УДК 681.51
doi: 10.18101/2304-5728-2016-4-23-33

***© Л. Н. Федорченко***

**Алгоритмы построения состояний анализатора для КСР-языка**

Рассматриваются алгоритмы построения состояний анализатора языка, определяемого специальной трансляционной контекстно-свободной грамматикой, правые части правил которой заданы в виде обобщённых регулярных выражений над объединённым алфавитом терминалов, нетерминалов и семантик.

**Ключевые слова:** КСР-грамматики, обобщённое регулярное выражение.

***© L. N. Fedorchenko***

**Algorithms for constructing parser states for CFR language**

We consider algorithms for constructing parser's states determined with the special translational context-free grammar in regular form.

**Keywords:** CFR-grammars, generalized regular expression.

## Введение

В настоящее время языковые технологии активно включаются в различные сферы нашей жизни, что привело к развитию современных транслирующих систем, использующих принцип синтаксического управления для производства программ обработки данных и ориентированных на разнообразный набор вычислительных устройств, применяемых как в производстве, так и в быту.

Опыт использования различных инструментальных систем [1–4] автоматизирующих построение трансляторов показывает, что его удобно задавать в виде КС-грамматики в регулярной форме (КСР-грамматики) [3–5], которая отличается от обычной КС-грамматики тем, что в ней помимо алфавитов нетерминалов и терминалов имеются дополнительные алфавиты *контекстных символов* (*семантик*) и предикатов, а правые части правил представляют собой обобщённые регулярные выражения над символами объединённого алфавита грамматики.

Управление процессом анализа определяется синтаксической структурой предложений входного языка и сосредоточено в управляющей таблице магазинного автомата, а сама программа анализа обращается к ней на каждом шаге её работы. Управляющая таблица содержит состояния магазинного автомата, определяющего порядок вызова семантик, проверку их на непротиворечивость предикатами в процессе синтаксического анализа входного потока.

Предлагаемый подход к синтезу состояний анализатора состоит в использовании синтаксической граф-схемы [3], построенной по КСР-грамматике. Определение регулярного выражения как формулы с тремя операциями: объединения, конкатенации и итерации над конечными множествами символов языка позволяет представить его в виде ориентированного графа [3], в котором узлы помечены символами языка, а ориентированные дуги определяют порядок их обхода. Этот граф является графическим аналогом правила грамматики для одной грамматической конструкции КСР-языка. Он используется как основа для генерации состояний магазинного автомата-преобразователя языка, представляемого этим графом.

В работе даётся определение состояния магазинного автомата и рассматривается алгоритм построения множества состояний анализатора для КСР-языка и его реализация.

Напомним основные определения.

## 1. Основные определения

Пусть дан алфавит символов $V = \{a_1, a_2, ... a_n\}$. $V^*$ – множество всех слов в алфавите $V$.

**Определение 1.** *Регулярным множеством* слов (регулярным языком) над алфавитом $V$ называют следующие множества:

1. $e = \{\varepsilon\}$ – (пустое слово);
2. $\emptyset$ – (пустое множество слов);
3. $\{a_i\}$ – элементарные множества слов над алфавитом $V$ для любого $a_i \in V$;
4. Множества $P \cup Q, PQ, P^*$, где $P$ и $Q$ – регулярные множества в алфавите $V$.

Добавим к традиционному набору операций операцию бинарной обобщённой итерации (#) над множествами слов в алфавите $V$. Она может быть определена через традиционную (унарную) операцию Клини как $P\#Q = P,(Q,P)^*$.

**Определение 2.** *Обобщённым регулярным выражением* множества $A$ называется слово $r(A)$ над расширенным алфавитом $W$, где $W = V \cup \{\#, *, +, ;, (,), \varepsilon, \emptyset\}$, причем:

1. Если $A = \emptyset$, то $r(A) = \emptyset$;
2. Если $A = e$, то $r(A) = \varepsilon$;
3. Если $A = \{a\}$, где $A \in V$, то $r(A) = a$;
4. Если $r(P) = p$ и $r(Q) = q$ – регулярные выражения для множеств $P$ и $Q$ тогда:

$r(A)=(p;q)$ для множества $(P\cup Q)$;
$r(A)=(p,q)$ для множества $(PQ)$;
$r(A)=(p^*)$ для множества $P^*$;
$r(A)=(p^+)$ для множества $P^+$;
$r(A)=(p\#q)$ для множества $(P\#Q)$.

*Ничто другое не является обобщённым регулярным выражением.*

Для представления этого множества цепочек используется понятие *синтаксической граф-схемы* (СГС), являющейся графовым аналогом правил КСР-грамматики [3], а вывод в грамматике заменяется более простой структурой – маршрутом (или путём) в СГС.

**Определение 3.** Под синтаксической граф-схемой понимают набор конечных ориентированных графов с помеченными вершинами и дугами. Каждый граф соответствует одному правилу КСР-грамматики и называется графом для нетерминала, определяющего КСР-правило.

Две вершины графа $\Gamma_A$ для нетерминала $A$ являются входными и выходными с метками $E_A$ и $F_A$. Внутренние вершины помечены терминальными и нетерминальными символами – операндами регулярного выражения правой части правила, определяющего нетерминал $A$, а дуги помечены контекстными символами (семантиками) – именами семантических процедур, которые должны исполняться по ходу синтаксического анализа.

Покажем графические представления основных операций над элементарными регулярными
выражениями

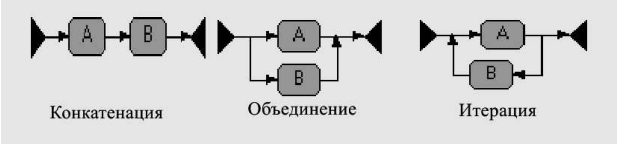



Рис. 1 Графическое представление основных операций над регулярными выражениями

Так задаются базисные элементы, к которым рекурсивно сводятся более сложные случаи.

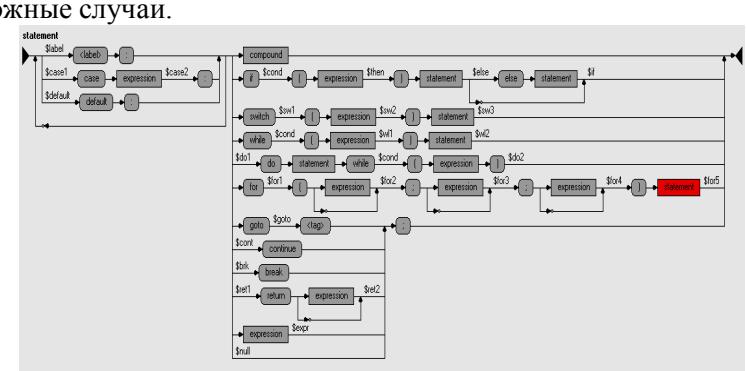



Рис.2. Граф для нетерминала statement

Для описания всевозможных маршрутов в граф-схеме грамматики её можно представить в линейной записи в виде массива. Рассмотрим пример.

**Пример 1**. Пусть КСР-грамматика $G_1 = (\{S,A,B,Q\}, \{a,b\}, P, S)$:
$P$ – конечное множество правил вида

$S$: $A,Q$ ; $B,b$.

$B$: $A,Q$.

$A$: $a$.

$Q$: $a,(Q;\varepsilon)$.

Здесь знак ‘,’– конкатенация, знак ‘;’ – объединение двух регулярных множеств цепочек.

Синтаксическую граф-схему для $G$ в линейной записи можно представить следующим образом (левый столбец – адреса массива, правый – символы грамматики и метазнаки, управляющие переходами по адресам массива).

*начало S*
0. ⟨ 5
1. A
2. Q
3. ↓7
4. B
5. b
6. *конец S*
7. *начало B*
8. *A*
9. *Q*
10. *конец B*
11. *начало A*
12. *a*
13. *конец A*
14. *начало Q*
15. *a*
16. ⟨19
17. *Q*
18. *конец Q*

По линейной записи граф-схемы строятся состояния автомата – анализатора языка.

*Состояние автомата* – это некоторое количество информации о распознаваемом языке. В данном случае под состоянием автомата понимаем множество вершин граф-схемы. Переход от одного состояния к другому, которое также является множеством (возможно пустым) вершин из граф-схемы, управляется текущим символом, поступающим из входного текста. Переходное состояние содержит информацию о том, какие символы допустимы для следующего перехода. Поэтому состояние содержит информацию о том, какие текущие символы разрешены в данный момент, а,

следовательно, какие цепочки (подслова) допустимы к моменту перехода в это состояние. Отсюда, состояние позволяет установить принадлежность слова языку, порождаемому данной компонентой граф-схемы.

**Определение 4.** *Состоянием в СГС* $\Gamma_G$ называется:

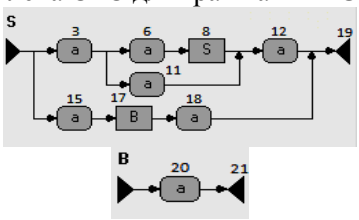
1. Либо *выходная* вершина $F_{A_i}$ графа $\Gamma_{A_i}$ для некоторого нетерминала $A_i$ из алфавита $N$;
2. Либо *терминальная* вершина в $\Gamma_G$;
3. Либо *пара* множеств $(S_1, S_2)$, где $S_1$ и $S_2$ – состояния в $\Gamma_G$;
4. Либо *объединение* состояний в $\Gamma_G$.

**Пример 2.** Рассмотрим грамматику $G_2$:

$$G = (\{S, B\}, \{'a'\}, P, S),$$

где P= $\{(1)S :'a', ('a', S;'a'), 'a';'a', B,'a'.\ (2)B :'a'.\}$, ($S$ – стартовый нетерминал)

На рисунке 3 представлена СГС для грамматики $G_2$.




Пронумеруем терминальные и нетерминальные вершины в возрастающем порядке. Нумерация происходит не по всем натуральным числам. Почему так, будет объяснено позже (см. описание алгоритма).

*Множество состояний в граф-схеме грамматики* $G_2$, построенное по определению 4 следующее:

$S_{T_0} = \emptyset$

$S_{T_1} = \{a-3, a-15\}$

$S_{T_2} = \{\{a-6, a-11\}, (\{a-20\}, B-17)\}$

$S_{T_3} = \{\{(\{a-3, a-15\}, S-8), a-12\}, (Ex_B - 21, B-17)\}$

$S_{T_4} = \{\{(\{a-6, a-11\}, (\{a-20\}, B-17), S-8), Ex_S - 19\}, \{a-18\}\}$

$S_{T_5} = \ldots$

Состояние $S_{Ti}$ дает нам представление возможных передвижений на *i*-ом шаге. В фигурных скобках находятся элементарные состояния перехода, а в круглых скобках находятся переходное состояние и состояние возврата. Если мы будем знать *n* состояний, то проверка принадлежности цепочки данной КСР-грамматики будет осуществляться за линейное время O(n).

## 2. Постановка задачи

Целью данной работы является построение множества состояний КСР-грамматики для последующей проверки входных цепочек символов на принадлежность их языку. Предложенная КСР-грамматика не должна иметь левой рекурсии, так как в этом случае алгоритм не закончит работу (зациклится). Леворекурсивная грамматика может быть приведена в КСР-грамматику без левой рекурсии посредством эквивалентного преобразования [3].

## 3. Описание алгоритма

Алгоритм построения КСР-анализатора состоит из трех основных этапов:

**Этап 1.** Преобразование граф-схемы в представление, приближенное к машинному (в виде одномерного массива).

**Этап 2.** Построение стека состояний.

**Этап 3.** Построение последовательности состояний анализатора.

***Преобразование граф-схемы в представление, приближенное к машинному***

Низкоуровневое представление граф-схемы строится по предложенной КСР-грамматике. В *определении 6* были описаны условные обозначения для построения данного массива.

Машинное представление строится следующим образом: каждое из КСР-правил анализируется путем последовательного прохождения терминальных и нетерминальных вершин. Вход по нетерминалу, а также разветвления путей заносятся в очередь. Когда анализ КСР-правила дошел до конца правила, из очереди достается первая запись. Аналогичным образом анализируется она, и все последующие записи в очереди.

Для примера из рисунка 3 граф-схема в линейной записи будет вида:

Таблица 1.

Граф-схема в линейной записи

| **Адрес** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | **:** | **<** | **15** | **‘a’** | **<** | **11** | **‘a’** | ***** | **1** | **↓** |
| **1** | **12** | **‘a’** | **‘a’** | **↓** | **19** | **‘a’** | ***** | **20** | **‘a’** | **$Ex_S$** |
| **2** | **‘a’** | **$Ex_B$** | | | | | | | | |

***Построение стека состояний***

Когда граф-схема представлена в удобном для анализа виде, по ней формируется стек состояний. На рисунке 3 показан стек состояний, построенный по граф-схеме из таблицы 1.

Стек состояний формируется снизу-вверх. В примере начальный элемент по адресу 200 (дно стека). Каждая секция имеет следующую структуру: длина секции (количество ячеек), хеш-функция (например, произведение значений всех элементов секции). Далее сами элементы (термина-

лы, указанные номером ячейки в линейной записи граф-схемы, и нетерминалы – адресами (число со знаком минус – переходное, а положительное – возвратное состояние)).

Первый блок стартовый, состояние $S_{T_0}$ . Следующее состояние $S_{T_1}$ показывает, что мы можем перейти по двум символам, находящимся в 15 и 3 ячейке массива линейного представления граф-схемы.

| Адрес | Значение |
|---|---|
| 133 | 3 |
| 134 | -168 |
| 135 | 158 |
| 136 | 2 |
| 137 | hash |
| 138 | 13 |
| 139 | 5 |
| 140 | hash |
| 141 | -165 |
| 142 | 189 |
| 143 | -149 |
| 144 | 178 |
| 145 | 3 |
| 146 | hash |
| 147 | -149 |
| 148 | 178 |
| 149 | 4 |
| 150 | hash |
| 151 | 22 |
| 152 | -195 |
| 153 | 158 |
| 154 | 3 |
| 155 | hash |
| 156 | -195 |
| 157 | 158 |
| 158 | 2 |
| 159 | hash |
| 160 | 19 |
| 161 | 3 |
| 162 | hash |
| 163 | -195 |
| 164 | 165 |
| 165 | 2 |
| 166 | hash |
| 167 | 12 |

| Адрес | Значение |
|---|---|
| 168 | 3 |
| 169 | hash |
| 170 | -172 |
| 171 | 189 |
| 172 | 2 |
| 173 | hash |
| 174 | 21 |
| 175 | 5 |
| 176 | hash |
| 177 | -192 |
| 178 | 189 |
| 179 | 11 |
| 180 | 6 |
| 181 | 3 |
| 192 | hash |
| 183 | 11 |
| 184 | 6 |
| 185 | 3 |
| 186 | hash |
| 187 | -192 |
| 188 | 189 |
| 189 | 2 |
| 190 | hash |
| 191 | 18 |
| 192 | 2 |
| 193 | hash |
| 194 | 20 |
| 195 | 3 |
| 196 | hash |
| 197 | 15 |
| 198 | 3 |
| 199 | 1 |
| 200 | 1 |

Рис.3. Стек состояний, построенный по граф-схеме из таблицы 1

Следующие два блока иллюстрируют вхождение по нетерминалу на символ в ячейке 20, а также ячейку возврата 18. Следующий блок объединяет их, говоря о том, по какому адресу находится возвратное состояние, а по какому переходное. По адресу 175 находится состояние $S_{T_2}$ . Данное состояние составное, оно включает в себя все блоки вниз до предыдущего состояния, а также может использовать любые другие блоки ниже данного.

Чтобы составить *i*-ое состояние алгоритм использует предыдущее состояние и рекурентно сканирует каждый блок, наращивая *i*-ое состояние.

Когда множество состояний построено на вход программы анализатора могут подаваться различные цепочки. За линейное время по переходам от состояний к состояниям будет составлен маршрут. Он будет включать по-

следовательность терминальных символов с их номерами в линейном представлении граф-схемы, каждый вход и выход по нетерминальным символам, глубину захода в каждый момент времени, а также поддерживать все такие маршруты.

## 4. Алгоритм построения анализатора

```
НАЧАЛО:          top:=0;
      State:= initialstate;
NEXT symbol:     symbol:=insymbol;
SAME symbol:     finals:=∅;
                 if state ≠ final state
                 then level:=state;
PROCESS stack:   nonterminals:=nonterminal(level);
                 if nononterminals=level
                 then stack[top+:=1]:=nonterminals;
                      level:=first(level); goto PROCESS stack
                 else  fin:= end (level);
                      if fin ≠ ∅
                      then   finals:=fin;
      ftop:=top
                      end if
                 end else;
                 partial state:= terminal(level)/symbol;
                 if partial state ≠ ∅
                 then   outset (partial state);
                 state:=successors(partial state);
goto NEXT symbol
                 end if;
                      nonterminals:=nonterminal(level);
                      if nonterminals ≠ ∅
                      then   stack[top+:=1]:=nonterminals;
                             level:=first(nonterminals); goto PROCESS stack
                      end if;
                      if finals ≠ ∅
                      then outset (finals);
                             state:=stack[top:=ftop]/mask(finals);
                             outset(state);
                             top:=top-1;
      state:=successors(state);
                             goto SAME symbol
                      else ERROR
                      end if;
                 end if;
                 outset(state);
```

## 5. Описание модулей и классов

В этом разделе будут описаны основные классы и структуры, используемые в программе на языке C++. Обращаю внимание, что это только часть классов и методов в них, так как остальные, не приведенные здесь, не представляют особого интереса. В частности, это – вспомогательные классы и методы, упрощающие работу, связанную с распознаванием текста, работой с визуальной составляющей программы и т.д. Автором реа-

лизации алгоритмов является студент 3-го курса СПбГУ Слёзкин Никита Евгеньевич.

Структура **ControlTableType** описывает элемент линейного представления граф-схемы. Подобный элемент может быть четырех видов: символ, адрес перехода, выход из какого-либо правила и операции.

```
enum type { CHAR, ADDRESS, EXIT, OPERATIONS };

struct ControlTableType
{
   type elem_type;
   char symbol;
   int int_value;
}
```

Класс **BuildControlTable** отвечает за построение и хранение линейного представления граф-схемы по КСР-грамматике, а также за удобное использование имеющимися данными. Методы класса позволяют получить различную информацию по значениям граф-схемы в линейном виде и т.д.

```
 class BuildControlTable
 {
    private:
        int quantity_of_rules;
        string ksr_gr[];
        ControlTableType ControlTable[];
    public:
        BuildControlTable(string[] reg, int
 quant_rules);
        ControlTableType GetValue(int in-
 dex_of_table);
        ControlTableType StartRule(int num-
 ber_of_rule);
        ControlTableType FinishRule(int num-
 ber_of_rule);
        Type GetType(int index_of_table);
 }
```

Класс **BuildStateStack** отвечает за построение стека состояний распознавателя. В конечном итоге после работы конструктора данного класса на выход получается массив stack[], алгоритмически построенный по вышенаписанным правилам (см. Описание алгоритма). С помощью специальных методов класса, можно провести двустороннюю связь между состоянием граф-схемы и индексом элемента в стеке. В дальнейшем, используя имеющиеся данные, мы должны научиться проверять принадлежность цепочек символов.

```
class BuildStateStack
{
   private:
       int StatesAddress[];
       int stack[];
       BuildControlTable b_c_table;
   public:
       BuildStsteStack(BuildControlTable BCTable);
       int GetStateByIndex(int index);
       int GetIndexByState(int state);
}
```

Классы **StringPath** и **CheckingStrings** работают в паре. **StringPath** хранит полученный путь в ходе передвижения по состояниям. Так как таких путей может быть больше чем один, paths является массивом. С помощью конструктора **CheckingString** добавляется полученный ранее стек состояний, а с помошью метода **Checking** добавляется цепочка символов, которую как раз и надо проверить на принадлежность грамматике. Эти две функции разделены на две части специально для того, чтобы при желании проверить несколько строк не нужно было пересобирать стек состояний каждый раз. Стек состояний привязывается один раз вначале создания объекта.

С помощью метода **NextState** объект типа **CheckingStrings** каждый вызов переходит на одно состояние, таким образом можно проследить до какого момента цепочка удовлетворяет грамматике.

```
class StringPath
{
   private:
         int path[];
   public:
       StringPath();
       void WritePathWithDeep();
}
```

```
class CheckingStrings
{
   private:
       BuildStsteStack b_s_stack;
       int number_of_paths;
       int state_now;
       StringPath paths[];
   public:
       CheckingStrings(BuildStsteStack BSStack);
       int NextState();
       int Checking(string string_value);
}
```

## Заключение

В результате проделанной работы имеется программа, строящая линейное представление граф-схемы и состояний анализатора для КСР-языка, с последующей проверкой цепочек на их принадлежность языку, порождаемому данной грамматикой.

В дальнейшей работе предполагаются следующие добавления:

1. Проверка входных грамматик на синтаксическую правильность;
2. Возможность задавать рекурсивные грамматики и выполнять преобразования;
3. Возможность использовать семантики и предикаты.

Представленная разработка намного эффективнее в плане времени в сравнении с построением интерпретатора. Это достигается за счет того, что блок создания состояний в анализаторе работает один раз вначале программы, все следующие операции происходят линейно относительно длины записи входной цепочки символов.

**Литература**

1. G. Rozenberg A. Salomaa (Eds.) Handbook of Formal Languages. Vol. 2. Berlin, Heidelberg, New York. Springer-Verlag, 1997. 527 p.
2. Bison – GNU parser generator. — URL: http://www.gnu.org/software/bison/.
3. Федорченко Л. Н. Регуляризация контекстно-свободных грамматик / LAP LAMBERT Academic Publishing GmbH & Co. KG Dudweiler Landstr. 99, 66123 Saarbrucken, Germany. 2011. С. 180.
4. Федорченко Л. Н. Синтаксически управляемая обработка данных для практических задач // Вестник БГУ. — 2013. — № 9. — С. 87 – 99.
5. Ludmila Fedorchenko and Sergey Baranov Equivalent Transformations and Regularization in Context–Free Grammars // Bulgarian Academy of Sciences / Cybernetics and Information Technologies (CIT). Vol. 14. No 4. Pp.11 – 28. Sofia 2015.

**References**

1. G. Rozenberg A. Salomaa (Eds.) Handbook of Formal Languages. Vol. 2. Berlin, Heidelberg, New York. Springer-Verlag, 1997. 527 p.
2. Bison – GNU parser generator. — URL: http://www.gnu.org/software/bison/.
3. Fedorchenko L. N. Reguljarizacija kontekstno-svobodnyh grammatik / LAP LAMBERT Academic Publishing GmbH & Co. KG Dudweiler Landstr. 99, 66123 Saarbrucken, Germany. 2011. C. 180.
4. Fedorchenko L. N. Cintaksicheski upravljaemaja obrabotka dannyh dlja prakticheskih zadach // Vestnik BGU. — 2013. — № 9. — S. 87 – 99.
5. Ludmila Fedorchenko and Sergey Baranov Equivalent Transformations and Regularization in Context–Free Grammars // Bulgarian Academy of Sciences / Cybernetics and Information Technologies (CIT). Vol. 14. No 4. Pp.11 – 28. Sofia 2015.

*Федорченко Людмила Николаевна,* кандидат технических наук, старший научный сотрудник лаборатории прикладной информатики Федерального государственного бюджетного учреждения науки Санкт-Петербургский институт информатики и автоматизации Российской академии наук (СПИИРАН), e-mail: LNF@iias.spb.su

*Fedorchenko Ludmila Nikolaevna,* Senior Researcher, PhD, St.Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences (SPIIRAS).