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.
Softvare 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 whish 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 aksiom 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 $y = y_1 y_2 ... y_n$, where $x_1, x_2, ..., x_n, y_1, y_2, ..., y_n \in X$, is the string $xy = x_1 x_2 ... x_n y_1 y_2 ... y_n$. This operation is associative, what means that X^+ with this operation makes a semigroup. This semigroup we call **free semigroup** over alphabet X. It is convevient 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 ($e \notin X^+$) 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 $L_1 L_2 = \{xy | x \in L_1 \; y \in L_2\}$.

Definition 2. Formal grammar, or shortly **grammar,** is the ordered quadriple $G=(V_T, V_N, S, P)$, 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 $\alpha \in (V_T \cup V_N)^* V_N (\Sigma \cup V)^*$ and $\beta \in (V_T \cup V_N)^*$, i.e. α is a string of terminals and nonterminals containing at least one nonterminal and β 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

$$L^* = \bigcup_{i \geq 1} L^i .$$

Hierarchy 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 **contex−senzitive** grammar if each production $\alpha\to\beta$ in P satisfies $|\alpha|\le|\beta|$.

3. G is of a **Type−2** grammar or **contex−free** if each production $\alpha\to\beta$ in P satisfies $|\alpha|=1$, which means that α 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:

$$A\to aB,\quad A\to a,\quad A\to\varepsilon,\quad A\in V_N,\ a\in V_T.$$

Many courses allow right−linear grammars to have productions of the form $A\to pB$, where $p\in V_T^{+}$ or $\alpha\to q$, where $\alpha\in V_N$ 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 $V_T \bigcup V_N$. For the sentential form $w'\in(V_T \bigcup V_N)^{*}$ we say to be **directly derived** from the sentential form $w\in(V_T \bigcup V_N)^{*}$ and write $w\Rightarrow w'$, if $w=puq$ and $w'=pvq$ and there is a rule $u\to v$ from P. Next, we say that the sentential form $w'\in(V_T \bigcup V_N)^{*}$ is **derivable** from the sentential form $w\in(V_T \bigcup V_N)^{*}$, and write $w\overset{*}{\Rightarrow}w'$, if $w=w'$ *is valid* or there is an array $w_1, w_2, ..., w_n \in(V_T \bigcup V_N)^{*}$, $n\ge 2$, such that: $w=w_1\Rightarrow w_2\Rightarrow,...,\Rightarrow w_n=w'$. In this case, sequence $w_1, w_2, ..., w_n$ is called the **derivation** of the sentential form for w' from w.

For the nonterminal symbol $S\in V_N$, the set $L(G, \delta) = \{w\in X^{*} \mid S \overset{*}{\Rightarrow} w\}$ 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 needful 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:*

$$A\to xB,\ \text{where}\ A,B\in V_N\ \text{and}\ x\in V_T,$$
$$A\to\varepsilon,\ \text{where}\ A\in V_N.$$

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 ε is a regular expression, and represents the language whose only member is the empty string, i.e. $\{\varepsilon\}$.

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 languges R and S, respectively, then $(r+s)$, (rs) and (r^{*}) are regular expressions that represent languages $R\bigcup S$, 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 otherise, and return it. The following code represents the first function:

```
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++,
                  rul=VP[[i]]/.Rule->List;
                  aux=Characters[ToString[rul[[2]]]];
                   k=Length[aux];
                     For[j=1,j<= k,j++,
                           aux[[j]]=ToExpression[aux[[j]]]
                        ];
                     PrependTo[aux, rul[[1]]];
                     p= p && MemberQ[VN,First[aux]];
                     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. $VN \cap VT = \phi$;
2. first element in any rule $\alpha \to \beta$ is an element from *VN*;
3. $\alpha(i) \in VT$, $i=1, \ldots, k-1$, $\alpha(k) \in VN \cup VT$, where $|\alpha|=k$.

For an automatic characterization of regular grammars we use the following :
In the case *p*=False the grammar (*VN*, *VT*, *AKS*, *VP*) is not regular; otherwise, in the case *p*=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*=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 $\alpha \to \beta$, where $\alpha = AKS$, are incorporated into the list

startrules, which can be generated applying the following code:

```
 For[j=1,j<=Length[VP],j++,
firstsymbol=ToString[(VP[[j]]/.Rule->List)[[1]]];
If[firstsymbol==ToString[AKS],AppendTo[startrules,VP[[j]]]
]
];
```

All terminal expessions can be generated inside the cycle

```
For[w=1,w<=Length[startrules], w++, ...
```

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

```
In[1]:= Regular[VN_, VT_, AKS_, VP_] :=
 Block[{p = True, d, rul, start, aux = {},
    auxlist, ls = {}, language = "", firstsymbol,
    firstsymbol2, separator1 = {}, seplist = {},
    setofexp = {}, change, i, j, h, n, w, k, word1 = "",
    listrules = {}, checklist = {}, startrules = {}, l = 1},
   IfRegular[VN, VT, VP, True];
    If[p,
       Print["Grammar is regular"];
        (* Then *)
       For[j = 1, j <= Length[VP], j++,
           firstsymbol =
     ToString[(VP[[j]] /. Rule -> List)[[1]]];
                                              (*  Print[
       "firstsymbol = ",firstsymbol];*)
           If[firstsymbol == ToString[AKS],
     AppendTo[startrules, VP[[j]]]
          ] ];
                                                    Print[
     "startrules = ", startrules];
         For[w = 1, w <= Length[startrules], w++,
           checklist = {};
           separator1 = {};
           checklist =
     AppendTo[checklist, startrules[[w]]];
             ls = List[startrules[[w]]];
             start =
     ToString[(ls[[1]] /. Rule -> List)[[2]]];
             (* Print[start];*)
             aux = Characters[start];
             sl = Last[aux];
           word1 = "";
           If[MemberQ[VN, ToExpression[sl]], word1 =
      word1 <> StringDrop[start, -1],  word1 = word1 <> start];
           listrules = {};
           For[k = 1, k <= Length[VP], k++,
             firstsymbol2 =
     ToString[(VP[[k]] /. Rule -> List)[[1]]];
             If[sl == firstsymbol2,
     AppendTo[listrules, VP[[k]]]];
                          ];
           VP1 = Sort[VP];
           VN1 = VN;
           VT1 = VT;
           Partof[
     listrules, checklist, word1, 1, separator1]

         ];
            language = "L(VP) = " <> language <> " ∅",
   Return["Grammar is not regular"] (* else *)
  ]
 ]
```

```
General::spell1 :
 Possible spelling error: new symbol name "language" is
   similar to existing symbol "Language". More...
```

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:

```
In[2]:= 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 ≤ 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 ≤ 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 ≤ 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 RegLang inserts separators into the string attained in parameter word to appropriate places and define printing of the form $L(VP) = expression \bigcup expression \bigcup ... expression \bigcup \emptyset$.

```
In[3]:= RegLang[p_, rec_] := (
          q = p;
          expre = {};
          expre1 = {};
          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[expre1, 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[expre2, pom2]];
           For[ka = 1, ka ≤ Length[expre1],
           ka++, expre1[[ka]] = Sort[expre1[[ka]]];
           expre2[[ka]] = Sort[expre2[[ka]]]];
           For[j = 1, j ≤ Length[expre1], j++, se = expre1[[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 Regular with actual values for the parameters *VN*, *VT*, *AKS* and *VP*.

Example 2.1.

```
In[4]:= Regular[{A, B, C, D}, {a, b, c, e}, B,
         {A -> aB, A → bB, B → bD, B → cA, B → e, D → eB}]
```

```
Grammar is regular
```

```
startrules = {B → bD, B → cA, B → e}
```

```
Out[4]= L(VP) = (be)^+cae ⋃(be(ca)^+)^+e ⋃((be)^+
          (ca)^+)^+e ⋃(be(cb)^+)^+e ⋃(be(cb)^+)^+e ⋃(ca(be)^+)^+
          e ⋃(ca)^+e ⋃(cb(be)^+)^+e ⋃(cb)^+e ⋃(cb)^+e ⋃ ∅
```

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 $A \to a$, $A \in VN$, replace the symbol a by A.

Rule 2. Otherwise, check the existence of the production rule $\alpha \to bB$, where $\alpha \in VN$, $b \in VT$, $B \in VN$ and bB are the last two symbols in the chain. If it is the case, replace bB in the chain by the nonterminal character α.

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.

```
In[5]:= Regular1[VN_, VT_, AKS_, VP_, Chain_] :=
 Block[{p = True, d, d2, drv, aux = {},
    i, j, k, brl, sim = "", rulelist = {}, auxchain,
    transform = {}, auxtransform = {}, end = False, deriv = {},
    rulderiv = {}, auxderiv = {}, auxrulderiv = {}},
   d = Length[VP];
       For[i = 1, i <= d, i++,
        drv = VP[[i]];
        drv = drv /. Rule -> List;
        rulelist = AppendTo[rulelist, drv]];
  IfRegular[VN, VT, VP, True];
 If[p,
   Print["grammar is regular"];
      auxchain = ToString[Chain];
      d = Length[rulelist];
         For[i = 1, i <= d, i++,
     drv = rulelist[[i]];
        If[ToString[drv[[2]]] == StringTake[auxchain, -1],
          AppendTo[transform,
         StringDrop[auxchain, -1] <> ToString[
          drv[[1]]]];
           AppendTo[deriv, List[
         i]];
          AppendTo[rulderiv, List[Last[transform]]];
          If[Last[transform] == ToString[AKS],
                     end = True; brl = Length[transform],
                     If[StringLength[Last[transform]] == 1,
                           transform = Drop[transform, -1];
                                 deriv = Drop[deriv, -1];
         rulderiv = Drop[rulderiv, -1]
          ] ] ]  ];

          While[(! end) && (transform != {}),
             d2 = Length[transform];
             For[i = 1, i <= d2, i++,
```

```
            For[j = 1, j <= d, j ++,
             drv =
       rulelist[[j]];
              If[ToString[drv[[2]]] ==
        StringTake[transform[[i]], -2],
                 AppendTo [auxtransform,
        StringDrop[transform[[i]], -2] <> ToString[drv[[1]]]];
                 AppendTo[auxderiv, Prepend[deriv[[i]], j]];
                 AppendTo[auxrulderiv, Prepend[rulderiv[[i]],
         Last[auxtransform]]];
                 If[Last[auxtransform] == ToString[AKS],
                      end = True; brl = Length[auxtransform],
                       If[
        StringLength[Last[auxtransform]] == 1,
                          auxtransform =
         Drop[auxtransform, -1];
                          auxderiv = Drop[auxderiv, -1];
                          auxrulderiv = Drop[auxrulderiv, -1]
          ]  ] ] ] ];

     transform = auxtransform;
    deriv = auxderiv;                  rulderiv = auxrulderiv;
        auxtransform = {};
    auxderiv = {};     auxrulderiv = {}];
     If[end,
            Print["Chain ",
     auxchain, " CAN be genereted by the grammar"];
            Print["and has the following derivation:"];
                                  Print[
     "deriv = ", deriv, " brl = ", brl];
               d = Length[deriv[[brl]]];
                For[i = 1, i <= d, i++,
                sim = sim <> (rulderiv[[brl]])[[i]];
                sim = sim <> " (rule ";
                sim =
      sim <> ToString[VP[[(deriv[[brl]])[[i]]]]] <> ") => "
             ];
       sim = sim <> auxchain,
                Return[
     "Chain  CANNOT be genereted by the grammar"]
        ],
   Return[ "grammar is not regular"]
  ]
 ]
```

```
General::spell1 : Possible spelling error: new symbol
   name "end" is similar to existing symbol "End". More...
```

Example 2.2. In this example we illustrate work of the function Regular1. Its actual parameters are four values representing a regular grammar (*VN*, *VT*, *AKS*, *VP*) and a string from VT^* . 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]:= Regular1[{A, B, C, D}, {a, b, c, e}, B,
  {A -> aB, A → bB, B → bD, B → cA, B → e, D → eB}, cbbebee]
```

```
grammar is regular
```

```
Chain cbbebee CAN be genereted by the grammar
```

```
and has the following derivation:
```

```
deriv = {{4, 2, 3, 6, 3, 6, 5}} brl = 1
```

```
Out[6]= B (rule B -> cA) => cA (rule A -> bB) => cbB (rule B ->
    bD) => cbbD (rule D -> eB) => cbbeB (rule B -> bD) =>
    cbbebD (rule D -> eB) => cbbebeB (rule B -> e) => cbbebee
```

References

1. N.Blachman. *Mathematica: A Practical Approach*, Englewood Cliffs, New Jersey:Prentice–Hall, 1992.

2. J.E. Hopcroft andJ.D. Ullman, *Introduction to Automata Theory, Languages, and Computation, second edition.* Addison–Wesley, 2001.

3. P.S. Stanimirovic,G.V.Milovanovic, *Programming package Mathematica and applications, monographs edition*, Electronic Faculty of Nis, 2002, in Serbian.

4. R. Maeder, *Programming in Mathematica,Third edition,* Addison–Wesley, Redwood City, California,1996.

5. S.Wolfram, *TheMathematica, 4th ed*, Wolfram Media/Cambridge University Press, 1999.