Automatic generation of regular languages from regular grammars

Saša V. Vukašinović
University of Niš, Faculty of Science, Department of Mathematics, Višegradska 33, 18000 Niš, Serbia
sasamike@bitinfo.co.yu

Predrag S. Stanimirović
University of Niš, Faculty of Science, Department of Mathematics, Višegradska 33, 18000 Niš, Serbia
pecko@pmf.ni.ac.yu

We develop a software for the characterization of regular grammars and construction of regular expressions by those grammars. Also, for an arbitrary expression we test whether or not it can be generated by a given regular grammar, and develop corresponding transformation rules which produce the considered expression.

Software is implemented in the programming language MATHEMATICA, using mainly the following useful possibilities: functional programming, symbolic processing, working with lists and, particularly, using self-recursion in work with functions.
Introduction and Preliminaries

In this paper we will represent an algorithm for automatic generation of an union of all expressions generated by a given regular grammar. This union makes a regular language corresponding to the regular grammar. In this section we will give theoretical background of our main subject. At first, we will restate known definitions of various types of grammars and restate classification of all grammars. Next, we will define languages set corresponding to those grammars and give a definition of regular languages and some theorems whose will be needful for the implementation of our algorithm. In the third section we will describe the implementation of our algorithm. This algorithm has four functional components. The first function does elimination of all grammars whose are not regular and pass only regular ones. The main function calls a self–recursive function which is, in reality, "heart" of the algorithm. This function forms expressions of the language and memorizes positions of separators in each single expression. Finally, the fourth function does definitive form of all expressions arising from the axiom of the grammar, inserts separators and operation "+" to appropriate places, and provides printing of completely generated language.

Let given a finite non–empty set X, which we will call alphabet, and its elements letters or symbols. A string (word) over the alphabet X is a non–empty finite sequence \( x_1 x_2 ... x_n \) of elements from \( X \).

The number of symbols in a string \( x \) is called its length, and it is denoted by \( |x| \).

The set of all nonempty strings over \( X \) is denoted by \( X^* \). In this set it is defined the operation called concatenation or joining, as following: concatenation of strings \( x = x_1 x_2 ... x_n \) and \( y = y_1 y_2 ... y_m \), where \( x_1, x_2, ..., x_n, y_1, y_2, ..., y_m \in X \), is the string \( xy = x_1 x_2 ... x_n y_1 y_2 ... y_m \). This operation is associative, which means that \( X^+ \) with this operation makes a semigroup. This semigroup we call free semigroup over alphabet \( X \).

It is convenient to introduce the empty string, which contains no symbols, and its length is 0. We use the notation \( e \) for the empty string. Unit extension of semigroup \( X^+ \) by element \( e \) (\( e \in X^+ \)) we call free monoid over alphabet \( X \), and it is denoted by \( X^* \). The element \( e \) is a unit of monoid \( X^* \). In the formal language theory, language over alphabet \( X \) is a subset of \( X^* \) over this alphabet. The members of a language are called the strings or words of language.

**Definition 1.** Let \( L_1 \) and \( L_2 \) be two languages. The concatenation of \( L_1 \) and \( L_2 \), denoted by \( L_1 L_2 \), is the language \( L_1 L_2 = \{ xy \mid x \in L_1 \land y \in L_2 \} \).

**Definition 2.** Formal grammar, or shortly grammar, is the ordered quadruple \( G = (V_T, V_N, S, P) \), where:

1. \( V_T \) is a finite non–empty set called terminal alphabet. The elements of \( V_T \) are called terminals.
2. \( V_N \) is a finite non–empty set disjoint from \( V_T \). The elements from \( V_N \) are called nonterminals or variables.
3. \( S \in V_N \) is a distinguished nonterminal called the start symbol.
4. \( P \) is a finite set of productions or rules. Elements from \( P \) are expressions of the form \( \alpha \to \beta \), where \( \alpha \in (V_T \cup V_N)^* \) \( V_N \cup (V_T \cup V_N)^* \) and \( \beta \in (V_T \cup V_N)^* \), i.e. \( \alpha \) is a string of terminals and nonterminals containing at least one nonterminal and \( \beta \) is an arbitrary string of terminals and nonterminals.

**Definition 3.** Let \( L \) be a language over \( V_T \). Define \( L^0 = \{ e \} \) and \( L^i = LL^{i−1} \) for \( i \geq 1 \). Then the Kleene closure of \( L \), denoted by \( L^* \), is the language
\[
L^* = \bigcup_{i \geq 0} L^i.
\]

The positive closure of \( L \), denoted by \( L^+ \), is the language
\[
L^+ = \bigcup_{i \geq 1} L^i.
\]
Hierarchies of grammars

Let $G = (V_T, V_N, S, P)$ be a grammar.

1. $G$ is called a Type-0 grammar or an unrestricted grammar.
2. $G$ is called a Type-1 grammar or context-sensitive grammar if each production $a \rightarrow \beta$ in $P$ satisfies $|a| \leq |\beta|$.
3. $G$ is of a Type-2 grammar or context-free if each production $a \rightarrow \beta$ in $P$ satisfies $|a|=1$, which means that $a$ is a single nonterminal.
4. $G$ is of a Type-3 or regular (or right-linear) grammar if each production has one of the following three forms:
   
   $A \rightarrow aB, \quad A \rightarrow a, \quad A \rightarrow \epsilon, \quad A \in V_N, a \in V_T$.

Many courses allow right-linear grammars to have productions of the form $A \rightarrow pB$, where $p \in V_T^*$ or $a \rightarrow q$, where $a \in V_N$ and $q \in V_T^*$. Languages generated by those grammars are called regular languages or languages of type 3.

This classification of grammars and languages is known as Chomsky’s hierarchy, who made it.

A sentential form of $G$ is any string of terminals and nonterminals, i.e. a string over $V_T \cup V_N$. For the sentential form $w \in (V_T \cup V_N)^*$ we say to be directly derived from the sentential form $w \in V_T \cup V_N)^*$ and write $w \Rightarrow w'$, if $w = puv$ and $w' = puv'$ and there is a rule $u \rightarrow v$ from $P$. Next, we say that the sentential form $w \in (V_T \cup V_N)^*$ is derivable from the sentential form $w \in V_T \cup V_N)^*$, and write $w \Rightarrow^* w'$, if $w = w'$ is valid or there is an array $w_1,w_2,\ldots,w_n \in (V_T \cup V_N)^*$, $n \geq 2$, such that: $w = w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n = w'$. In this case, sequence $w_1,w_2,\ldots,w_n$ is called the derivation of the sentential form for $w'$ from $w$.

For the nonterminal symbol $S \in V_N$, the set $L(G, \delta) = \{ w \in X^* | S \Rightarrow^* w \}$ is known as the language generated by grammar $G$ starting at $S$. The language $L \subseteq V_T^*$ we call generated by grammar if there is a grammar $G = (V_T, V_N, S, P)$ and nonterminal symbol $S \in V_N$ such that $L = L(G, S)$.

This is basic theoretical background needed for the implementation of the following algorithm. Before we do this, we will give a theorem which has practical role in technical realization of our algorithm.

**Theorem 1.** The language $L \subseteq V_T^*$ over finite alphabet $V_T$ is regular if and only if it is generated by the grammar $G = (V_T, V_N, S, P)$, where each rule in $G$ is in one of the following two forms:

- $A \rightarrow \epsilon$, where $A,B \in V_N$ and $A \in V_T$.
- $A \rightarrow \alpha \beta$, where $A,B \in V_T$.

See Hopcroft et al. (2001).

**Definition 4.** The regular expression over $V_T$ and the language generated by this expression are defined inductively as follows.

1. The symbol $\emptyset$ is a regular expression, and represents the empty language.
2. The symbol $\epsilon$ is a regular expression, and represents the language whose only member is the empty string, i.e. $\{ \epsilon \}$.
3. Each $c \in V_T$ is a regular expression, and represents the language $\{c\}$, whose unique member is the string consisting of the single character $c$.
4. If $r$ and $s$ are regular expressions representing the languages $R$ and $S$, respectively, then $(r+s), (rs)$ and $(r^*)$ are regular expressions that represent languages $R \cup S$, $RS$, and $R^*$, respectively.

Algorithm for automatic generation of regular languages from regular grammars
Our algorithm is made by four functions. Formal parameters of the first function IfRegular are:

- \( VN \) – list of nonterminal symbols,
- \( VT \) – list of terminal symbols,
- \( AKS \) – list of starting symbols or axioms,
- \( VP \) – list of rules and
- \( pe \) – logical indicator.

This function determinates if the grammar, defined by parameters ahead, is regular or not. In this function we set the value to logical variable \( p \) equal to TRUE in the case when given grammar is regular, and FALSE otherwise, and return it. The following code represents the first function:

```plaintext
IfRegular[VN_, VT_, VP_, pe_] :=
   Block[{rul, aux = {}, i, j, k},
       p = pe;
       If[Intersection[VT, VN] == {}, (*then*)
          For[i = 1, i <= Length[VP], i++,
             rule = VP[[i]] /. Rule -> List;
             aux = Characters[ToString[rule[[2]]]]; 
             k = Length[aux];
             For[j = 1, j <= k, j++,
                aux[[j]] = ToExpression[aux[[j]]]];
             PrependTo[aux, rul[[1]]];
             p = p && MemberQ[First[aux], VP];
             For[j = 2, j <= k - 1, j++,
                p = p && MemberQ[VT, aux[[j]]]];
             p = p && (MemberQ[VN, aux[[k]]] || MemberQ[VT, aux[[k]]])
          ];
       (*else*)
       p = False;
       Return[p];
    ]
```

We check the following main criteria for the characterization of the grammar \((VN, VT, AKS, VP)\) to be regular:

1. \( VN \cap VT = \emptyset \);
2. first element in any rule \( \alpha \rightarrow \beta \) is an element from \( VN \);
3. \( \alpha(i) \in VT, i = 1, \ldots, k-1, \alpha(k) \in VN \cup VT, \) where \( k = l \).

For an automatic characterization of regular grammars we use the following:

In the case \( p = \text{False} \) the grammar \((VN, VT, AKS, VP)\) is not regular; otherwise, in the case \( p = \text{True} \), the grammar is regular.

Formal parameters of the main function, named Regular, are:

- \( VN \) – list of nonterminal symbols,
- \( VT \) – list of terminal symbols,
- \( AKS \) – list of starting symbols or axioms, and
- \( VP \) – list of rules.

This function calls the function IfRegular. In case \( p = \text{FALSE} \) program stops and print a proper message. Conversely, function continues by the initialization of variables lrules, clist, cword, s and sep. These variables are formal parameters of the function named Partof, which is called by the function Regular.

All rules from the list \( VP \) of the form \( \alpha \rightarrow \beta \), where \( \alpha = AKS \), are incorporated into the list.
starrules, which can be generated applying the following code:

```math
For[j=1,j<\text{Length}[\text{VP}],j++,
  \text{firstsymbol}=\text{ToString}([\text{VP}[[j]]]/\text{Rule} \rightarrow \text{List}[[1]]);
  \text{If}[\text{firstsymbol}=\text{ToString}[\text{AKS}],\text{AppendTo}[\text{starrules},\text{VP}[[j]]]]
] ;
```

All terminal expressions can be generated inside the cycle

```
For[w=1,w<\text{Length}[\text{starrules}],w++,...
```

in which we use all elements from the list starrules which are capable to generate regular expressions.

```math
\text{Regular}[\text{VN},\text{VT},\text{AKS},\text{VP}]=:
\text{Block}[[p=\text{True},d,\text{rul},\text{start},\text{aux}={},
  \text{auxlist},\text{ls}={},\text{language}="",\text{firstsymbol},
  \text{firstsymbol2},\text{separator1}={},\text{seplist}={},
  \text{setofexp}={},\text{change},i,j,h,n,w,k,\text{wordl}="",\text{listrules}={},\text{checklist}={},\text{starrules}={},l=1],
\text{IfRegular}[\text{VN},\text{VT},\text{VP},\text{True}];
\text{If}[p,\n  \text{Print}["Grammar is regular"];\n  (* Then *)\n  \text{For}[j=1,j<\text{Length}[\text{VP}],j++,
    \text{firstsymbol}=\text{ToString}([\text{VP}[[j]]]/\text{Rule} \rightarrow \text{List}[[1]]);\n    (* Print["
      \text{firstsymbol}="","\text{firstsymbol}];*)\n    \text{If}[\text{firstsymbol}==\text{ToString}[\text{AKS}],\n      \text{AppendTo}[\text{starrules},\text{VP}[[j]]]]\n  ] ;
  \text{Print["starrules = ",\text{starrules}];}\n  \text{For}[w=1,w<\text{Length}[\text{starrules}],w++,
    \text{checklist}={};\n    \text{separator1}={};\n    \text{checklist}=\text{AppendTo}[\text{checklist},\text{starrules}[[w]]];\n    \text{ls}=\text{List}[\text{starrules}[[w]]];\n    \text{start}=\text{ToString}([\text{ls}[[1]]]/\text{Rule} \rightarrow \text{List}[[2]]);\n    (* Print[start];*)\n    \text{aux}=\text{Characters}[\text{start}];\n    \text{sl}=\text{Last}[\text{aux}];\n    \text{wordl}="";\n    \text{If}[\text{MemberQ}[\text{VN},\text{ToExpression}[\text{sl}]],\text{wordl}=\text{wordl}<>\text{StringDrop}[\text{start},-1],\text{wordl}=\text{wordl}<>\text{wordl}];\n    \text{listrules}={};\n    \text{For}[k=1,k<\text{Length}[\text{VP}],k++,\n      \text{firstsymbol2}=\text{ToString}([\text{VP}[[k]]]/\text{Rule} \rightarrow \text{List}[[1]]);\n      \text{If}[\text{\text{sl}}==\text{firstsymbol2},\n        \text{AppendTo}[\text{listrules},\text{VP}[[k]]]];\n    ];\n    \text{VP}[[1]]=\text{Sort}[\text{VP}];\n    \text{VNI1}=\text{VN};\n    \text{VT1}=\text{VP};\n    \text{Partof}[\n      \text{listrules},\text{checklist},\text{wordl},1,\text{separator1}]
  ] ;
  \text{language}="L(\text{VP})="<>\text{language}<>" \emptyset",\n  \text{Return["Grammar is not regular"] (* else *)}
]
```
Goal of the function Partof is to form, in dependence of the list of rules called lrules, a list of words obtained by concatenation of terminals at the next derivation. Also, this function in the same time memorizes all positions of separators through forming is already done. Those positions define usage of the operation "+". Next, function Partof calls, self-recursively, itself and following appearance of nonterminal play an axiom. In this way, it forms a list of words and list of positions of separators whose are formal parameters \( p \) and \( rec \) of the function named RegLang, which be called in each pass through Partof. This is represented by the following code:

```mathematica
Partof[lrules_, clist_, cword_, s_, sep_] :=
  Block[{i, j, k, v, pomlist, pomlist1, word, ls1, start1, aux1, sl1, hlist = {}, firstsimbol3},
    word = cword;
    s1 = s;
    pomlist = lrules;
    pomlist1 = clist;
    separator = sep;
    For[i = 1, i \leq Length[pomlist], i++,
      ls1 = List[pomlist[[i]]];
      start1 = ToString[(ls1[[1]] /. Rule -> List)[[2]]];
      aux1 = Characters[start1];
      sl1 = Last[aux1];
      hlist = {};
      For[j = 1, j \leq Length[VP1], j++,
        firstsimbol3 = ToString[(VP1[[j]] /. Rule -> List)[[1]]];
        If[sl1 == firstsimbol3, AppendTo[hlist, VP1[[j]]]];]
    If[Not[MemberQ[pomlist1, pomlist[[i]]]],
      AppendTo[pomlist1, pomlist[[i]]];
      If[MemberQ[VN1, ToExpression[sl1]],
        word = word <> StringDrop[start1, -1],
        word = word <> start1];
      Partof[hlist, pomlist1, word, 1, separator];
      word = StringDrop[word, -1];
      pomlist1 = Drop[pomlist1, -1],
    ];
    For[k = 1, k \leq Length[pomlist1], k++,
      If[pomlist1[[k]] = pomlist[[i]], v = k;]
      AppendTo[separator, {v, StringLength[word]}]];]
    If[s1 == 1, RegLang[separator, word];
      separator = {};
    s1 = s1 + 1;
  ]
```

"General :: spell": Possible spelling error: new symbol name "word" is similar to existing symbols (cword, Word). More...

"General :: spell1": Possible spelling error: new symbol name "hlist" is similar to existing symbol "clist". More...
Finally, the function \texttt{RegLang} inserts separators into the string attained in parameter word to appropriate places and define printing of the form \( L(\text{VP}) = expression \cup expression \cup \ldots \cup expression \cup \emptyset \).

\begin{verbatim}
In[3]:= RegLang[p_, rec_] :=
  q = p;
  expre = {};
  exprel = {};
  expre2 = {};
  ca = {};
  If[Length[p] ≥ 1,
    For[i = 1, i ≤ Length[p], i++, f = {};
      For[j = 1, j ≤ Length[q],
        j++, If[(p[[i]][[2]] ≥ q[[j]][[2]]) \&\&
          (p[[i]][[1]] ≤ q[[j]][[1]]) \|\|
          (p[[i]][[2]] ≤ q[[j]][[2]]) \&\&
          (p[[i]][[1]] > q[[j]][[1]]), f = AppendTo[f, j]];
        ce = Part[p, f];
        ca = AppendTo[ca, ce];
        ca = Union[ca];
        For[i = 1, i ≤ Length[ca],
          i++, expre = AppendTo[expre, rec]];
        For[j = 1, j ≤ Length[ca], j++, pom1 = {};
          pom2 = {};
          pom3 = expre[[j]];]
        For[k = 1, k ≤ Length[ca[[j]]],
          k++, pom1 = AppendTo[pom1, ca[[j]] [[k]] [[1]]]];
        AppendTo[expre, pom1];
        For[k = 1, k ≤ Length[ca[[j]]], k++, te = 0;
          For[el = 1, el ≤ Length[pom1], el++,
            If[pom1[[el]] ≤ ca[[j]] [[k]] [[2]], te = te + 1];
          pom2 = AppendTo[pom2, ca[[j]] [[k]] [[2]] + te + 1];
          AppendTo[expre, pom2]];]
        For[ka = 1, ka ≤ Length[expre],
          ka++, exprel1[[ka]] = Sort[expre[[ka]]];
          exprel2[[ka]] = Sort[expre2[[ka]]];]
        For[j = 1, j ≤ Length[expre1], j++, se = exprel1[[j]];
          zag = StringInsert[rec, "", se];
          pom2 = expre2[[j]];
          ind = True;
          For[ka = 1, ka ≤ Length[pom2], ka++,
            If[pom2[[ka]] > StringLength[zag] + 1, ind = False];
            If[ind, zag = StringInsert[zag, "", pom2];
               language = language \< zag \< " |"
               ]];
          , language = language \< zag \< " |"
        ]
  ]

The algorithm executes by call of the function \texttt{Regular} with actual values for the parameters \texttt{VN}, \texttt{VT}, \texttt{AKS} and \texttt{VP}.

\textbf{Example 2.1.}

\begin{verbatim}
In[4]:= Regular[{A, B, C, D}, {a, b, c, e}, B, 
{A \to aB, A \to bB, B \to bD, B \to cA, B \to e, D \to eB}]
\end{verbatim}

\begin{verbatim}
Grammar is regular
\end{verbatim}

\begin{verbatim}
startrules = {B \to bD, B \to cA, B \to e}
\end{verbatim}

\begin{verbatim}
Out[4]= L(\text{VP}) = {be} \cup (be) \cup \emptyset
\end{verbatim}
We also solve the following problem. Given an arbitrary word; decide whether the word can be generated by the regular grammar. If it is possible, generate the list of productions which generate the word. In the following procedure we use an additional parameter, Chain, which represents the observed word.

In order to detect the derivation rules which generate given chain, we use bottom-up parsing which is defined by the following rules.

**Rule 1.** If the last symbol in given chain is a terminal character \( a \in VT \), and there exists a production rule \( A \rightarrow a, \ A \in VN \), replace the symbol \( a \) by \( A \).

**Rule 2.** Otherwise, check the existence of the production rule \( a \rightarrow bB, \ a \in VN, \ b \in VT, B \in VN \) and \( bB \) are the last two symbols in the chain. If it is the case, replace \( bB \) in the chain by the nonterminal character \( a \).

If both of these rules are not applicable, then the chain can not be generated by given regular grammar. Otherwise, we continue these rules until the starting chain is transformed to a nonterminal \( A \in VN \).

When the nonterminal \( A \) is equal to the axiom, then the given chain can be generated by the grammar. Otherwise, the chain again is not be generated by given regular grammar.

During the application of **Rule 1** and **Rule 2** we remember order numbers of applied transformation rules in the list deriv. Also, all transformations which produce the given chain are remembered by means of variable transform.
Example 2.2. In this example we illustrate work of the function Regular1. Its actual parameters are four values representing a regular grammar \((V_N, V_T, AKS, VP)\) and a string from \(V_T^*\). In the case when the grammar is regular, the function detects whether the string can be generated by the grammar. In the case when both answers are positive, function gives production rules from \(VP\) which derive the string starting from \(AKS\).
IN[6]:=

\[
\text{Regular1[\{A, B, C, D\}, \{a, b, c, e\}, B,}
\text{ \{A \rightarrow aB, A \rightarrow bB, B \rightarrow bD, B \rightarrow cA, B \rightarrow e, D \rightarrow eB\}, cbbebee]}\]

\[
\text{grammar is regular}
\]

\[
\text{Chain cbbebee CAN be generated by the grammar}
\]

\[
\text{and has the following derivation:}
\]

\[
\text{deriv = \{\{4, 2, 3, 6, 3, 6, 5\}\} brl = 1}
\]

\[
\text{Out[6]= B (rule B \rightarrow cA) \rightarrow cA (rule A \rightarrow bB) \rightarrow cbB (rule B \rightarrow}
\text{ bD) \rightarrow cbbD (rule D \rightarrow eB) \rightarrow cbbeB (rule B \rightarrow bD) \rightarrow}
\text{cbbebD (rule D \rightarrow eB) \rightarrow cbbebeB (rule B \rightarrow e) \rightarrow cbbebee}
\]

References


