DE GRUYTER OPEN

BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **14**, No 4

Sofia • 2014 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.1515/cait-2014-0003

Equivalent Transformations and Regularization in Context-Free Grammars

Ludmila Fedorchenko[1], *Sergey Baranov*[2]

[1] St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences, 14 liniya, 39, V.O. St. Petersburg, 199178, Russia
[2]ITMO University, Av. Kronverkski, 49, St. Petersburg, Russia
Emails: LNF@iias.spb.su SNBaranov@gmail.com
http://www.spiiras.nw.ru/index.php?newlang=english

Abstract: *Regularization of translational context-free grammar via equivalent transformations is a mandatory step in developing a reliable processor of a formal language defined by this grammar. In the 1970-ies, the multi-component oriented graphs with basic equivalent transformations were proposed to represent a formal grammar of ALGOL-68 in a compiler for IBM/360 compatibles. This paper describes a method of grammar regularization with the help of an algorithm of eliminating the left/right-hand side recursion of nonterminals which ultimately converts a context-free grammar into a regular one. The algorithm is based on special equivalent transformations of the grammar syntactic graph: elimination of recursions and insertion of iterations. When implemented in the system SynGT, it has demonstrated over 25% reduction of the memory size required to store the respective intermediate control tables, compared to the algorithm used in Flex/Bison parsers.*

Keywords: *CFR-grammar, a syntactic flow-chart* (*graph-scheme*), *equivalent transformations of grammars.*

1. Introduction

The automated analysis of formal languages which started in 1960-ies, is a natural part of computational linguistics. Various tools were developed for processing context free and context dependent formal languages, which need to be formalized so that their properties to be studied by computer machinery.

Nowadays data processing technologies are widely used in translation systems for a variety of computer devices with many applications supporting them. For example Flex/Bison parsers [7, 13] are often used to quickly obtain a particular

language processor from a formal definition of a language. However, these parsers have certain limitations on the type of the input formal grammar, and the employed algorithm often requires enormous memory for storing intermediate data.

A significant problem was that of the prompt adjustment (transformation) of the syntactic definition of a particular language to a particular form to allow for automated or manual development of its processor; another problem concerns the limitations of the selected method of syntactic analysis.

The former problem is aggravated by the great diversity of tools employed for formal language definition and representation, from conventional Backus-Naur Forms (BNF) to two-level affix (or other types) grammars, to various versions of the HyperText Markup Language (HTML). The latter often leads to linguistic ambiguity (or non-determinism) of the processing automaton. Solutions to these two problems are closely connected to the formal presentation of the given formal language and the processing environment.

The system SynGT (Syntax Graph Transformations), developed at SPIIRAS provides a solution to the above mentioned problems. The diagrams (graphs) of the rules are used to represent a context-free grammar (CF-grammar) of a formal language. The graph transformations are applied to perform equivalent grammar transformations of the input grammar to obtain its context-free regular form.

The paper is structured as follows.

Section 2 provides an overview of the existing tools for automated grammar transformations based on authors' experience and literature search, including the Compiler Compilers News Group [8] in Internet.

In Section 3 we recall the definition of a CF-grammar in a regular form (CFR-grammar) and give examples of graphic representation of such grammars rules.

In Sections 5 and 6 the system SynGT is presented, as well as the main results concerning regularization with it. A new algorithm implemented in SynGT eliminates the recursion (left, central, or right) that often spoils CF-grammar rules and imposes unnecessary limitations on the language syntax. Examples of SynGT applications are discussed, where the maximal regularization of a grammar resulted in the maximal efficiency of the generated language processor.

2. Tools for automated grammar transformations

The existing approaches to the language development and commercial tools for automated construction of language processors, such as Lex-Yacc [9, 10], Antlr [11], SYNTAX [12], Forth Technology [6] and Flex/Bison [13] anticipate equivalent transformations of the source CF-grammar; some of them are executed automatically during the compilation of a new parser; others are expected to be performed manually at the user's discretion, which is error-prone and may significantly hamper the development process.

The regularization procedure in SynGT [2] performs equivalent transformations of the initial CF-grammar in order to obtain a form, suitable for further optimization of the respective language parser. The user interface with an adequate set of transformation functions [4] is designed for quick finding and

resolving linguistic ambiguities (elimination of non-determinism) in the given set of grammar rules in the form of graphs. Theoretical research performed in 1980-ies [1] laid a solid foundation for automated generation of a syntactic parser from an input grammar with respective constraints. Later on, this mathematical model and the respective grammar regularization algorithm were refined to optimize the parser (see [2, 4, 5]) in SynGT.

The variety and diversity of the syntactic definitions of languages requires the adjusting of their grammars to a particular translational automaton of SynGT, which means that equivalent transformations of these grammars must be performed (automatically or manually) prior to generation of its syntactic parser.

The significant requirements to the technologies of language processor development (and, in particular, to parsers) are in efficient use of the development resources (namely, the time and effort required to develop such software product), the minimal size of the parser in case it is incorporated into microprocessors, and the overall low cost of the development. It is desirable to minimize the time required to compile a new grammar adapted to the selected method of analysis. This is achieved through creation of specialized tools for developers.

A characteristic feature of such systems is that the information about conflicts is so highly redundant that it is difficult for the user to interpret it. For example, the Bisons system can sometimes generate a few pages of warnings for possible errors in a grammar, most of which are superimposed as a result of an automated correction of one primary error.

Other typical features of these systems are their price and limitations on the types of the initial grammar, even though modern systems have already expanded the initial grammar class up to LALR (Look Ahead Left to Right) grammars, which is the basis for almost all deterministic languages. According to the information from the Compiler Compilers Newsgroup [9], no sound techniques exist to resume the analysis of the grammar rules after a conflict situation (error) was detected. Equivalent transformation of the grammar is performed manually and is often incomplete. A parser generated from the resulting grammar does not have the minimal number of the parsing automaton states.

The Kleene theorem (see [1]) states that the class of regular sets is the minimal class containing all final-state sets closed w.r.t. the operations of union, concatenation, and closure; therefore, regular sets (sets, recognizable by final-state machines) may be represented by regular expressions.

This mathematical model can only be used for regular languages; the whole set of regular expressions over a fixed alphabet can be regarded as a regular language described by a context-free grammar (meta-grammar) with a single rule, and then this language is recognized by a final-state machine with a minimal number of states; thus, an optimal verifier of regular expressions can be created.

In SynGT such a finite-state machine is implemented for verification of the correctness of regular expressions representing the rules of translational grammar. It has displayed 25% reduction of memory consumption compared to the algorithm used in the Flex/Bison parsers on the same examples.

3. Definitions

In this section we give the classic definitions and notations used in the theory of formal languages and abstract automata and then modify some of them. We recall a definition of the CF-grammar in a Regular form (CFR-grammar) and give examples of graphic representation of the rules of such a grammar.

The language syntax is usually defined via the notation, called Backus-Naur form. In the SynGT system regular expressions with binary operations are used. They are much easier to transform into a reversed Polish notation for subsequent analysis and processing. From the viewpoint of the generated language, these forms are equivalent.

Definition 1. An alphabet $V=\{\xi_1, \xi_2,...,\xi_n\}$ is a finite set of letters ξ_i. The concept of a letter is not defined. A word in the alphabet V is either a letter of V, or a concatenation $x\xi$ where x is a word in the alphabet V and $\xi \in V$. If x is a word in V, then $|x|$ denotes its length – the number of occurrences of letters in x; a word of the length zero which contains no letters is called an empty word and is sometimes denoted as ε; V^+ is the set composed by all words in V, and $V^* = V^+ \cup \{\varepsilon\}$. A language L_V over some alphabet V is an arbitrary subset of $V^*: L_V \subset V^*$.

For an infinite language there is a problem of its finite representation which in its turn, is a representation itself by words over an alphabet with implied interpretation, which relates it to concrete representation of the given language. Usually such an ”upper-level” representation language is called a meta-language, which, like any other language, is nothing more than a countable set of words. Further discussions are devoted to the meta-language of regular expressions which is a basis for the class of generative systems – CFR-grammars.

Definition 2. A union of two languages A_V and B_V denoted as $A_V; B_V$ is a set-theoretic union of sets A_V and B_V of words over V. A product of two languages A_V and B_V denoted as A_V, B_V is a set of words in V which consists of all possible words of the type xy where $x \in A_V$ and $y \in B_V$. A generalized iteration of two languages A_V and B_V denoted as $A_V \# B_V$ (occasionally called iteration with a separator, or iteration according to G. S. Tseytin) is a set which consists of all possible words of the type $x_1, y_1 x_2, y_2,..., y_{n-1}, x_n$ where $x_i \in A_V$, $1 \le i \le n$, $y_j \in B_V$, $1 \le j \le n-1$, and n is a positive integer.

Two particular cases when an operand of the generalized iteration is empty, are called a *right-hand side iteration* and a *left-hand side iteration* respectively: $A\# = A\#\{\varepsilon\} = \{A; A, A; A, A, A; \ldots\}$ and $\# A = \{\varepsilon\}\# A = \{\varepsilon; A; A, A; \ldots\}$; the operation # is treated as a unary operation in this case.

The generalized iteration # can be expressed through the classical Kleene operation *: $A\#B = A, (B, A)*$.

Definition 3. The regular languages in V (subsets of V*) are defined

inductively:

- the sets $\{\xi_1\}, \{\xi_2\}, \ldots \{\xi_n\}$ and $\{\varepsilon\}$ where $\xi_i \in V$ are regular;
- if A_V and B_V are regular, and $\otimes$ is one of the operations: union (;), product (,), or generalized iteration (#), then the set $C_V = A_V \otimes B_V$ is regular.

The set of all Regular Languages (RL) in the alphabet *V*, denoted by $\mathrm{RL}(V)$ is the minimal subset of the set 2^{V^*} that contains $\{\xi_1\}, \{\xi_2\}, \ldots, \{\xi_n\}, \{\varepsilon\}$, and is closed under operations in $\otimes$.

In accordance with the given definitions, a set of words from the alphabet V is regular if and only if it equals to either $e = \{\varepsilon\}$ (ε is an empty word), or $\{\xi_i\}$ for some $\xi_i \in V$, or if it can be constructed from them by a finite number of operations from $\otimes$.

S. C. Kleene (see [1]) has introduced the notion of a regular expression in an alphabet V. Kleene iterations A^* and A^+ may thus be identified with $\# A = \{\varepsilon\} \# A$, and $A\# = A\#\{\varepsilon\}$ respectively.

We introduce the notion of a generalized regular expression. When unambiguity allows, the brackets around the regular subexpressions may be omitted. We also introduce square brackets as a shorthand: $[A] = (A; \{\varepsilon\})$; i.e., for "possible *A*".

Definition 4. A generalized regular expression representing a regular language L_V over the alphavet V and denoted as $r(L_V)$ is defined inductively: $r(\{\xi\}) = \xi$ where $\xi \in V \cup \{\varepsilon\}$, $r(A \otimes B) = (r(A) \otimes r(B))$ for any $A, B \in \mathrm{RL}(V)$.

The set of all regular expressions in V is denoted by $\Re(V)$. For each regular expression A let $L(A)$ be the corresponding regular language.

Each regular expression represents a regular language, and for each regular language a regular expression exists which represents this language (this can be easily proved through induction on the construction process). However, since a regular language can be constructed with operations from $\otimes$ in many different ways, in general, different regular expressions may represent the same regular language.

Here the well known problem stems – how to find a minimal regular expression representing the given language.

Definition 5. Two regular expressions $A, B \in \Re(V)$ are equivalent (denoted as $A \equiv B$), if they represent the same regular language in $V : L(A) = L(B)$.

It is well known that regular expressions representing the same regular language can be transformed into each other using certain identities. Some of them are provided in Table 1 below, where $A, B, C \in \Re(V)$.

Table 1. Identities for regular expressions

$(A; B) \equiv (B; A)$	$((A; B); C) \equiv (A; (B; C))$	$(C, (A; B)) \equiv ((C, A); (C, B))$
$(A, \{\varepsilon\}) \equiv (\{\varepsilon\}, A)$	$((A, B), C) \equiv (A, (B, C))$	$((A; B), C) \equiv ((A, C); (B, C))$
$(A; \{\varepsilon\}) \equiv [A]$	$(A\#B)\#C \equiv A\#(B; C)$	$A^* \equiv [A+]$

All operations in a generalized regular expression are binary, which is convenient for the program stack to be used when working with the reverse Polish form of regular expressions. Regular expressions can be used within a larger grammatical framework to represent regular CF-grammars (CFR-grammars).

It is convenient to add semantics to the grammar alphabet as operands of regular expressions. The aim is to allow some executable code be incorporated into the compiled text.

Definition 6. A contex-free grammar in a regular form (CFR-grammar) G_R is a quintuple of finite sets $G_R = (N, T, \Sigma, P, S)$, where N is a set of nonterminals, T is a set of terminals, Σ is a set of semantics (contexts), P is a finite set of CFR-rules for the nonterminals: $P \subset N \times \Re(N \cup T \cup \Sigma)$, and S is the starting (initial) nonterminal of the grammar.

A CFR-rule *for a nonterminal* $A \in N$ *(also called an A-rule for the nonterminal* A) is a pair (A, R) where $R \in \Re(N \cup T \cup \Sigma)$; it is represented as "*A*: *R*." with a period after *R*.

In CF-grammar terms, each pair (A, R) can be interpreted as a set of rules for $A: \{A \to x \mid x \in L(R),\ R \in \Re(N \cup T \cup \Sigma)\}$.

Definition 7. A nonterminal $A \in N$ is called recursive, if an inference of the type $A \overset{*}{\Rightarrow} \alpha A \beta$ exists, where $\alpha, \beta \in (N \cup T \cup \Sigma)^*$. In other words, if A may be derived from itself. In particular, a recursive nonterminal is called left/right-recursive when $\alpha / \beta = \varepsilon$.

An algorithm eliminating recursive nonterminals from a CF-grammar is described in Section 6.1 along with its generalization for a CFR-grammar.

4. Transformation of a CF-grammar into an equivalent CFR-grammar

Usually, the development of a language parser starts with transforming the initial CF-grammar into an equivalent well-formed (proper) CFR-grammar.

The process consists of the following steps:

- eliminating non-productive nonterminals;
- substituting regular expressions for non-recursive nonterminals;
- eliminating left/right-recursive nonteminals;
- identifying common prefixes for nonterminals;
- eliminating recursive alternatives for a nonterminal;
- eliminating the remaining recursive nonterminals;
- introducing a new nonterminal for common subexpressions;
- deleting superfluous nonterminals.

For a detailed description of the process see [5].

5. CFR-Rules for Transformations in SynGT

In SynGT the right-hand sides of rules are regular expressions with binary operations from $\otimes$ over terminals, semantics, and nonterminals. Without loss of

generality each nonterminal may be assumed to be defined by a single CFR-rule; i.e., this nonterminal occurs in the left-hand side of all rules only once. Indeed, two CFR-rules "$A:R_1.$" and "$A:R_2.$" can be replaced by just one: "$A:R_1;R_2.$". In the examples below the terminal symbols are enclosed in single quotes (like ',' or ';') and semantics symbols start with the character '$'.

The above definition of a CFR-grammar ensures that every rule of a classical CF-grammar is also a rule in a CFR-grammar. However, the reverse is not true. Usually, several CF-rules are needed to represent a regular set of words generated by a single CFR-rule. As an example see the rule for a "procedure call" in ADA.

A new feature distinguishing CFR-grammar from a classical CF-grammar is its mechanism of generating language words. In a conventional CF-grammar, this is a derivation tree, whereas in the CFR-grammar for each nonterminal A the set of words derivable by its A-rule is a value of the regular expression in the right-hand side of this A-rule, which thus generates an infinite set of words.

In SynGT this set is represented by a finite oriented labeled graph considered as a deterministic Finite State Machine (FSM), each node corresponding to a state of the FSM, and its label specifying the letter to be generated when the FSM transits into this state. A collection of such graphs representing the right-hand sides of CFR-rules of the given CFR-grammar *G* is called a *syntactic flow-chart* (*graph-scheme*) for *G*. Thus, an infinite set of words generated by the grammar *G* is represented by an infinite set of paths in such graph.

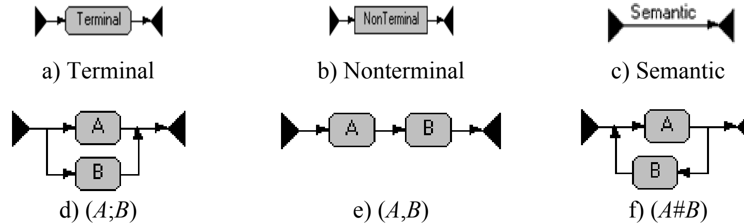
Syntactic flow-charts (as a mechanism of language generating) render the structure of CFR-rules more clearly for the user than regular expressions or various derivatives of a CF-grammar [2]. The conversion of a generating scheme into a recognizing one and then into a parsing scheme becomes more transparent; the invocations of semantics are easily associated with the flow-chart arcs labeled with the names of the respective procedures. A set of paths in a flow-chart is easier to understand than the source CFR-grammar *G*, in order to define the process of generation of words of the language *L*(*G*) and check its grammar properties.

Definition 8. A syntactic flow-chart *is a set of finite oriented graphs with labelled nodes and arcs. Each graph corresponds to some CFR-rule of the CFR-grammar and is called a graph for the nonterminal defined by this rule.*

More formally, with each CFR-grammar $G=(N, T, \Sigma, P, S)$ its counterpart is associated, which is a syntactic flow-chart $\Gamma_G=(N,T, \Sigma, C, S)$, such that the language $L(\Gamma_G)=L(G)$. Here $C=\{\Gamma_A\}$ is a set of component graphs Γ_A defining nonterminas $\{A\in N\}$. Each Γ_A defines a language construct (possibly with executable actions specified by its semantics interpreted as names of operations to be performed in this order while traversing the respective path in Γ_A).

The syntactic flow-chart for a CFR-grammar is recursively created from schemes for terminals/nonterminals and semantic – Fig. 1a-c – and the operations defined previously – Fig. 1d-f. The starting and finishing nodes have special notations. The internal nodes are labeled with terminals and nonterminals (operands of the regular expression of the right-hand side of the rule defining the nonterminal A), and the arcs are labeled with semantics – the names of procedures to be

executed when moving along the path.



a) Terminal b) Nonterminal c) Semantic

d) (*A*;*B*) e) (*A*,*B*) f) (*A*#*B*)

Fig. 1. Graphic representation of the basic elements of regular expressions and operations

Fig. 2 presents a flow-chart for the regular expression:

((('a'; 'b'), $S1, ' ', $S2, ('a'; 'b'), 'abc')) # (($S3,", $S4) # 'abc'). Note that this expression contains only terminals ('a', 'b', and 'abc') and semantics (named $S1, $S2, $S3, and $S4). In Fig. 2 $ sign is omitted because the arcs are labeled only by semantics words.

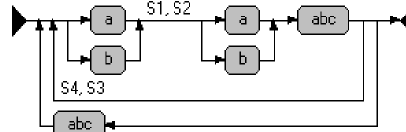


Fig. 2. Sample flow-chart representing a regular expressions with semantics

Fig. 3 represents the right-hand side of a CFR-rule for the non-terminal *procedure_call* defining the construct "procedure call" in Ada programming language.

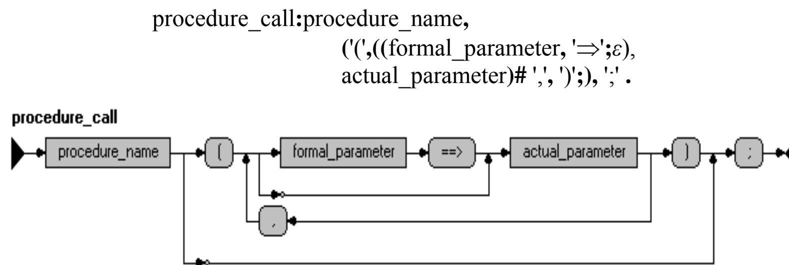


Fig. 3. A flow-chart corresponding to the rule for a non-terminal procedure_call

In order to generate language words, the relation of reachability in graph Γ_G is introduced for the nodes of the SFC (syntactic flow-chart); it reflects the letter order in the generated words.

A path P_x in the graph Γ_G which generates a word $x \in L_G$ contains a sequence of nodes marked by letters of the word x; its first and last elements are the initial and the last nodes of the graph for the starting nonterminal of the CFR-grammar, with each element being reachable from the previous one (if it exists), and the corresponding sequence of node marks composes the word x.

Each state of the respective FSM (language processor) is matched to a set of nodes in the syntactic flow-chart Γ_G with constraints to ensure its determinism.

These constraints are formulated in terms of the flow-chart properties and are taken into account during its construction. In fact, an FSM-state is either an output node of a flow-chart for a nonterminal, or a terminal node, or a pair of states (S_1, S_2) – the former, S_1, called the *transition* state and the latter, S_2 – called the *return* state, or a union of a number of states.

Occurrences of nonterminal nodes in Γ_G violate the regularity of the language $L(\Gamma_G)$ generated by Γ_G that is why a state cannot be always just a set of nodes (as in the regular case), but also a pair – the transition state and the return state. Each state may contain other states as well, thus forming a hierarchy of complex states.

The transition from one state into another, which is also a set of nodes (including an empty set) in SFC, is controlled by the current letter taken from the input word. Only certain letters are allowed in each state for a transition from this state to any next one; therefore, one can see which subword of the language being recognized resulted in transition into this state and it becomes possible to establish whether the input word belongs to the language generated by the given syntactic flow-chart Γ_G.

To illustrate the SynGT approach, let us consider building up a simple language recognizer from SFC. For this we need the following definition.

Definition 9. The state of any node β other than any finishing node in a flow-chart Γ_G denoted as S_β, is

$$S_\beta = \begin{cases} \alpha \mid \alpha \in \mathrm{succ}(\beta) & \text{if } \alpha \text{ is a terminal or finishing node in } \Gamma_G, \\ (\mathrm{S}_{E(m(\alpha))}, \mathrm{S}_\alpha) \mid \alpha \in \mathrm{succ}(\beta) & \text{if } \alpha \text{ is a nonterminal node in } \Gamma_G, \end{cases}$$

where succ(β) is a set of nodes – direct descendents of β, E_A – the entering (starting) node of the flow-chart for the nonterminal A and $m(\alpha)$ is the label which marks α.

Example 1. Let G be a CFR-grammar with three nonterminals: S, C_1, C_2; two terminals: ‘*a*’; ‘*b*’ and three grammar rules with graphs presented in Fig. 4:

G= ({S, C$_1$, C$_2$},{‘a’; ‘b’}, {(S : ‘a’, (C$_1$; C$_2$), ‘a’.), (C$_1$: ‘b’, S .), (C$_2$: ‘b’.)}, S).

Each node in the graphs is additionally marked with a number from 1 up to 7 in brackets to distinguish them.

The states of the entering (starting) and the finishing nodes of each flow-chart are denoted by $E(N)$ and $F(N)$ respectively, N being the name of the nonterminal of the respective flow-chart. Thus:

$S_{E(S)} = \{a(1)\}$; $S_{a(1)} = (b(5), S_{C_1(2)}), (\{b(7)\}, S_{C_2(3)})\}$; $S_{b(5)} = (\{a(1)\}, S_{S(6)})$;

$S_{a(4)} = \{F(S)\}$; $S_{S(6)} = \{F(C_1)\}$; $S_{b(7)} = \{F(C_2)\}$; $S_{C_1(2)} = S_{C_2(3)} = \{a(4)\}$.

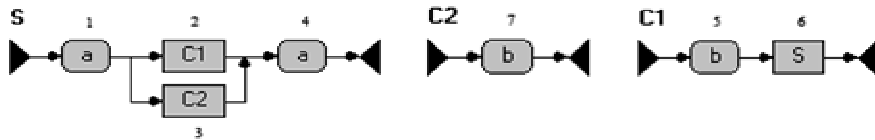


Fig. 4. Graphic representation of rules in Example 1

Definition 10. A transition state from the state S by letter ξ, denoted by S/ξ is a projection on the set of the nodes of S which are marked by ξ.

A transition state S/ξ in the flow-chart Γ_G may be defined recursively:

– if $S = \varnothing$ or $S = \{F(A_i)\}$, A_i being a nonterminal node in Γ_G, then $S/\xi = \varnothing$;

– if $S = \{\alpha\}$, α being a terminal node in Γ_G, then $S/\xi = \varnothing$ if $m(\alpha) \neq \xi$ or $S/\xi = S_\alpha$ if $m(\alpha) = \xi$, $m(\alpha)$ being the label of α;

– if $S = (S_1, S_2)$ then

$$S/\xi = \begin{cases} (S_1/\xi,\ S_2) & \text{if} \quad S_1/\xi \neq \varnothing; \\ S_2/\xi & \text{if} \quad S_1/\xi = \varnothing \ \text{and} \ F_A \in S_1 \ \text{for some} \ \Gamma_A; \\ \varnothing & \text{otherwise.} \end{cases}$$

The following transitions exist in Example 1 for the word 'abc':

$$S_0 = S_{E(S)} = \{a(1)\};$$

$$S_1 = S_0/a = S_{a(1)} = \{(b(5)\}, S_{C_1(2)}), (\{b(7)\}, S_{C_2(3)})\};$$

$$S_2 = S_1/b = \{(S_{b(5)},\ S_{C_1(2)}), (S_{b(7)},\ S_{C_2(3)})\} =$$

$$= \{((\{a(1)\},\ S_{S(6)}),\ S_{C_1(2)}), (F(C_2),\ S_{C_2(3)})\};$$

$$S_3 = S_2/a = \{((S_{a(1)}, S_{S(6)}), S_{C_1(2)}), S_{C_2(3)}/a\} =$$

$$= \{((S_{a(1)}, S_{S(6)}), S_{C_1(2)}), \{a(4)\}/a\} =$$

$$= \{((S_{a(1)}, S_{S(6)}), S_{C_1(2)}),\ S_{a(4)}\} =$$

$$= \{((\{b(5)\}, S_{C_1(2)}), (\{b(7)\}, S_{C_2(3)})\}, S_{S(6)}), S_{C_1(2)}), F(S)\}.$$

Various methods of defining the notion of a state in SFC and a transition state make possible the modeling of a push-gown translator.

References [1-5] describe the features of the syntactic flow-charts when the states of any nodes always exist (i.e., the process of generation of the transitional state is not cycled and is terminated), and formulate the determinacy conditions for the transition operation for the current symbol in the process of PDA-recognizer synthesis.

A description of the model [5] ends by a formulation of the theorem about the language that can be processed by the described recognizer; this theorem yields an algorithm, with which it is possible to check whether a word belongs to the language generated by the syntactic flow-chart, the number of algorithm steps being proportional to the length of the word.

6. Regularization algorithms in parser development

Regularization of the language grammar is a part of the language implementation cycle, which consists of a user's cycle and a semi-automatic developer's (implementer's) cycle. The development of a language (selecting the language

model, the parsing model and other implementation models) is made by the user, while the further language transformation process is carried out automatically.

The problem of equivalence of two context-free languages defined by different grammars is known to be generally unsolvable. However, quite often a language grammar can be equivalently converted into a form feasible for particular deterministic analysis, preserving the language defined by this new grammar form.

Let us consider some equivalent transformations of CFR-grammars used in language parsing and describe the algorithms for automated regularization of their grammars implemented in SynGT.

More formally, the regularization of a CF-grammar consists in applying a series of equivalent transformations starting with the source CF-grammar G and ending with a new CFR-grammar G' equivalent to G, but with no recursive nonterminals. If all nonterminals of the CF-grammar G are not recursive, they can be eliminated from the right-hand sides of the grammar rules. In that case CFR-rules for these nonterminals are deleted and the grammar G is transformed into a grammar G' consisting of a single rule, the right-hand side being a regular expression composed of terminals, semantics, and regular operations only. This regular expression defines the same regular language $L = L(G) = L(G')$. The algorithm will be described in Section 6.1.

During regularization of a grammar, the right-hand sides of its rules become regular expressions, and a syntax graph built up from the source grammar can be converted into a final state automaton.

Let us consider an algorithm for eliminating the left/right-recursive nonterminals in a CFR-grammar G, when the regular expression in the right-hand side of the rule for a recursive nonterminal contains a generalized iteration. This case was not considered in [1-3].

To make it simpler, let us consider a rule for a nonterminal A which is both left- and right-recursive and the recursion is direct (an indirect recursion is known to be reducible to a direct one through a series of substitutions). In SynGT a new algorithm of direct equivalent transformation of a left/right-recursive terminal is implemented which uses binary operations only.

6.1. Elimination of recursion in CFR-rules

Let us consider the following A-rule: "(A: (A, r_{11}, A; A, r_{12}; r_{21}, A; r_{22})).", where r_{11}, r_{12}, r_{21}, and r_{22} are regular expressions over the alphabet $N \cup T \cup \Sigma$. The right-hand side of the rule consists of four parts (A_i-fragments), $i = 1, 2, 3, 4$:

- A_1-fragment A, r_{11}, A contains both a left- and a right-hand side recursive occurrences of A;
- A_2-fragment A, r_{12} contains a left-hand side recursive occurrence of A;
- A_3-fragment r_{21}, A contains a right-hand side recursive occurrence of A; and
- A_4-fragment r_{22} contains no occurrences of A_2.

Let us consider what words of $L(A)$ can be generated by this A-rule in four steps.

Step 1. Let us consider the A_1-fragment of the A-rule: A, r_{11}, A. It generates words $A, Ar_{11}A; Ar_{11}Ar_{11}A, \ldots,$ also generated by the regular expression $A\#r_{11}$.

Step 2. Let us consider the A_2-fragment of the A_2-rule: A, r_{12}. It generates words $Ar_{12}, Ar_{12}r_{12}, Ar_{12}r_{12}r_{12}, \ldots,$ are also generated by the regular expression $A,(r_{12})^*$. Substituting this regular expression for A in the expression of Step 1, the expression $(A,(r_{12})^*)\#r_{11}$ is obtained.

Step 3. A similar substitution is performed for the right-hand side recursive occurrence of A in the A_3-fragment which generates words $r_{21}A, r_{21}r_{21}A, r_{21}r_{21}r_{21}A, \ldots,$ also generated by the regular expression $(r_{21})^*, A$. The substitution of this expression for A in the expression obtained at Step 2 results in the regular expression $((r_{21})^*, A, (r_{12})^*)\#r_{11}$.

Step 4. Substituting the A_4-fragment r_{22} for A in the expression obtained at Step 3 results in the final recursion-free regular expression $((r_{21})^*, r_{22}, (r_{12})^*)\#r_{11}$ for the nonterminal A.

After these preliminary transformations of regular expressions having been performed, the rule is reduced to a form in which the set $L(r_{12})$ does not contain any words of the type αA, the set $L(r_{21})$ does not contain any words of the type $A\alpha$, and the set $L(r_{22})$ does not contain any words of the type αA and $A\alpha$, α being a random word; thus there is neither left-hand side, nor right-hand side recursion for the nonterminal A.

Enumerating all nonterminals in the grammar G and applying the described transformation to each nonterminal one-by-one with immediate substituting the transformation results for all occurrences of the given nonterminal in the remaining rules, produces a grammar G' with no left-(right-)recursions, which is equivalent to the initial grammar G.

If r_{11}, r_{12}, r_{21}, and r_{22} are regular expressions with no occurrences of a nonterminal A, then A and the rule for it can be deleted from the grammar with simultaneous substituting the expression $((r_{21})^*, r_{22}, (r_{12})^*)\#r_{11}$ for all occurrences of A in the remaining rules.

6.2. Converting a CF-grammar into a non-recursive regular expression

Let $G = (N, T, P, S)$ be a proper CF-grammar without recursive nonterminals.

Definition 11. Nonterminal A depends on nonterminal B, if there is an A-rule of the form $(A:(\alpha, B, \beta))$ where $\alpha, \beta \in ((N \cup T)^*)$. The set D of all such pairs $(A, B) \in D$ is called the dependency relation among nonterminals of G. If $B = A$ then nonterminal A is called to be recursive.

Nonterminal A is called absolutely independent if $\neg\exists B:(A, B) \in D$. In other words, the independent nonterminals are determined by rules with terminals only in their right-hand sides.

The scheme of converting a CF-grammar into a regular expression is the following.

1. The set N of all nonterminals is split into non-intersecting subsets (levels) l_i, $0 \le i \le k <|N|$, according to the level of nonterminal dependency. Nonterminals, whose rules contain only terminals in their right-hand sides compose the lowest, zero level l_0; i.e., a regular set of words composed of terminals only is derivable from nonterminals of l_0 with just a single application of the respective grammar rule.

2. For every next level l_i and for all nonterminals at this level, a substitution of the values of nonterminals of the level l_{i-1} for all these nonterminals is performed. The last, highest level l_k contains only one element – the starting nonterminal S, which is replaced by the resulting regular expression at the final step of this conversion.

This is done through the hierarchy of dependency relation D for all nonterminals $A \in N$; i.e., if the right-hand side of an A_j-rule contains an occurrence of a nonterminal B_j, then the pair belongs to the dependency relation D by Definition 11. If D is represented in the form of a table, then pairs of terminals (A_j, B_j) identify its cells.

Now the task is to split all non-recursive nonterminals into disjoint subsets $s_0, s_1, \ldots, s_m$ where $m \le |N|$ with the following properties:

- all nonterminals $A \in s_0$ are absolutely independent;
- nonterminals of any set s_i, $1 \le i \le m,$ are independent from each other; i.e., for two nonterminals $(A, B) \in s_i$ $(A, B \notin D)$;
- nonterminals of any set s_i, $1 \le i \le m,$ are directly computable from the regular values of nonterminals from the previous sets s_j, $1 \le j \le i-1;$ in other words, if $A \in s_i$ then all occurrences of nonterminals in the right-hand side of its A-rule are replaced with regular values of these nonterminals from already calculated sets s_j, $1 \le j \le i-1$.

The set of nonterminals s_l is said to refer to the level l, $0 \le l \le m$. The highest level m consists of the starting nonterminal S only, if it does not occur in the right-hand sides of the rules.

Left/right-hand side recursions are eliminated as described in Section 6.1 when identified in rules being transformed.

This splitting method was described in [14]. Here an example is provided.

Example 2. Let *G* = (*N*, *T*, *P*, *S*) be a CFG, where:

T = {‘d’, ‘.’, ‘\’ , ‘e’, ‘+’ ; ‘–’};

$N = \{ A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10}, A_{11}, A_{12}, A_{13}, A_{14}, A_{15}\}$;

$S = A_{15}$;

P consists of 15 rules enlisted in Table 2. With the described algorithm we obtain the following dependency relation D among the nonterminals:

$$D=\{ (A_2, A_1),\ (A_3, A_2),\ (A_3, A_{14}),\ (A_5, A_3),\ (A_5, A_4),\ (A_6, A_{11}),\ (A_6, A_{14}),$$
$$(A_7, A_5),\ (A_7, A_6),\ (A_8, A_{14}),\ (A_9, A_{14}),\ (A_{10}, A_9),\ (A_{11}, A_8),\ (A_{11}, A_{10}),$$
$$(A_{13}, A_{12}),\ (A_{13}, A_{13}),\ (A_{14}, A_{13}),\ (A_{15}, A_7),\ (A_{15}, A_{11}),\ (A_{15}, A_{14})\}.$$

The element (A_{13}, A_{13}), being self-dependent, which means that the nonterminal A_{13} is left-recursive. The described equivalent transformation for this grammar results in the following grammar rule: $A_{13} : A_{12}, (A_{12})^* . \equiv (A_{12})\#$.

Table 2. Rules for Example 2

A_1: '+' ; '–'.	A_2: A_1; ε.	A_3: A_2, A_{14}.	A_4: '\' ; 'e'.
A_5: A_4, A_3.	A_6: A_{14}; A_{11}.	A_7: A_6, A_5.	A_8: '.' , A_{14}.
A_9: A_{14}.	A_{10}: A_9; ε.	A_{11}: A_{10}, A_8.	A_{12}: 'd'.
A_{13}: A_{12}; A_{13}, A_{12}.	A_{14}: A_{13}.	A_{15}: A_{14}; A_{11}; A_7.	

6.3. Sorting grammar nonterminals by independency levels

The final resulting set is obtained using the following

Sorting algorithm

Input: $G = (N, T, P, S)$ is a converted CF-grammar without left recursion; $D \subseteq N \times N$ is the dependency relation among nonterminals in G.

Output: $M = \{s_0, s_1, \ldots, s_m\}$, where for $\forall k, 0 \le k \le m$, $s_k \subset N$ is a subset of nonterminals of the given grammar of the level k.

Step 1. Identifying the initial level of all absolutely independent nonterminals:

$$l = 0;\ s_0 = \{A \mid \forall (A, B \in N) : \neg\exists(\alpha, \beta \in (N \cup T) : A \to \alpha B \beta \in P)\}.$$

Step 2. Constructing a set of nonterminals of the next level of independency:

$$l = l + 1;\ s_l = \{A \mid \forall (A \in N) : (\exists B \in N) : (B \in s_{l-1}) \wedge (A, B) \in R \wedge (A \ne B)\}.$$

Step 3. Eliminating nonterminals of the subset s_l from all subsets s_k, $0 < k \le l - 1$;

for $\forall (A \in s_l)$:

do for k **from** 0 **to** $l - 1$ **do if** $A \in s_k$ **then** $s_k = s_k \setminus \{A\}$ **od od**.

Step 4. Deciding whether to continue sorting of non-terminals:

if $s_l \ne \varnothing$ **then goto Step 2**.

Step 5. Terminating the process of sorting non-terminals:

$m = l - 1$; {The maximal level of non-terminals}

Applying this algorithm to Example 2, we receive the following:

$$s_0 = \{A_1, A_4, A_{12}\};\ s_1 = \{A_2, A_{13}\};\ s_2 = \{A_{14}\};\ s_3 = \{A_3, A_8, A_9\};$$
$$s_4 = \{A_5, A_{10}\};\ s_5 = \{A_{11}\};\ s_6 = \{A_6\};\ s_7 = \{A_7\};\ s_8 = \{A_{15}\}.$$

The dynamics of calculating the dependency levels is provided in Table 3. Here ’T’ denotes the identified nonterminal and ’ Ŧ ’ denotes its elimination from a lower level of the dependency relation.

Table 3. Dynamics of calculating the dependency levels

s_j	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}
8															T
7							T								Ŧ
6						T	Ŧ								Ŧ
5						Ŧ	Ŧ				T				Ŧ
4					T		Ŧ			T	Ŧ				
3			T		Ŧ	Ŧ		T	T						Ŧ
2			Ŧ				Ŧ							T	
1		T			Ŧ								T		
0	T			T								T			

Calculation of the regular values of nonterminals starts by grammar rules without nonterminals in their right-hand sides, such A-rules always exist because the given grammar is well-formed [1]. They all are at zero level s_0 of dependency.

There are three nonterminals of the zero level: A_1, A_4 and A_{12} in the considered Example 2. The relevant regular expressions for them are $R(A_1)$ = (’+’ ; ’–’), $R(A_{41})$ = (’\’ ; ’e’), and $R(A_{12})$ = (’d’). Raising to higher levels of dependency and substituting regular expressions for the respective nonterminals, the final regular value $R(A_{15})$ for the starting nonterminal $S = A_{15}$ is obtained:

$$R(S) = R(A_{15}) = ((\text{d}), (\text{d})^*) ;$$
$$(((((\text{d}), (\text{d})^*)) ; \varepsilon), ‘.’, ((\text{d}), (\text{d})^*)) ;$$
$$(((\text{d}), (\text{d})^*), ((((\text{d}), (\text{d})^*)) ; \varepsilon), ‘.’, ((\text{d}), (\text{d})^*)) ,$$
$$((‘\backslash’ ; ‘\text{e}’), (((((‘+’ ; ‘–’), \varepsilon), \varepsilon)), ((\text{d}), (\text{d})^*)))) \equiv$$
$$\equiv \text{d}^+ ; \text{d}^*, ‘.’, \text{d}^+ ; (\text{d}^+; \text{d}^*, ‘.’, \text{d}^+), (‘\backslash’ ; ‘\text{e}’), [‘+’ ; ‘–’], \text{d}^+ \equiv$$
$$\equiv (\text{d}^+; \text{d}^*, ‘.’, \text{d}^+), [(‘\backslash’ ; ‘\text{e}’), [‘+’ ; ‘–’], \text{d}^+].$$

7. Conclusion

A method of automated generation of language processors for a large class of grammars was proposed. Grammar regularization is a method for converting a CF-grammar into a new grammar in the regular form (CFR-grammar) through a series of equivalent transformations performed on the grammar syntactic flow-charts – a graphic analogue of grammar rules in a formal textual notation.

The proposed new algorithm of regularization demonstrated an increase in efficiency as compared to the methods used in parser compiling systems Flex/Bison and ANTLR [7]–[11].

Table 4. Comparative characteristics of compilers and tools for their development

№	Characteristics	SynGT (ver. 1.4)	Beta FORTH 95	LexYacc	Flex/Bison (ver. 2.85)	ANTLR (ver. 3.5.2)
1	Grammar type	CF-regular	LALR(1)	LALR(1)	LALR(2)	LL-regular
2	Platform	Wndows XP	Wndows XP	Wndows XP	Wndows XP	Wndows XP
3	Time (hours)	10	20	32	45	25
4	Space(bytes)	618 455	178 462	112 205	416 038	Archve 19.5 Mbytes

The algorithm can be used for developing software for small-size ad hoc language processors. Algorithms of equivalent transformations of grammars which were developed and implemented demonstrated a improvement, compared to other known systems of the compiler series GNU [7]–[10].

The algorithm and the method were realized in the system SynGT of equivalent transformations of CFR-grammars which allows for automated adjustment using language syntax within the framework of a simple syntactic analysis. The system can be used in other technologies for constructing language processors in a variety of domains, including computational linguistics.

This work was partially financially supported by the Government of the Russian Federation, Grant 074-U01.

References

1. A h o , A., R. S e t h i , J. U l l m a n. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986.
2. F e d o r c h e n k o , L. An Algorithm of Parsing Languages Generated with R-Grammars. – In: Algorithms and Systems for Research Automation. Moscow, Nauka, 2008, 15-20 (in Russian).
3. F e d o r c h e n k o, L. Syntax Graph Transformations in the System SynGT and Regularization of Grammars. – In: International Multi- Conference on Advanced Computer Systems (ACS-CISIM'2004), 14-16 June 2004, Elk, Poland.
http://acs.wi.ps.pl/info.php
4. F e d o r c h e n k o, L. On Regularization of Context-Free Grammars. – Izvestiya Vuzov. Priborostroyeniye, Vol. **49**, 2006, No 11, 50-54 (in Russian).
5. F e d o r c h e n k o, L. Regularization of Context-Free Grammars. LAP LAMBERT Academic Publishing, Saarbrucken, 2011.
6. B a r a n o v, S., C. L a v a r e n n e. Open C Compiler in Forth. – In: EuroForth'95, 27-29 October 1995, Schloss Dagstuhl.
7. Bison – GNU Parser Generator.
http://www.gnu.org/software/bison/
8. The comp.compilers newsgroup.
http://compilers.iecc.com/index.phtml/
9. **http://plan9.bell-labs.com/magic/man2html/1/lex/**
10. Lex – A Lexical Analyzer Generator.
http://dinosaur.compilertools.net/lex/
11. ANTLR (A Nother Tool for Language Recognition).
http://www.antlr.org/
12. Syntax: An Advanced Technology of Directed Syntax Processing (in Russian).
http://www.math.spbu.ru/user/mbk/SYNTAX/Syntax.html/
13. Win Flex-Bison.
http: / / sourceforge. net/ projects/ winflexbison/
14. M a r t y n e n k o , B . K . Regular Languages and CF Grammars. – Computer Tools in Education, Vol. **1**, 2012, 14-20 (in Russian).