ON THE MECHANIZATION OF KLEENE ALGEBRA IN FORMAL LANGUAGE ANALYZER

Dmitry Cheremisinov

Abstract: *Pattern matching is the technique of searching a text string based on a specific search pattern. The pattern specified by the regular expression forms the basis for building a variety of formal language texts converters. Kleene algebra or the algebra of regular events is an algebraic system that captures properties of several important structures arising in computer science like automata and formal languages, among others. Regular expressions are formulas of Kleene algebra. In this paper we present a formalization of regular expressions as Kleene algebra in the formal language analyzer. It is proposed to change the traditional model of the language parser by pattern matching based on the finite state machine into the algebra of patterns with side effects. The proposed deterministic semantic of regular expression eliminates the need to switch from the regular expression engine and user code execution environment and back again.*

Keywords: *Formal language parser, regular expression, Kleene algebra, finite automaton, deterministic semantic of regular expression.*

ACM Classification Keywords: *I. Computing Methodologies: I.1 SYMBOLIC AND ALGEBRAIC MANIPULATION: I.1.3 Languages and Systems, Special-purpose algebraic systems.*

Introduction

Regular expressions [Friedl, 2006] are often used in practice in order to build programs, which are text analyzers. It is quite common to generate useful and efficient parsers for programming languages from a formal grammar. It is also quite common for programmers to avoid such tools when making parsers for simple computer languages, such as file formats and communication protocols. Such languages are often regular and tools for processing the context-free languages are viewed as too heavyweight for the purpose of parsing regular languages. The extra run-time effort required for supporting the recursive nature of context-free languages is wasted.

Processor of regular expressions is a parser based on a deterministic finite automaton. This machine may be represented as a static data of the program in the form of the output/transitions table. The table-controlled analyzers are used in Perl, Python, Emacs, Tel and Net. The processor of regular

expressions does not "catch" the subexpression of the original regular expression, because an indication of recognition of a regular language sentence is transition to the final state.

Unlike the problem of recognizing language that traditionally is considered in the theory of formal languages, the task of language analyzer is to build the structure of the analyzed text, when it is parsed. A program that attempts to verify written text for grammatical correctness is a grammar checker. The recognition algorithm has the form of grammar. To construct a grammar analyzer it is necessary to add steps that form a data structure that represents the result of the analysis. Regular expressions describe regular languages. They have the same expressive power as regular grammars. Actions to recognize parts of the text alternate with actions of constructing the parse results in parser algorithm.

The structure of the analyzed text is reflected in the structure of a regular expressions. The way to use the information about this structure is to include the action code in the analysis process. The action code constructs the results of analysis. The need for interleaving of the analysis and action puts restrictions on how to include action code in the parsing algorithm. Since the actions must be performed after the transition of the machine to the final state, the actions that should be performed after the recognition of subexpressions may be included in the program only by splitting a regular expression into smaller units. The more actions are included in the analysis process, the fewer benefits from a regular expression processor as a software tool, since it reduces the code generated by a regular expression and it augments the percentage of software "glue".

It is proposed to substitute the traditional model of the operation of pattern matching based on the finite state machine model for the model of the pattern algebra. Regular expression language becomes deterministic in the interpretation of the pattern algebra, ensuring the inclusion of the action code not only at the end of the expression, but also after subexpressions.

The regular expression language

Regular expressions are represented as a set of possible formulas of a Kleene algebra. So, Kleene algebra is formal semantics, or interpretation of regular expressions as a formal language. We now recall some basic definitions of formal languages and Kleene algebra that we need throughout the paper. For further details one can use the works of Hopcroft et al. [HMU, 2000] and Kozen [Kozen, 1997].

An alphabet Σ is a nonempty set of symbols. A word w over an alphabet Σ is a finite sequence of symbols from Σ. The empty word is denoted by ε and the length of a word w is denoted by $|w|$. The concatenation " $\cdot$ " of two words w_1 and w_2 is a word $w = w_1 \cdot w_2$ obtained by juxtaposing the symbols of w_2 after the last symbol of w_1. The set Σ^* contains all words over the alphabet Σ. The triple $(\Sigma^*, \cdot, \varepsilon)$ is a monoid.

A language L is subset of Σ^*. If L_1 and L_2 are two languages, then $L_1 \cdot L_2 = \{xy | x \in L_1 \text{ and } y \in L_2\}$. The operator $\cdot$ is often omitted. For $n \geq 0$, the n^{th} power of a language L is inductively defined by $L^0 = \{\}$, $L^n = LL^{n-1}$. The Kleene's star L^* of a language L, is $\bigcup_{n\geq 0} L^n$. A regular expression (r.e.) r over Σ represents a regular language $L(r) \subseteq \Sigma^*$ and is inductively defined like that: ∅ is a r.e. and $L(\emptyset) = \emptyset$; ε is a r.e. and $L(\varepsilon) = \{\}$; $a \in \Sigma$ is a r.e. and $L(a) = \{a\}$; if r_1 and r_2 are r.e., $(r_1 + r_2)$, (rr) and $(r_1)^*$ are r.e., respectively with $L((r_1 + r_2)) = L(r_1) \cup L(r_2)$, $L((r_1 r_2)) = L(r)L(r)$ and $L((r_1)^*) = L(r_1)^*$. We adopt the usual convention that "$*$" has precedence over $\cdot$, and "$\cdot$" has higher priority than "$+$", and we omit outer parentheses. Let RegExp be the set of regular expressions over Σ, and let Reg_Σ be the set of regular languages over Σ. Two regular expressions r_1 and r_2 are equivalent if $(r_1) = L(r_2)$, and we write (the equation) $r_1 = r_2$. The equational properties of regular expressions are axiomatically captured by a KA, normally called the algebra of regular events, after the seminal work of S.C. Kleene [Kleene,1956].

A $\mathcal{K} = (K, 0, 1, +, \cdot, *)$ is an algebraic structure such that $(K, 0, 1, +, \cdot)$ is an idempotent semiring and where the operator "$*$" (Kleene's star) is characterized by a set of axioms. We also assume a relation $\leq$ on K, defined by $a \leq b \Leftrightarrow_{def} a + b = b$, for any $a, b \in K$.

Let S be a chain (a binary relation on Σ^*, which is transitive, antisymmetric, and total) of words w over an alphabet Σ. For chain S we can define the open interval $(a, b) = X$. The constituency set C is the set of intervals in S. This set of intervals contains S and each word $w \in S$ as its elements, and is constructed this way, so any two intervals belonging to C either do not intersect or one of them is contained in the other. The elements of such sets are called constituents. The constituent x dominates the constituent y, if y is a part of x and y differs from x. Constituents of a constituency set of regular expression r are symbols from Σ and operation symbols from the set $\langle \cdot, +, \cdot\star, 1, 0 \rangle$. Given a constituency set C, the analysis process divides up a word into major parts or immediate constituents, and these constituents are in turn divided into further immediate constituents. The process continues until irreducible constituents are reached. The end result of analysis is presented in a tree form that reveals the hierarchical immediate constituent structure of the word at hand. This is a parse tree of regular expression. Regular expressions can express the regular languages, exactly the class of languages accepted by deterministic finite automata.

Let R be a regular expression. We can construct a finite automaton M with $L(M) = L(R)$ recursively. The idea is to use the fact that the set of languages of finite automata is closed under union, concatenation, and Kleene star operations. In general case, R is the constant, either $R = x$ for $x \in \Sigma$, $R = \varepsilon$ or $R = \emptyset$. In all cases, $L(R)$ is finite. Hence, there exists a trivial finite automaton for $L(R)$.

Otherwise, R is an operation applied to one or two smaller expressions. Either $R = R_1 \vee R_2$, $R = R_1 R_2$, or $R = R_1 *$. Since R_1 and R_2 are smaller regular expressions, we can construct automata M_1

and M_2 with $L(M_1) = L(R_1)$ and $L(M_2) = L(R_2)$. Then there exist finite automata for the languages $L(M_1) \cup L(M_2)$, $L(M_1)L(M_2)$ and $L(M_1)*$. Therefore, there exists a finite automaton for $L(R)$. We can construct a finite (nondeterministic) automaton for $L(R)$ by conversion constituency set of R.

The finite automaton is $M = (\Sigma, Q, q_0, T, P)$, where Σ is an alphabet; Q – a finite set of states; $q_0 \in Q$ is the initial state; $T \subset Q$ – a set of terminal conditions; P – the transition function defined by the set of rules in the form of $q_i a_k q_j$, where q_i and q_j are the states, the input symbol $a_k \in \Sigma$ or it is the empty symbol ε. A finite automaton is deterministic finite automaton (DFA), if each of its transitions is uniquely determined by its source state and input symbol, and reading an input symbol is required for each state transition. A nondeterministic finite automaton (NFA) does not need to obey these restrictions. The transitions without consuming an input symbol are called ε-transitions. ε-transitions appear in the component automata, when constituents $R_1 \vee R_2$ or $R*$ are transformed.

An analysis state of the word $v \in \Sigma^*$ is an ordered pair (v, i), with i being an integer $0 \leq i \leq |v|$; $|v|$ is the length of the word. An analysis state (v, i) is the representation of the word in hand in the form of the concatenation $v = br, |b| = i$. A transition rule $q_i a_k q_j$ of an automaton M can be applicable into analysis state (v, i), if it can be represented as a concatenation $v = ba_k r, |b| = i$ and the current state of the machine is q_i. If the automaton M is the NFA then its current state is a set of states and q_i must enter into the set. The application of the transition rule transfers the automaton M to the state q_j (if automaton M is the NFA then q_j becomes an element of the current state) and the current analysis state becomes $(v, i + 1)$. The automaton M is applicable to the analysis state (v, i) if there is a rule for the initial state q_0 applicable in the analysis state (v, i), and after applying all possible transition rules the automaton M is in a terminal state. Initial analysis state (v, l) and the analysis state (v, m), when the machine is in a terminal state, give a representation of the word v as the concatenation $v = bxt, |b| = l, |bx| = m$. The word x is recognized as a part of v by a regular expression R if $L(M) = L(R)$ for the automaton M. The set of all words that are recognized by a regular expression R generates a regular language.

In this interpretation (semantic) of the regular expression language a regular expression is partial function from the analysis states of word in hand. Text analysis with the interpretation of the analysis operation as a partial function from the set of analysis states is the analysis by patterns [Aho, 1990]. String searching algorithms try to find a place where a string called pattern is found within a larger string or text. In most programming languages there is the realization of such an operation, in which the pattern is recognizable word itself (regular expression constructed by concatenation only). The pattern search operation is a lexicographic enumeration of analysis states for the purpose of finding (one or several) occurrences of a pattern. Boost.Regex [Boost, 2002] allows using regular expressions in C++

programs into pattern search operations. As the Boost library is a part of the standard library since C++11 programmers don't depend on Boost.Regex if their development environment supports C++11.

Nondeterministic nature of standard interpretation of the regular expression language

The process of applying transition rules to the automaton M may be fixed in the form of a parse tree in the same way as it is done by parsing. The set of the transition rules of the automaton M is a grammar in which the alphabet of non-terminal symbols is a set of states. A parse tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar. But the set of the transition rules of the nondeterministic automaton M is not a context-free grammar. An internal vertex of the automaton parsing process tree is labeled by the current state (a current state of a non-deterministic machine is a set of nonterminal symbols). If such an internal vertex is split then the terminal vertexes appear, which are dead-ends of analysis. In this case the non-determinism of the standard semantic of regular expressions is manifested. The parse tree cannot be built by linking actions with machine instructions (parsing algorithm does not "catch" subexpressions).

There exists an algorithm (the power set construction) that can transform the NFA M into a more complex DFA with identical functionality. The set of the transition rules of a deterministic automaton is a context-free grammar and a parse tree can be built by linking actions with transition rules of DFA. However, this set of transition rules has no structural similarity with the constituency set of regular expression in hand, and therefore the structure of the regular expression cannot be used during the analysis of a text.

We can use language formed by a limited subset of regular expressions for the analysis of sub-expressions. Regular expressions of this limited subset are transformed into deterministic automata. Some transition rules with an empty input symbol can be deleted preserving equivalence of automata. However, this approach significantly reduces the figurative possibility of regular expression language and significantly increases the risk of programming errors, as the procedure for establishing the properties of determinism is not trivial. This approach is proposed for use in the Ragel State Machine Compiler [Thurston, 2007].

The pattern algebra

A partial function is a function that is defined only on a part of its domain. A McCarthy conditional expression [McCarthy, 1960] has the form $(p_1 \to f_1; p_2 \to f_2; \ldots; p_n \to f_n)$ defining a partial function h, coinciding with one of the functions f_i, where the number i satisfies the following condition:

$$\exists \mathrm{i}(p_i(x) \vee \forall j\left(\left(j < i \Rightarrow \neg p_j\left(((x))\right)\right)\right);$$

where the symbol ¬ denotes the inverse of logical value. If such i does not exist the function h is not specified. Conditional expressions are a device for expressing the dependence of quantities on propositional quantities. Here variables p_i's correspond to propositional expressions and the variables f_i's are expressions of any kind. A propositional expression (predicate) is an expression whose permissible values are T (for truth) and F (for falsity). The rule for determining whether the value of a McCarthy conditional expression is defined can be determined in the following way. Examine the p's from left to right. If a p whose value is T is encountered before any p with undefined value is encountered, then the value of the conditional expression equals to the value of the corresponding f (if it is defined). If any undefined p is encountered before a true p, or if all p's are false, or if the f corresponding to the first true p is undefined, then the value of the conditional expression is undefined.

Let f_i и p_i are symbols of some functions and predicates. The partial conversion function f, specified on the set of analysis states to parse a string, is the pattern, if for any states $\alpha = (w, j), \beta = (v, i), \beta = f(\alpha)$ and $w = v, j \le i$. Let $p(f, \alpha)$ is the predicate whose value is T if the pattern f is defined on the analysis state α. If we have for patterns $f_1 = f_2$ then $p(f_1) = p(f_2)$.

The basic relationship describing a function is that of application. Let a function that applies functions to arguments is called as an apply function. This establishes a one-to-one correspondence between functions of two variables and functions returning functions, which we know under the name of currying. We will use the infix notation for apply function $f\#\beta = \alpha$ that maps a pattern f and the analysis state β into the analysis state $\propto$. Then we introduce the following operations on the patterns.

The catenation fg of patterns f and g is the function defined by the conditional expression $\mathrm{fg}\#\alpha = \mathrm{p}(\mathrm{f}, \alpha) \to \mathrm{g}\#(\mathrm{f}\#\alpha)$. The alternation $\mathrm{f}\vee\mathrm{g}$ of patterns f and g is the function defined by the conditional expression $\mathrm{f}\vee\mathrm{g}\#\alpha = \mathrm{p}(\mathrm{f}, \alpha) \to (\mathrm{f}\#\alpha);\ \mathrm{p}(\mathrm{g}, \alpha) \to (\mathrm{g}\#\alpha)$. The iteration f^* of a pattern f is the function defined by the conditional expression with the infinite number of members $\mathrm{f}*\#\alpha = \neg\mathrm{p}(\mathrm{f}_1, \alpha) \to \alpha;\ \neg\mathrm{P}(\mathrm{f}_2, \alpha) \to \mathrm{f}_1\#\alpha; \ldots; \neg\mathrm{P}(\mathrm{f}_\mathrm{i}, \alpha) \to \mathrm{f}_{\mathrm{i}-1}\#\alpha; \ldots$. The n-th power of a pattern f^n is the function defined by the recursion $\mathrm{f}^1\#\alpha = \mathrm{f}\#\alpha$, $\mathrm{f}^\mathrm{n}\#\alpha = \mathrm{f}\#(\mathrm{f}^{\mathrm{n}-1}\#\alpha)$.

The introduced functions form the pattern algebra the elements of which in contrast to Kleene algebra are the analysis states, i.e. pairs (v, i), where $\mathrm{v}\in\sum^*$ and i is an integer. The nontrivial constants of the introduced algebra are primitive patterns recognizing the occurrence of words consisting of the only character of the alphabet Σ. The constant 1 of this algebra presents the identity pattern. It has the following properties: if f is a pattern, then $1^* = 1,\ 1\mathrm{f} = \mathrm{f},\ \mathrm{f}1 = \mathrm{f},\ 1\vee\mathrm{f} = 1,\ \mathrm{f}\vee 1 = \mathrm{g}$. The pattern g is the everywhere defined pattern if $\mathrm{g}\#\alpha = \mathrm{p}(\mathrm{f}, \alpha) \to (\mathrm{f}\#\alpha);\ \neg\mathrm{p}(\mathrm{f}, \alpha) \to 1$.

The regular expression language agrees with the set of formulas of the pattern algebra. The operation of applying pattern is similar to the process of applying transition rules of an automaton, which was built

by the regular expression. The difference is that a nondeterministic selection of transition rules of the automaton is replaced by the arranged invocation of checks in the expression of alternation operations. By this feature the pattern algebra is deterministic semantics of the language of regular expressions.

The analysis states α and $\beta = f(\alpha)$represents the word v in the form of the concatenation axc, i.e. the pattern f recognizes the word x as a part of v. The set of all words that are recognized by the pattern f forms the language L[f] recognizable by the pattern f. It is easy to verify that $L[f] \subset L(f)$.

The complement operation is included often in the signature of the Kleene algebra for the convenience of practical use. If a and b are regular expressions then the complement $(a-b)$ define s the language $L(a-b)$ of words in $L(a)$ but not in $L(b)$. The negation operation is more convenient in pattern algebra, it is defined as $\neg\#\alpha = \neg p(f, \alpha) \to \alpha$.

The language substitution

The language substitution is a rule of operating strings of symbols. It is an extension of the word substitution rule of Markov algorithm. The language substitution can be specified as a triple (L_1, L_2, ϕ) where L_1, L_2 are languages and the function ϕ: $L_1 \to L_2$ [Cheremisinov, 1981]. The apply function of language substitution ϕ: $L_1 \to L_2$ to input string produces the word W from a word V (i.e. input string) in the following way:

1. Check to see whether any of $x \in L_1$ can be found in the word V.
2. If none is found, the apply function is undefined.
3. If one $x \in L_1$ is found to form word W the first of them can be used to replace the leftmost occurrence of the matched text in the input string with $y = \phi(x) \in L_2$.

A pattern f that recognizes language L[f] with juxtaposed single-valued function ϕ: $L[f] \to L_2$ is a pattern with side effect [Sebesta, 2009]. The operation $\beta = f(\alpha)$ represents the basic effect of the pattern. The side effect of a pattern can be another pattern. The patterns with side effects are the types of recursive functions. As a class of general recursive function coincides with the class of Turing computable functions (Turing–Church thesis), so patterns with side effects are the effectively calculable functions. The class of patterns with side effects is Turing complete or computationally universal. The consequence of the algorithmic completeness of the regular expression patterns with side effects results in the possibility of recognition of any type of language in Chomsky's classification, not just the class of regular languages.

Algebra of patterns with a side effect is built by the modification of the interpretation of the constant 1 of this algebra, i.e. the identity pattern. In this algebra only the identity patterns has side effects. Algebra of patterns with a side effect contains the set of identity patterns, which differ by their side effects. An

identity pattern with a side effect corresponds to a function that is applicable to the analysis state when the previous pattern was applicable too. If the side effect defines the substitution of a word that was recognized by the previous pattern, then the catenation of the identity pattern with the side effect corresponds to a word substitution rule of Markov algorithm. The regular expressions of the algebra of patterns with a side effect define Markov algorithms. The application of this regular expression to the word V is the substitution of languages. Thus, regular expression of the algebra of patterns with a side effect is the algorithm for transforming words by substitution rule based on the constituents set of the regular expression. The construction of this algorithm is implemented by "embedding" the identity pattern with the side effect into the analysis procedure. Operations representing the side effects of identity patterns specify the algorithmic basis of language substitution functions.

Regular expression with the semantics of the pattern algebra could be imagined as a program on a special programming language. Application operations of the primitive patterns define the set of operations that make up this language. The control operations of the programming language correspond to the functional forms of regular expression. They specify the means of sequencing the application of primitive patterns. The variables whose values represent parsing states are not explicitly referred to the program text, their existence is assumed for each such program.

The prototypes of the algebra of regular expressions with a side effect are macro languages. In these languages a macroinstruction is a rule or pattern that specifies how a certain input sequence (macro-call) should be mapped to a replacement output sequence according to a defined procedure (macro-procedure). The mapping process that instantiates (transforms) a macro use into a specific sequence is known as macro expansion. Language substitution (L_1, L_2, ϕ) in the form of macro is described by a pattern describing characteristics of language L_1 and by algorithm of constructing the language L_2. In the case of language substitution in the form of regular expression, the constituents of a regular expression are defined as macro-call formal parameters; words that are recognized by constituents are the actual parameters; the side effect forms macro-procedure. The regular expression language is similar to XSLT language, but unlike XSLT, the regular expression language is better suited to handle unformatted text.

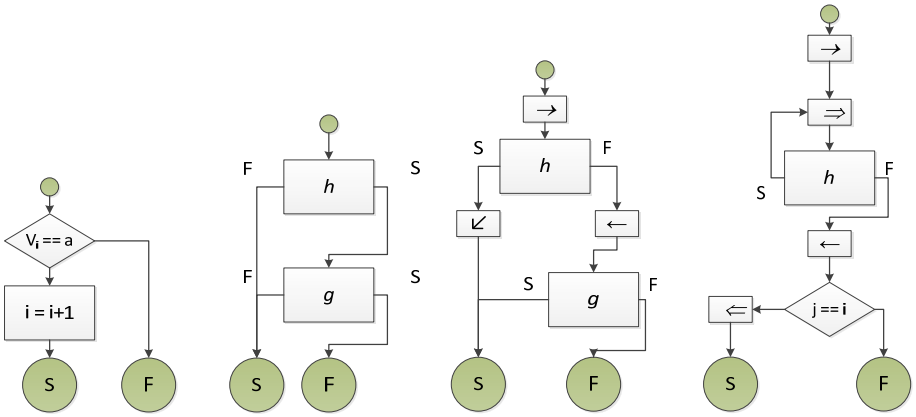
The compilation of regular expressions with a side effect

Let U be a set of variables and V be a set of possible values of variables of U. Let us define a machine Ω which memory states are functions from U into V, so the set S of all memory states coincides with the set of functions $s: U \to V$. The control state of the machine Ω is a pair (v,i) where v is a word and a i – an integer. $v \in L(\Omega)$ defines the text of the program for the machine Ω, $v \in \Sigma^*$. Let the programming language $L(\Omega)$ is a regular expression language. The set $\Psi(v)$ of

control states of program v of the machine Ω coincides with a set of analysis states of program v. The set $\Psi(\Omega)$ is the union of $\Psi(v)$ for all programs of the language $L(\Omega)$.The instruction set of Ω is a set of patterns with side effects defined on Cartesian product $\Psi(\Omega)*L(\Omega)$. If the pattern f is an instruction of Ω, then the set of analysis states coincides with the set of control states where the instruction is applicable. The side effect of f is the function that transforms the given memory state into resulting one. The machine Ω is specified by the pattern $(f_1 \vee f_2 \vee ... \vee f_n)*$, where f_i is a member of the instruction set of the machine Ω. The regular expressions with side effect are programs of $L(\Omega)$.

Given the program p of the machine Ω. For each control state $\alpha \in \Omega(p)$ we can describe a transition function $\pi_i : S \to \Omega(p)$, that defines control state of the machine Ω after execution of an instruction allowable in control state α. The union of all π_i specifies a function $\pi:\Omega(p)*S \to \Omega(p)$ that is denoted as a schema of p. The schema of p can be represented by a graph $G(\Omega(p),\pi)$. The vertices of G correspond to the control states and the edges are marked with values of memory variables from U. A program scheme describes (models) a concrete program. Concrete programs can be obtained from schemata by means of interpretation which consists in bringing some concrete variables and operations into correspondence with formal variables and operations.

An instruction of the machine Ω that changes only the state of the memory is called a converter, the command that does not alter any memory state is called a resolver. Resolvers are patterns without side effect. In graphical scheme representation of recognizers are denoted by diamonds, converters – by rectangles (Figure 1).



a) b) c) d)

Figure 1. The templates of program schemata which perform analysis of character strings

Memory U for the machine Ω performing analysis of character strings is composed of the variables u,i representing the analysis state of the parse string; stack to store numeric values and work variable j of integer type. The converters with labels $\rightarrow$, and $\leftarrow$ $\swarrow$ define the operations that store he variable i on the stack, which is empty at the beginning; load the variable i from the stack; delete the top-most element of the stack, respectively. The converters marked by $\Leftarrow$ or $\Rightarrow$ define the operations that push the work variable j into stack or move j into i.

The program flow of program scheme p in the memory state s_0 is defined as follows. The program flow is the travel under the scheme step by step. A program scheme has always one active instruction, which is pointed by a control state. The program flow starts from the control state $(p,0)$ in a state s_0 of the memory U of the machine Ω. The execution of a converter y is the transition of memory state x into state $y(x)$. The execution of a recognizer does not change the current memory state but selects one of the outgoing arcs of the resolver to continue travel under the scheme. If recognizer is labeled by a variable u, then we select the arc labeled by the value $u(x)$. Execution of the program scheme is completed when traveling lead us to the scheme output. The current memory state is a result of execution of the program scheme in this case. Otherwise, the result is undefined. Thus, we decided to consider program scheme as an imperative language and to use a structural operational semantics. The operational semantic system in its entirety is an interpreter that links the program text to the set of possible executions.

For the synthesis of the program scheme that implements the algorithm specified by the pattern a we use reverse Polish notation of a pattern R. The synthesis algorithm uses a stack of a program schemes. We parse reverse Polish notation of the pattern R. If current token is a primitive pattern, which recognizes a single symbol, then we push into the stack the program scheme on Figure 1,a. If current token is an operation of a regular expression, then we form new program scheme from the popped one and push it into the stack. To form a new program scheme we use stereotype of the program scheme on Figure 1,b (1,c,1,d) if an operation is the catenation (alternation, iteration). As a result the stack contains the only scheme which is a program scheme that implements this pattern R, if the initial regular expression is syntactically correct.

The interpretation of converters and recognizers of program schemes as expressions of conventional programming language is the basis of the compiler of regular expressions, which performs the analysis of character strings. Selection of the object language is mainly determined by the capabilities of the programming system in which the language is used as an input. Object language should allow the use of character string data type, and where there is the possibility to access the individual characters in a string for this string data type.

The regular expression language and a compiler form a programming system of patterns which performs analysis of character strings. Currently the compiler of this system is a preprocessor that converts patterns into programs in the programming language C. To represent the analysis state of the character string the pointer $char*A$ is used in programs in C. The stereotype of primitive patterns (Figure 1,a) is given by $\mathrm{int\ B = *(A++) == n;}$ where n is a recognizable symbol. The catenation (Figure 1,b) is the stereotype $h\ if(!B)\{\ g\ \}$. The alternation (Figure 1,c) is the stereotype $\mathrm{P}.push\_back\ (\mathrm{A});\ h\ if\ (\mathrm{B})\{\ \mathrm{A}\ =\ P.pop\_back\ ();\ g\ \}\ else\ \{\ \mathrm{P}.pop\_back\ ();\ \mathrm{B}\ =\ 1;\}$. The iteration (Figure 1,d) is the stereotype

$do\ \{\ \mathrm{P}.push\_back\ (\mathrm{A});\ if\ (!*B)\{\mathrm{B}\ =0;break;\}\ h\ \mathrm{B}\ =(!\mathrm{B});\}while\ (!\mathrm{B}\ );\ if\ (!\mathrm{B}\ )\mathrm{B}\ =-1;else\ \{\mathrm{A}\ =\mathrm{P}.pop\_back();\mathrm{B}\ =0;\}$□

Analyzers based on deterministic finite automaton have linear time complexity $O(|S|)$ for strings of length $|S|$, because it does not need to be rolled back (do not check twice a symbol of the analyzed text). The analyzers, which are built by programming system of patterns, have the rollbacks as it is seen from the stereotypes on Figure 1.

Conclusion

We have presented an operational semantics for the regular expression language. Our semantics of regular expression gives the basis for building programming tools for the language analysis. The pattern algebra represents both functional model of the constituent analysis of texts and the control flow model of primitive templates execution during text analysis. Generally accepted formalism to describe the structure of the constituents of the text is the grammar. The use of patterns with a side effect is able to make the universal formalism from the regular expressions, as well as grammar. The implementation of string analysis based on regular expression patterns with a side effect is potentially more effective than the grammar parsing algorithm, because it is possible to control the sequence of applications of grammar rules. The grammar parsing algorithm is not a part of the grammar formalism. Using grammar rules as patterns is more natural in the case of a language substitution, especially in simple cases, compared with the attribute grammars, because to compute language substitution $f\ (x)\ =\ y$ by the attribute grammars we have to take into account the parsing algorithm. However, the concern about the control of the sequence of grammar rules is not so good. The optional degree of freedom may increase the complexity of the description analysis.

The programming system of regular expression patterns has been successfully used for the construction of the analyzers of interchange data formats [Cheremisinov, 2013].

Bibliography

[Friedl, 2006] J.E.F.Friedl. Mastering Regular Expressions, 3rd Edition, O'Reilly, Sebastopol, CA, 2006.

[HMU, 2000] J.Hopcroft, R.Motwani, and J.D.Ullman. Introduction to Automata Theory, Languages and Computation. Addison Wesley, 2000.

[Kozen, 1997] Dexter Kozen. Automata and Computability. Undergrad. Texts in Computer Science. Springer-Verlag, 1997.

[Kleene,1956] S.C.Kleene. (1956), Representation of events in nerve nets and finite automata, in Claude Shannon & John McCarthy, ed., 'Automata Studies' , Princeton University Press, Princeton, NJ , 1956, pp. 3-41 .

[Aho, 1990] Aho A. Algorithms for finding patterns in strings // Handbook for theoretical computer science, MIT Press. – Vol. A.,1990. – P. 257-300.

[Boost, 2002] Jeremy Siek, Lie-Quan Lee and Andrew Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2002.

[Thurston, 2007] A. D.Thurston. Ragel State Machine Compiler User Guide http://www.colm.net/files/ragel/ragel-guide-6.9.pdf, 2007.

[McCarthy, 1960] J.McCarthy. Recursive function of symbolic expressions and their computation by machine, part 1. Comm. ACM – 1960, v.3, n. 4. – P. 184-195.

[Cheremisinov, 1981] D.I.Cheremisinov. Language substitution programming // Programming, Moskva: Nauka, – 1981. – № 5. – P. 30-37.

[Sebesta, 2009] R.W.Sebesta. Concepts of Programming Languages, 9th Edition, Addison-Wesley, 2009.

[Cheremisinov, 2013] D.I.Cheremisinov. "Design automation tool to generate EDIF and VHDL descriptions of circuit by extraction of FPGA configuration,"*Design & Test Symposium, 2013 East-West*, Rostov-on-Don, 2013, pp. 1-4.

Authors' Information

Dmitry Cheremisinov *– The United Institute of Informatics Problems of National Academy of Sciences of Belarus, leading researcher, Surganov str., 6, Minsk, 220012, Belarus; e-mail: cher@newman.bas-net.by*

Major Fields of Scientific Research: Logic design automation, System programing