with a set of S-rules of auxiliary S-machines and transitional S-rules, and all commands of the form (9) are matched with one S-rule that does not perform calculations but changes state. The matching is done by adding to the S-rules a label of a particular Turing machine command.

The search for commands of the form (8), (9) is performed by the *splitPosNegCmds* function, the comparison of sets of S-rules to a command of the form (8) is performed by the function *copySMForCommand*, and the generation of S- rules for a command of the form (9) function *genPos22Rule*. In addition, the S-machine symmetrization function *symSM* and the $\sigma(C)$ (*sigmaFunc*) function, the meaning of which was described in the background, are implemented.

The states of the resulting S-machine are all the states of the labeled auxiliary S-machines, combined with all also labeled E_i and F_q, which were obtained after renaming the Turing machines, where $i = 1..k$ is the tape number, $q \in Q$ state of the Turing machine. The S-machine alphabet are the combined alphabets of the auxiliary S-machines with the Turing machine alphabet.

E. Constructing a Group Presentation from an S-Machine

The process of constructing a group representation on an S-machine has been described in detail in Theorem II.6. In this subsection, we'll show you what functions do this build. **SM2GR** — the module that contains these functions (see Fig. 6).



```
sm2gr :: (SM, ...) -> GR

hubRelation :: Word -> [SmbR]

transitionRelations :: [SRule] -> ... -> [GrRelation]
```

Fig. 6. Diagram of a module of constructing a group presentation by an S-machine

Transitional, auxiliary relations and the hub relation are generated in the *transitionRelations*, *auxiliaryRelations* and *hubRelation* functions, respectively. Genertors of the group presentations are forming in the *sm2gr* function, which also unifies all relations.

F. Supporting tools

The subsection describes additional modules of the application developed in this work, which were written, firstly, in order to facilitate perception and, secondly, to test the equivalence of transformations in the context of preserving the recognized language during transitions from machine to machine and which can be seen above in Fig. 2.

For the first, modules were developed that generate LaTeX code, with the ability to generate it for displaying grammars, Turing machines, configurations of Turing machines, S-machines. For this generation, the HaTeX[2] library was used. The *ShowLaTeX* class was created and implemented for all types that needed to be printed in LaTeX. Thus, modules in the Haskell language **GrammarPrinter**, **TMPrinter**, **SMPrinter** and **ConfigPrinter** were implemented, in which the *ShowLaTeX* class is implemented for the *Grammar*, *TM*, *SM* and *Configs* respectively. Examples of this generation can be seen in the Appendix A and B.

For the second, modules were written that interpret Turing machines and S-machines separately on given input data, the general concept of which is breadth-first search of the configuration tree with the stopping if the final configuration is in the front set. So, when traversing in breadth-first, to each configuration from the front, it is necessary to apply the commands of the Turing machine or S-rules that can be applied to this configuration, and the resulting configurations, after application, add to the front set.

Such breadth-first traversal works well in the case of a Turing machine, and in the S-machine interpreter, due to its strong non-determinism, a set was also used in which the traversed configurations are stored. If, after applying the rule, the resulting configuration is already in this set, then such a configuration is not added to the front. This eliminates the possibility of returning to the configurations already passed, which makes it easier to find the final configuration.

[2]HaTeX is a library that implements LaTeX syntax and several useful abstractions.
Library description: https://hackage.haskell.org/package/HaTeX-3.22.2.0
Date of last visit: 14.05.20

The result of the work of the interpreters is the history of the passed configurations, which led to the final one and from which LaTeX can be generated if necessary.

The functions for interpreting the Turing machine are contained in the **TMInterpreter** module, where the *interpretTM* function performs the interpretation. Similarly, the S-machine interpreter module **SMInterpreter**, where *interpretSM* interprets. In addition, this module contains the *getRestrictedGraph* function, which builds a graph of a given height. This graph can then be printed to a dot file using the *writeGraph* function of the **DotGraphWriter** module, which prints it using the Graphviz[3] library.

In addition, **GapFuncWriter** and **MapleFuncWriter** modules were implemented that have *writeGap* and *writeMaple* functions which print the group presentation into the source file of the mathematical packages GAP[4] and Maple[5] respectively, for further analysis of the groups presentations by them.

Thus, we have the opportunity to visually demonstrate the result of the algorithm and check whether the recognized languages match after transitions.

IV. Evaluation

This section presents the results of the pipeline execution of algorithms for constructing a presentation of a group from context-free grammars in Chomsky normal form with the derivation of the numerical sizes of machines at all transformation steps.

To estimate the size of the resulting group presentation, we ran algorithms with non-deterministic and deterministic symmetrization on three grammars: one rule grammar (11), grammar of "a*" language (12) and unambiguous grammar of Dick language on one type of brackets (13).

$$S \to a \tag{11}$$

$$\begin{aligned} S &\to AS \mid \varepsilon \\ A &\to a \end{aligned} \tag{12}$$

$$\begin{aligned} S &\to AC \mid \varepsilon \\ C &\to SD \\ D &\to BS \\ A &\to a \\ B &\to b \end{aligned} \tag{13}$$

Table I shows the sizes of each of the sets of our algebraic types, built in the process of the algorithm with nondeterministic symmetrization of the Turing machine. It is obvious from it that an algorithm was obtained that builds a rather large presentation of the group even on simple initial data. So, grammar from just one rule turns into about 90 thousand group relations and 56 thousand generators.

[3]Graphviz is an open source graph rendering toolkit.
Library description: https://hackage.haskell.org/package/graphviz
Date of last visit: 14.05.20

[4]GAP (Groups, Algorithms, Programming) is a software package, an open source computer algebra system with a special emphasis on computational group theory.
Package site: https://www.gap-system.org/
Date of last visit: 14.05.20

[5]Maple is a software package, a computer mathematics system focused on complex mathematical calculations, data visualization and modeling.
Package site: https://www.maplesoft.com/products/Maple/
Date of last visit: 14.05.20

But analyzing this table, you can see that the main growth in size occurs at the stages of symmetrization of the Turing machine and transformation of the symmetric Turing machine into an S-machine. Therefore, in order to reduce the size of the symmetric Turing machine, the symmetrization algorithm of the deterministic Turing machine was implemented, which is also discussed above. The Table II shows the results of its work.

Note that, indeed, the use of this algorithm reduced the size of the symmetric Turing machine, the S-machine, and the group presentation by several times in comparison with the use of the symmetrization algorithm for nondeterministic Turing machines. It is clear that the smaller the group presentation, the easier it is to find words that are equal to the group unit. From this we can conclude that in the case of deterministic grammars, one should use symmetrization for deterministic machines.

Attempts were also made to check for equality to groups' unit some words from the obtained groups presentations using the mathematical packages GAP and Maple. Unfortunately, due to the complexity of the calculations, the mathematical packages with which the experiments were carried out did not complete their work in an acceptable time.

V. Conclusion

During this work, an algorithm for transforming a context-free grammar into a Turing machine of a special notation was implemented, on the basis of the article [11] an algorithm for constructing a group presentation by a Turing machine was developed, interpreters for a Turing machine and an S-machine were developed, and a number of construction experiments were carried out, which showed advantages of using the algorithm of deterministic symmetrization of Turing machines.

Since the described solution does not currently have its application, there is a need to improve it, namely, if you want to get a presentation of a group of satisfactory sizes, then in further work it is necessary to find out how to transform symmetric Turing machines into S-machines in a more optimal way.

We can argue that it is possible to find a more optimal algorithm for constructing an S-machine from a symmetric Turing machine for a number of reasons. Firstly, there are alternative ways of constructing an S-machine from the symmetric Turing machine [12]. Secondly, since some languages can be used to construct a group presentation, for example, for the language $L = \{a\}$, we can construct a group presentation and a relation from the Theorem II.7 such that

$$G = \langle k_1, k_2, a \mid k_1 a k_2 = 1 \rangle$$

and $K(a) = k_1 a k_2$. It is clear that such a construction works only for finite languages, but this suggests the idea that, perhaps, there is a simpler construction in the general case.

In addition, the question of interpreting the group's presentation remains relevant, since the existing mathematical packages could not cope with this task.

APPENDIX A
TRANSFORMATION OF THE GRAMMAR

This section provides an example of the work of the above algorithm for converting the elementary grammar into a Turing machine with its subsequent interpretation.

From the following one-rule grammar:

Nonterminals

S

Terminals

a

Rules

$S \to a$

The following Turing machine was obtained using the algorithm implemented in the framework of this work:

Input alphabet

a

Tape alphabets

1) a
2) S, a'

States

1) q_0^1, q_1^1
2) q_0^2, q_1^2, q_2^2

Start states

q_0^1, q_0^2

Access states

q_1^1, q_2^2

Commands

$$\begin{bmatrix} aq_0^1\omega & \to & aq_0^1\omega \\ Sq_1^2\omega & \to & a'q_1^2\omega \end{bmatrix}$$

$$\begin{bmatrix} aq_0^1\omega & \to & q_0^1\omega \\ a'q_1^2\omega & \to & q_1^2\omega \end{bmatrix}$$

$$\begin{bmatrix} \alpha q_0^1\omega & \to & \alpha q_1^1\omega \\ \alpha q_1^2\omega & \to & \alpha q_2^2\omega \end{bmatrix}$$

$$\begin{bmatrix} q_0^1\omega & \to & q_0^1\omega \\ q_0^2\omega & \to & Sq_1^2\omega \end{bmatrix}$$

At this point let's explain the notation of the commands. Every Turing machine command is enclosed in square brackets, where each line represents the single-tape Turing machine command that belongs to the corresponding tape. The single-tape Turing machine command has been announced above and in the article [11]. Looking at one of the commands above, you can understand that they refer to a Turing machine with two tapes, and on the first line of the command is a single-tape command related to its first tape, and on the second to the second tape. Thus, the line number corresponds to the tape number of the Turing machine.

Then we interpreted the resulting Turing machine on the input line a and got the history of configuration changes:

Configurations

№	Tape 1			Tape 2		
1	αa	q_0^1	ω	α	q_0^2	ω
2	αa	q_0^1	ω	αS	q_1^2	ω
3	αa	q_0^1	ω	$\alpha a'$	q_1^2	ω
4	α	q_0^1	ω	α	q_1^2	ω
5	α	q_1^1	ω	α	q_2^2	ω

APPENDIX B
TRANSFORMATION OF THE TURING MACHINE

In this section, an example of the operation of three phases of the symmetrization algorithm for non-deterministic Turing machines, namely the *threePhaseProcessing* function, and

	Grammar			**TM**				**TM'**				**SM**			**G**	
	Σ	N	R	X	Γ	Q	Θ	X	Γ	Q	Θ	Y	Q	Θ	A	R
1 rule	1	1	1	1	3	5	4	1	14	270	206	14	88246	2363	89508	56187
a^*	1	2	3	1	4	7	9	1	26	547	434	26	344118	5741	347370	204903
Dyck	2	4	6	2	8	11	19	2	54	1131	900	54	1469136	15064	1478859	957619

TABLE I
CARDINALITIES OF SETS OF MACHINES WHEN USING THE SYMMETRIZATION ALGORITHM OF NONDETERMINISTIC TURING MACHINES

	Grammar			**TM**				**TM'**				**SM**			**G**	
	Σ	N	R	X	Γ	Q	Θ	X	Γ	Q	Θ	Y	Q	Θ	A	R
1 rule	1	1	1	1	3	5	4	1	6	39	34	6	6058	501	6410	7637
a^*	1	2	3	1	4	7	9	1	8	73	72	8	15888	1024	16565	17657
Dyck	2	4	6	2	8	11	19	2	16	161	158	16	67754	2837	69772	71533

TABLE II
CARDINALITIES OF SETS OF MACHINES USING THE SYMMETRIZATION ALGORITHM OF DETERMINISTIC TURING MACHINES

also the symmetrization algorithm for a deterministic Turing machine, is considered.

From a Turing machine containing one rule and accepting a language from one word a:

Input alphabet

a

Tape alphabets

1) a

States

1) q^1_0, q^1_1

Start states

q^1_0

Access states

q^1_1

Commands

$$\begin{bmatrix} aq^1_0\omega & \rightarrow & q^1_1\omega \end{bmatrix}$$

After three phases of symmetrization by the algorithm for nondeterministic machines, we obtain a symmetric Turing machine that does not satisfy the 4th property of the Theorem II.4:

Input alphabet

a

Tape alphabets

1) a
2) $\begin{bmatrix} aq^1_0\omega & \rightarrow & q^1_1\omega \end{bmatrix}$

States

1) q^1_0, q^1_1, q^1_2
2) q^2_0, q^2_1, q^2_2, q^2_3

Start states

q^1_2, q^2_0

Access states

q^1_1, q^2_3

Commands

$$\begin{bmatrix} & & aq^1_0\omega & \rightarrow & & & q^1_1\omega \\ \left[aq^1_0\omega\right. & \rightarrow & \left.q^1_1\omega\right] q^2_2 & \rightarrow & q^2_2 \left[aq^1_0\omega\right. & \rightarrow & \left.q^1_1\omega\right] \end{bmatrix}$$

$$\begin{bmatrix} \alpha q^1_1\omega & \rightarrow & \alpha q^1_1\omega \\ \alpha q^2_2\omega & \rightarrow & \alpha q^2_3\omega \end{bmatrix}$$

$$\begin{bmatrix} & & \alpha q^1_1\omega & \rightarrow & & & \alpha q^1_1\omega \\ q^2_2 \left[aq^1_0\omega\right. & \rightarrow & \left.q^1_1\omega\right] & \rightarrow & \left[aq^1_0\omega\right. & \rightarrow & \left.q^1_1\omega\right] q^2_2 \end{bmatrix}$$

$$\begin{bmatrix} & & \alpha q^1_1\omega & \rightarrow & \alpha q^1_1\omega \\ \left[aq^1_0\omega\right. & \rightarrow & \left.q^1_1\omega\right] q^2_2\omega & \rightarrow & q^2_2\omega \end{bmatrix}$$

$$\begin{bmatrix} q^1_2\omega & \rightarrow & q^1_0\omega \\ q^2_1\omega & \rightarrow & q^2_2\omega \end{bmatrix}$$

$$\begin{bmatrix} q^1_2\omega & \rightarrow & & & q^1_2\omega \\ q^2_0\omega & \rightarrow & \left[aq^1_0\omega\right. & \rightarrow & \left.q^1_1\omega\right] q^2_1\omega \end{bmatrix}$$

You can notice that in the example above, the Turing machine commands are used as symbols on the last tape, the introduction of such a tape is necessary to symmetrize the nondeterministic Turing machine. Here, you should think of commands within a command as simple symbols on the tapes that you can replace, move, add, and remove. They themselves no longer perform the function of commands but are simply symbols that indicate belonging to a particular command. With the same success, it was possible to use any other characters for a similar encoding of a symbol's belonging to a command, but we decided that it would be more descriptive.

And applying the symmetrization algorithm of a deterministic Turing machine to the same elementary machine, we obtain a symmetric Turing machine:

Input alphabet

a

Tape alphabets

1) a
2) a''

States

1) q^1_0, q^1_1
2) $q^{1.5}_0$, $q^{1.5}_1$, $q^{1.5}_2$

Start states

q^1_0, $q^{1.5}_0$

Access states

q^1_1, $q^{1.5}_1$

Commands

$$\begin{bmatrix} aq_0^1\omega & \rightarrow & q_1^1\omega \\ q_0^{1.5}\omega & \rightarrow & q_2^{1.5}\omega \end{bmatrix}$$

$$\begin{bmatrix} q_1^1\omega & \rightarrow & aq_0^1\omega \\ q_2^{1.5}\omega & \rightarrow & q_0^{1.5}\omega \end{bmatrix}$$

$$\begin{bmatrix} q_1^1\omega & \rightarrow & q_1^1\omega \\ \alpha q_1^{1.5}\omega & \rightarrow & \alpha q_2^{1.5}\omega \end{bmatrix}$$

$$\begin{bmatrix} q_1^1\omega & \rightarrow & q_1^1\omega \\ \alpha q_2^{1.5}\omega & \rightarrow & \alpha q_1^{1.5}\omega \end{bmatrix}$$

Unfortunately, the size of S-machines does not allow us to give an example of their construction here, but printing S-machines is possible. Similarly, we cannot give here even the smallest example of transforming a grammar into an S-machine with the inference of intermediate Turing machines. It will take a lot of pages, but you can repeat the experiments yourself, and print it to LaTeX by command-line flags, which are been implemented for this, examples of its use you can find in the readme[6].

References

[1] N. Chomsky, *Some methodological remarks on generative grammar*. Aspects of the Theory of Syntax, 1961.

[2] ——, *Syntactic structures*. Walter de Gruyter, 2002.

[3] J. Hellings, "Querying for paths in graphs using context-free path queries," 2015.

[4] R. Azimov and S. Grigorev, "Context-free path querying by matrix multiplication," in *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences Systems (GRADES) and Network Data Analytics (NDA)*, ser. GRADES-NDA '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3210259.3210264

[5] W. Huber, V. J. Carey, L. Long, S. Falcon, and R. Gentleman, "Graphs in molecular biology," *BMC Bioinformatics*, vol. 8, no. 6, p. S8, Sep 2007. [Online]. Available: https://doi.org/10.1186/1471-2105-8-S6-S8

[6] J. Scott, "Social network analysis," *Sociology*, vol. 22, no. 1, pp. 109–127, 1988.

[7] A. Okhotin, "Conjunctive grammars," *J. Autom. Lang. Comb.*, vol. 6, no. 4, p. 519–535, Apr. 2001.

[8] ——, "Boolean grammars," *Information and Computation*, vol. 194, no. 1, pp. 19 – 48, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0890540104001075

[9] A. Meduna and O. Soukup, *Modern Language Models and Computation: Theory with Applications*, 01 2017.

[10] A. Okhotin, "Conjunctive and boolean grammars: The true general case of the context-free grammars," *Computer Science Review*, vol. 9, pp. 27 – 59, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S157401371300018X

[11] M. V. Sapir, J.-C. Birget, and E. Rips, "Isoperimetric and isodiametric functions of groups," *Annals of Mathematics*, vol. 156, no. 2, pp. 345–466, 2002. [Online]. Available: http://www.jstor.org/stable/3597195

[12] A. Ol'shanskii, "Space functions of groups," *Transactions of the American Mathematical Society*, vol. 364, 10 2010.

[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation: Pearson New International Edition*. Pearson Higher Ed, 2013.

[14] A. Ol'shanskii, *Geometry of defining relations in groups*. Springer Science & Business Media, 2012, vol. 70.

[15] A. V. Anisimov, "Languages over free groups," in *Mathematical Foundations of Computer Science 1975 4th Symposium, Mariánské Lázně, September 1–5, 1975*, J. Bečvář, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 167–171.

[16] D. Muller and P. Schupp, "Context-free languages, groups, the theory of ends, second-order logic, tiling problems, cellular automata, and vector addition systems," *Bulletin of the American Mathematical Society*, vol. 4, 07 1981.

[17] J.-C. Birget, "Time-complexity of the word problem for semigroups and the higman embedding theorem," *International Journal of Algebra and Computation*, vol. 08, no. 02, pp. 235–294, 1998. [Online]. Available: https://doi.org/10.1142/S0218196798000132

[18] H. R. Lewis and C. H. Papadimitriou, "Symmetric space-bounded computation," *Theoretical Computer Science*, vol. 19, no. 2, pp. 161 – 187, 1982. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397582900585

[19] A. Olshanskii and M. Sapir, "Groups with small dehn functions and bipartite chord diagrams," 2004.

[6]https://github.com/YaccConstructor/LangToGroup/blob/master/README.md