

Санкт-Петербургский государственный университет
Математико-механический факультет

КРАВЧЕНКО ЕВГЕНИЙ АРТУРОВИЧ

РЕФЕРАТ
**СИНТЕЗ РАСПОЗНАВАТЕЛЯ КСР-ЯЗЫКА
ПО СИНТАКСИЧЕСКОЙ ГРАФ-СХЕМЕ**

Математическое обеспечение и администрирование
информационных систем

Группа №344

Санкт-Петербург 2017

1 Введение

На сегодняшний день область применимости языковых технологий очень велика. Они активно используются в различных сферах нашей жизни, что привело к созданию множества различных программ для обработки данных, использующих принципы синтаксического анализа.

Опыт построения систем для автоматического построения трансляторов показывает, что их удобно задавать с помощью КСР(КС грамматики в регулярной форме) грамматик. Процесс анализа при этом проводится с помощью магазинного автомата. На каждой итерации программа анализа обращается к управляющей таблице этого автомата для определения следующего состояния. Такой автомат можно построить по синтаксической граф-схеме КСР языка. Регулярное представление правил грамматики позволяет представить их в виде ориентированных графов. На их основе строятся состояния автомата.

В данной работе рассматривается алгоритм построения состояний распознавателя КСР-языка и его реализация.

2 Постановка задачи

Задача - по заданной КСР-грамматике построить КСР-распознаватель, проверяющий входные цепочки символов на принадлежность КСР-языку, порождаемому этой грамматикой. При этом на КСР-грамматику накладывается ограничение в виде отсутствия левой рекурсии.

3 Описание и структура программы

Программа состоит из трех этапов:

1. По КСР-грамматике построить эквивалентную ей линейную граф-схему
 - a Закодировать входную грамматику
 - b Полученную последовательность чисел преобразовать в граф-схему
2. По граф-схеме построить состояния КСР-распознавателя
 - a Сгенерировать множество состояний
 - b Удалить эквивалентные состояния
3. Построить распознаватель

4 Представление данных

Программа принимает текстовый файл, содержащий запись КСР-грамматики в виде последовательности " $N : r(A).$ ", где N - нетерминал, $r(A)$ - некоторое обобщенно-регулярное выражение. При этом первый нетерминал считается начальным. Файл должен заканчиваться

строкой "Eogram".

Синтаксическая граф-схема - это графовый аналог правил КСР-грамматики. Ее представление в виде одномерного массива называется линейной граф-схемой. Таким образом, линейная граф-схема - это представление КСР-грамматики в виде последовательности символов-команд " \downarrow , $<$, $*$, $.$ ", адресов перехода и терминалов.

Состояние - элемент одного из следующих видов:

- a. $\{a - n\}$, где a - терминал, n - номер правила
- b. $\{F\}$ - конечное состояние
- c. $\{s_1, s_2, s_3, \dots s_n\}$, где s_i - состояния
- d. $(s, N - n)$, где s - состояние, N - нетерминал, n - номер правила

5 КСР-грамматика

КС грамматика в регулярной форме - это КС грамматика, в которой правые части правил представлены в виде обобщенно-регулярных выражений над символами объединенного алфавита грамматики.

Изначально грамматика представлена в виде последовательности лексем, однако для программы такое представление неудобно, поэтому первый шаг программы - кодирование КСР грамматики. Программа считывает правила (каждое из которых начинается с нетерминала и двоеточия, а оканчивается точкой) до тех пор, пока не встретит "Eogram". Затем каждое правило разбивается на отдельные лексемы - нетерминалы и терминалы. Каждой лексеме присваивается уникальный номер(код) и она добавляется в кодировочную таблицу. Последним действием все лексемы заменяются на свои коды.

6 Граф-схема

Второй шаг - построение граф-схемы. Для этого каждое из каждого закодированного правила извлекается нетерминал и соответствующее ему регулярное выражение. По последнему строится(1) линейное представление его синтаксической граф-схемы. Построение проходит по следующим правилам:

a	->	a
(A)	->	A
A ; B	->	< n A ↓ m B
[A]	->	< n A
A*	->	< n A ↓ m
A#B	->	A (B;A)*
(A#)	->	A*
(#A)	->	A A
(;A)	->	< n A
(A;)	->	(;A)
N	->	* N

Здесь a - произвольный терминал (его закодированное представление), N - нетерминал (его код), A, B - регулярные выражения. m и n - адреса переходов.

Полученная последовательность записывается в виде массива целых чисел (символы "↓, <, *, ." также имеют кодовое представление) и сопоставляется с соответствующим нетерминалом. Затем к массиву соответствующему начальному нетерминалу "приписываются" все остальные, при этом для каждого нетерминала N^i запоминается номер первого элемента соответствующего ему массива N_{pos}^i в полученном массиве (2). Для того, чтобы полученный массив стал линейной граф-схемой, осталось заменить все коды нетерминалов N^i на номера N_{pos}^i (3).

Пример (для наглядности, без кодировки):
 Firts : 'Hello', Second, ('!'; '!!!!').
 Second : 'World'.
 Eogram.

После (1):

	0	1	2	3	4	5	6	7	8	9
First	'Hello'	*	Second	<	8	'!'	↓	9	'!!!!'	.
Second	'World'	.								

После (2):

0	1	2	3	4	5	6	7	8	9	10	11
'Hello'	*	Second	<	8	'!'	↓	9	'!!!!'	.	'World'	.

N^i	N_{pos}^i
First	0
Second	10

После (3):

0	1	2	3	4	5	6	7	8	9	10	11
'Hello'	*	10	<	8	'!'	↓	9	'!!!!'	.	'World'	.

7 Определение состояний в граф-схеме

Следующий этап программы - построение состояний.

Состояния делятся на несколько видов:

- a. $\{a - n\}$ - если существует префикс непрочитанной части входной цепочки совпадающий с терминалом a , то можно прочитать этот префикс, перейдя в состояние n
- b. $\{F\}$ - конечное состояние, если цепочка прочитана полностью, а стек пуст, то цепочка принята, если стек не пуст, то вытолкнуть один элемент и использовать его в качестве номера следующего состояния
- c. $\{s_1, s_2, s_3, \dots s_n\}$ - вектор состояний, можно перейти в любое из них
- d. $(s, N - n)$ - можно перейти в состояние s , при этом в стек добавляется номер n

Для построения используется следующая рекурсивная функция:

```
f(i):
    next := i + 1
    if table[next].type == terminal
        return { table[next] - next }
    elif table[next].type == number
        return f(table[next])
    elif table[next].type == *
        return ( f(next) , table[next] - next + 1 )
    elif table[next].type == <
        return { f(next), f(next + 1) }
    elif table[next].type == ↓
        return f(next)
    elif table[next].type == .
        return { F }
```

Данная функция каждой позиции в граф-схеме сопоставляет некоторое состояние, которое определяет, какие цепочки символов могут быть приняты в текущий момент. (Начальное состояние при $i = -1$)

Рассмотрим пример:

A : 'a', B, 'c'.

B : 'b'.

Eogram.

Линейная граф-схема:

0	1	2	3	4	5	6
'a'	*	5	'c'	.	'b'	.

Применение функции f даст следующий результат:

```
f(-1) : { 'a' - 0 }
f(0) : ( { 'b' - 5 }, B - 2 )
f(1) : ( { 'b' - 5 }, B - 2 )
f(2) : { 'c' - 3 }
f(3) : { F }
f(4) : { 'b' - 5 }
```

$f(5) : \{ F \}$

Очевидно, что среди этих состояний есть как эквивалентные, так и недостижимые. Для того чтобы определить эквивалентные состояния, используется хэш-суммы. После удаления дублей и недостижимых состояний, получим:

$S = \{ 'a' - 0 \}$
 $S0 = (\{ 'b' - 5 \}, B - 2)$
 $S2 = \{ 'c' - 5 \}$
 $S5 = \{ F \}$

Для данного примера это минимальное количество состояний.

На данный момент состояния представляют собой цепочку вложенных друг в друга элементов различных классов. При анализе цепочки такое представление не удобно. Поэтому с полученными состояниями проводится еще одно преобразование : с помощью рекурсивного обхода строятся состояния вида $S_i / a = S_k [st_1...st_n]$, где S_i - состояние, a - цепочка символов, S_k - состояние, в которое попадет распознаватель при прочтении цепочки, будучи в состоянии S_i . $st_1...st_n$ - числа которые при этом должны попасть на стек.

Для приведенного выше примера результат будет следующим:

$S / 'a' = S0 []$
 $S0 / 'b' = S5 [2]$
 $S2 / 'c' = S5 []$
 $S5 = F$

Для того, чтобы проверить строку на принадлежность грамматике, начнем проверять префиксы входной цепочки и применять соответствующие переходы. Для примера посмотрим на проверку строки 'abc' для построенных выше правил:

(начальное состояние - S, входная цепочка - 'abc', стек пуст)

$(S, 'abc', []) \xrightarrow{a} (S0, 'bc', []) \xrightarrow{b} (S5, 'c', [2]) \xrightarrow{\text{pop}(\)}$
 $(S2, 'c', []) \xrightarrow{c} (S5, , []) \xrightarrow{\text{pop}(\)} (, , [])$

Значит входная цепочка принадлежит грамматике, поскольку при проверке распознаватель попал в конечное состояние, при котором не осталось непрочитанных символов и стек оказался пустым.

8 Основные классы или функции в программе

Программа написана на языке C++, ниже описаны основные функции и классы программы. (Если для какого-либо метода/функции нет описания, значит он является вспомогательным)

Таблица для кодировки представлена в программе следующим образом:

```
class c_Table
{
    vector<string> nonterms;
    vector<string> terms;
```

```

public:
    enum ct_TYPE {
        NONT = 0,
        TERM = 1
    };

    c_Table();

    int get_int(const string & s);
    string get_string(int i);
    int add(const string & s, ct_TYPE type);
    ct_TYPE getType(int i);
}

```

Элементы в таблице могут быть двух видов : терминалы и нетерминалы, которые хранятся в соответствующих массивах.

Функция *add* позволяет добавить элемент *s* с типом *type* в таблицу и возвращает код, присвоенный этому элементу.

Функция *get_int* по элементу возвращает его код, *get_string* - наоборот.

getType возвращает тип элемента по его коду.

Кодировка проводится с помощью объекта следующего класса:

```

class c_Encoder
{
    string getNext(istream & input);
    string getNextWord(const string & s, int & n);
    vector<int> enc(const string & s);

    c_Table table;
public:
    vector< vector<int> > encoded;

    int getTint(const string & s);
    string getTstr(int i);
    c_Table::ct_TYPE getType(int i);

    void Encode(istream & input);
}

```

Тут *table* - кодировочная таблица, *encoded* - закодированное представление грамматики в виде двумерного массива.

Функция *Encode* производит кодировку. В качестве параметра передается поток ввода, результат работы попадает в *table* и *encoded*.

Линейная граф-схема представлена следующим классом:

```

class lin_Table
{
    class element;

    c_Encoder encoder;
    void recgen(int i, int l, int r);
    void generate(int i);
    string elem_to_str(element e);
    unordered_map<int, int> nonts;

public:
    class element {
    public:
        enum elType {
            NUM = 0,
            TERM = 1,
            NONT = 2,
            ARROW = 3,
            ASTER = 4,
            CASES = 5,
            DOT = 6
        };
        elType t;
        int val;
    };

    vector<element> table;

    int get_Nont(int pos);
    string get_FCTstr(int i);

    lin_Table(istream & input);
    void print(std::ostream & out);
}

```

Здесь *encoder* - закодированная грамматика, *table* - ее линейное представление, являющееся массивом элементов типа *element*. *element* - это класс, у которого есть поля 'значения' (*val*) и 'тип' (*t*). Тип определяет роль элемента в таблице, значение необходимо только для элементов типов *NUM*, *TERM*, *NONT*.

Преобразование в граф-схему выполняется внутри конструктора класса. Для этого необходимо передать поток ввода. При этом задействуются приватные рекурсивные функции класса, а также переменная *nonts*, которая позволяет выполнить преобразование (3) (см. пункт 6 Граф-схема).

print печатает полученный массив в переданный поток вывода.

Следующие классы описывают состояния:


```

class Element_base {
public:
    virtual int getType() = 0;
    virtual void print(std::ostream & out, lin_Table & tbl) = 0;
    virtual long hash(tableType * t) = 0;
    virtual void update(samestates & m, tableType & t, bool * valid) = 0;
};

class Element_final : public Element_base {
public:
    Element_final() {};
    int getType();
    void print(std::ostream & out, lin_Table & tbl);
    long hash(tableType * t);
    void update(samestates & m, tableType & t, bool * valid) {}
};

class Element_simple : public Element_base {
public:
    int term, pos;
    long H;
    Element_simple(int t, int p);
    int getType();
    void print(std::ostream & out, lin_Table & tbl);
    long hash(tableType * t);
};

class Element_vector : public Element_base {
public:
    long H;
    vector<shared_ptr<Element_base>> elements;
    Element_vector(shared_ptr<Element_base> e1, shared_ptr<Element_base> e2);
    int getType();
    void print(std::ostream & out, lin_Table & tbl);
    long hash(tableType * t);
    void update(samestates & m, tableType & t, bool * valid);
};

class Element_call : public Element_base {
public:
    int Nont, retpos;
    shared_ptr<Element_base> element;
    long H;
    Element_call(int n, int p, shared_ptr<Element_base> e);
    int getType();
    void print(std::ostream & out, lin_Table & tbl);
    long hash(tableType * t);
};

```

```

    void update(samestates & m, tableType & t, bool * valid);
};

```

Как видно, тут в точности описаны все 4 типа состояний, причем все они наследуются от одного базового класса *Element_base*. Каждый класс имеет метод для печати а поток - *print*, вычисления хэш-суммы - *hash*, а также обновления - *update* (данный метод необходим при удалении дублей, он заменяет все указатели на элемениы с одинаковой хэш-суммой на один общий). Следует также отметить поле *H*. Это поле используется для того, чтобы не пересчитывать хэш-сумму каждый, что может быть довольно долгим процессом, так как функция *hash* рекурсивно обходит все состояния. После того как хэш посчитан, он сохраняется в *H*, и при следующем вызове *hash* будет просто возвращено это значение.

Определим следующие типы:

```

typedef unordered_map<long, vector<int> > samestates;
typedef map< int, shared_ptr< Element_base > > states_t;
typedef map< int, vector< pair< string, pair<int, vector<int> > > > > simple_states;

```

Где *states_t* - тип состояния, *samestates* - тип словаря, сопоставляющего хеш-сумме множество номеров состояний, *simple_states* - тип состояний для распознавателя.

Следующий класс отвечает за построение состояний:

```

class Analyzer_states{
    shared_ptr<Element_base> getState(lin_Table & tbl, tableType & table, int pos);
    void simplify(int state, shared_ptr<Element_base> e, vector<int> & st, lin_Table
& table);

public:
    states_t States;
    simple_states SS;

    Analyzer_states(lin_Table & tbl);
    void printStates(ostream & out, lin_Table & t);

    simple_states get_states(lin_Table & table);
};

```

Как и в *lin_Table*, поздание состояний происходит в конструкторе. Сначала при помощи *getState* генерируется и минимизируется множество состояний *States*, которое затем функцией *simplify* преобразуется в множество состояний *SS*, которое непосредственно используются в распознавателе. *get_states* возвращает состояния распознавателя, *printStates* - печатает в поток вывода.

Сам распознаватель представлен в виде следующего класса:

```

class Analyzer
{
    simple_states states;
    int rec_analyzer(const string & str, int l, int state, vector<int> & st);

public:

    int analyze(const string & str);
    Analyzer(istream & input);
}

```

В *states* хранятся правила(переходные состояния) распознавателя. Метод *analyze* проверяет строку на принадлежность грамматике(используя рекурсивную функцию *rec_analyzer*). Если строка принята, то возвращается *INT_MAX*, иначе позиция, на которой произошла ошибка.

Таким образом, для того, чтобы создать распознаватель КСР-грамматики из файла *grm.txt*, достаточно написать

```
Analyzer A(ifstream("grm.txt"));
```

Проверить строку на принадлежность можно следующим образом:

```
int res = A.analyze("Hello, world!");
```

9 Результат

Программа реализована на языке C++ с использованием возможностей стандарта C++11. Также были использована библиотека *Windows.h* для взаимодействия с консолью ОС Windows. Вся программа разбита на 5 модулей. Суммарное количество строк исходного кода около 950. Размер исполняемого файла составляет 86Кб.

Сначала спрашивается имя файла с грамматикой. Затем способ ввода(с клавиатуры / из файла).

Программа работает в консольном режиме: строит распознаватель по входной КСР-грамматике, а затем проверит входные цепочки на принадлежность грамматике. При этом, если цепочка не принята, она разбивается на 2 части: префикс, который может являться началом цепочки данного КСР-языка и оставшаяся часть (которую распознаватель не принял). Обе части выводятся в консоль и при этом подсвечиваются различными цветами.

Посмотрим на работу распознавателя на примере грамматики *Yard*:

```

Program : ( ; 'static' ) , 'program' , '<tag>' , '(' , ( 'input' ; 'output' ; '<tag>' ) # ( ',' ) , ')' , ( '#' ; 'Declaration' ) , ';' , CompoundStatement , '.' .
ArraySpecification : 'array' , FormalBounds , FormalComponent ; '<tag>' , ( ; FormalBounds ) , ( ; FormalComponent ) .
CompoundStatement : 'begin' , ( NetExpression ) # ( ';' ) , 'end' ; 'if' , NetExpression , 'then' , ( NetExpression ) # ( ';' ) , ( ; 'else' , ( NetExpression ) # ( ';' ) ) , 'endif' .
Constant : '<number>' ; '<real>' ; '<string>' ; '<char>' .

```

```

ConstantDeclaration : 'const' ,( '<tag>' , '=' , Operand )#( ';' ) .
Declaration : ConstantDeclaration ; VariableDeclaration ; NodeDeclaration .
FormalBounds : '[' , ( ; ( '<tag>' ; Constant , ( ; '..' , Constant ) )#( ',' ) )
, ']' .
FormalComponent : 'of' , Specification .
ForStatement : 'for' , 'all' , '<tag>' , 'do' ,( NetExpression ;
CompoundStatement ) .
NetExpression : ( #( '<operation>' ) , Operand , ( ; PrimitiveResource ) )#(
'<operation>' ) ; Operand , ( ; PrimitiveResource ) , ':=' , NetExpression ;
ForStatement .
NodeDeclaration : 'node' , VariableDeclaration ; ( ; 'static' ) , 'node' , '<tag>' ,
( ; ( '(' , ( '<tag>' )#( ',' ) , ')' ) ) ; CompoundStatement .
Operand : '<tag>' , ( #( SliceConstructor ; '^' ) ) ; '(' , NetExpression , ')' ;
'<tag>' , ( ; '(' , ( '<tag>' )#( ',' ) , ')' ) ; Constant .
PointerSpecification : 'pointer' , ( ; ( ; 'to' ) , '<tag>' ) ) .
PrimitiveResource : '@' , ( 'input' ; 'output' ; '<tag>' ) .
SliceConstructor : '[' , ( SubscriptExpression )#( ',' ) , ']' .
Specification : ( 'bool' ; 'byte' ; 'char' ; 'dword' ; 'int' ; 'longint' ; 'shortint'
; 'word' ; 'bitrow' ; 'double' ; 'pointer' ; 'real' ; 'string' ) , ( ; '*' ,
'<number>' ) ; PointerSpecification ; ArraySpecification .
SubscriptExpression : '<tag>' ; 'all' ; NetExpression ; '(' , ( SubscriptExpression
) , ')' .
VariableDeclaration : ( ( '<tag>' )#( ',' ) , ':' , Specification )#( ';' ) .
Eogram.

```

По данной грамматике программа строит следующую линейную граф-схему:

	0	1	2	3	4	5	6	7	8	9
0	<	3	static	program	<tag>	(<	16	<	13
1	input	↓	14	output	↓	17	<tag>	<	22	,
2	↓	6)	<	30	;	*	118	↓	23
3	;	*	54	.	.	<	44	array	*	133
4	*	155	↓	53	<tag>	<	49	*	133	<
5	53	*	155	.	<	67	begin	*	172	<
6	64	;	↓	57	end	↓	89	if	*	172
7	then	*	172	<	78	;	↓	71	<	88
8	else	*	172	<	88	;	↓	81	endif	.
9	<	105	<	102	<	99	<number>	↓	100	<real>
10	↓	103	<string>	↓	106	<char>	.	const	<tag>	=
11	*	237	<	117	;	↓	108	.	<	130
12	<	126	*	107	↓	128	*	414	↓	132
13	*	208	.		<	153	<	141	<tag>	↓
14	148	*	90	<	148	..	*	90	<	153
15	,	↓	136		.	of	*	310	.	for
16	all	<tag>	do	<	169	*	172	↓	171	*
17	54	.	<	205	<	194	<	181	<operation>	↓
18	176	*	237	<	187	*	287	<	192	<operation>
19	↓	176	↓	203	*	237	<	200	*	287
20	:=	*	172	↓	207	*	159	.	<	234
21	<	217	node	*	414	↓	232	<	220	static
22	node	<tag>	<	232	(<tag>	<	231	,	↓
23	225)	↓	236	*	54	.	<	276	<
24	263	<	257	<tag>	<	255	<	252	*	300
25	↓	253	-	↓	244	↓	261	(*	172
26)	↓	274	<tag>	<	274	(<tag>	<	273
27	,	↓	267)	↓	278	*	90	.	pointer
28	<	286	<	285	to	<tag>	.	@	<	298
29	<	295	input	↓	296	output	↓	299	<tag>	.
30	[*	388	<	308	,	↓	301]	.
31	<	385	<	381	<	374	<	371	<	368
32	<	365	<	362	<	359	<	356	<	353
33	<	350	<	347	<	344	<	341	bool	↓
34	342	byte	↓	345	char	↓	348	dword	↓	351
35	int	↓	354	longint	↓	357	shortint	↓	360	word
36	↓	363	bitrow	↓	366	double	↓	369	pointer	↓
37	372	real	↓	375	string	<	379	*	<number>	↓
38	383	*	279	↓	387	*	35	.	<	404
39	<	400	<	397	<tag>	↓	398	all	↓	402
40	*	172	↓	413	(*	388	<	412	,
41	↓	405)	.	<tag>	<	420	,	↓	414
42	:	*	310	<	428	;	↓	414	.	

После этого строится множество состояний:

```

S_first = { { 'program' - 3 } , { 'static' - 2 } }
S_2 = { 'program' - 3 }
S_3 = { '<tag>' - 4 }
S_4 = { '(' - 20 }
S_16 = { { ')' - 28 } , { ',' - 20 } }
S_20 = { { '<tag>' - 16 } , { 'output' - 16 } , { 'input' - 16 } }
S_25 = ( { ( { ( { { 'if' - 67 } , { 'begin' - 62 } } ,
    CompoundStatement - 427 ) , { 'node' - 220 } , { 'static' - 219 }
    , { 'node' - 212 } } , NodeDeclaration - 427 ) , ( { '<tag>'
    - 414 } , VariableDeclaration - 427 ) , ( { 'const' - 115 } ,
    ConstantDeclaration - 427 ) } , Declaration - 28 )
S_28 = { { ';' - 30 } , { ';' - 25 } }
S_30 = ( { { 'if' - 67 } , { 'begin' - 62 } } , CompoundStatement
    - 32 )
S_32 = { '.' - 427 }
S_37 = ( { '[' - 133 } , FormalBounds - 39 )
S_39 = ( { 'of' - 155 } , FormalComponent - 427 )
S_44 = { { F } , ( { 'of' - 155 } , FormalComponent - 427 ) , (
    { '[' - 133 } , FormalBounds - 48 ) }
S_48 = { { F } , ( { 'of' - 155 } , FormalComponent - 427 ) }
S_58 = { { 'end' - 427 } , { ';' - 62 } }
S_62 = ( { ( { 'for' - 159 } , ForStatement - 427 ) , ( { ( {
    { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 }
    , { '<number>' - 427 } } , Constant - 427 ) , { '<tag>' - 263 }
    , { '(' - 257 } , { '<tag>' - 253 } } , Operand - 195 ) , (
    { ( { { '<char>' - 427 } , { '<string>' - 427 } , { '<real>'
    - 427 } , { '<number>' - 427 } } , Constant - 427 ) , { '<tag>'
    - 263 } , { '(' - 257 } , { '<tag>' - 253 } } , Operand - 182 )
    , { '<operation>' - 190 } } , NetExpression - 58 )
S_67 = ( { ( { 'for' - 159 } , ForStatement - 427 ) , ( { ( {
    { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 } ,
    { '<number>' - 427 } } , Constant - 427 ) , { '<tag>' - 263 } ,
    { '(' - 257 } , { '<tag>' - 253 } } , Operand - 195 ) , ( { (
    { { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 }
    , { '<number>' - 427 } } , Constant - 427 ) , { '<tag>' - 263 }
    , { '(' - 257 } , { '<tag>' - 253 } } , Operand - 182 ) , {
    '<operation>' - 190 } } , NetExpression - 69 )
S_69 = { 'then' - 76 }
S_72 = { { 'endif' - 427 } , { 'else' - 86 } , { ';' - 76 } }
S_76 = ( { ( { 'for' - 159 } , ForStatement - 427 ) , ( { ( {
    { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 } , {
    '<number>' - 427 } } , Constant - 427 ) , { '<tag>' - 263 } , {
    '(' - 257 } , { '<tag>' - 253 } } , Operand - 195 ) , ( { ( {
    { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 } ,

```

```

    { '<number>' - 427 } } , Constant - 427 ) , { '<tag>' - 263 } , {
    '(' - 257 } , { '<tag>' - 253 } } , Operand - 182 ) , { '<operation>'
    - 190 } } , NetExpression - 72 )
S_82 = { { 'endif' - 427 } , { ';' - 86 } }
S_86 = ( { ( { 'for' - 159 } , ForStatement - 427 ) , ( { ( { {
    '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 } , {
    '<number>' - 427 } } , Constant - 427 ) , { '<tag>' - 263 } , {
    '(' - 257 } , { '<tag>' - 253 } } , Operand - 195 ) , ( { ( {
    { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 } ,
    { '<number>' - 427 } } , Constant - 427 ) , { '<tag>' - 263 } ,
    { '(' - 257 } , { '<tag>' - 253 } } , Operand - 182 ) , { '<operation>'
    - 190 } } , NetExpression - 82 )
S_108 = { '=' - 109 }
S_109 = ( { ( { { '<char>' - 427 } , { '<string>' - 427 } , { '<real>'
    - 427 } , { '<number>' - 427 } } , Constant - 427 ) , { '<tag>' - 263 }
    , { '(' - 257 } , { '<tag>' - 253 } } , Operand - 111 )
S_111 = { { F } , { ';' - 115 } }
S_115 = { '<tag>' - 108 }
S_133 = { { ']' - 427 } , ( { { '<char>' - 427 } , { '<string>' - 427 }
    , { '<real>' - 427 } , { '<number>' - 427 } } , Constant - 142 ) ,
    { '<tag>' - 147 } }
S_142 = { { ']' - 427 } , { ',' - 151 } , { '..' - 145 } }
S_145 = ( { { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 }
    , { '<number>' - 427 } } , Constant - 147 )
S_147 = { { ']' - 427 } , { ',' - 151 } }
S_151 = { ( { { '<char>' - 427 } , { '<string>' - 427 } , { '<real>'
    - 427 } , { '<number>' - 427 } } , Constant - 142 ) , { '<tag>' - 147 } }
S_155 = ( { ( { { '<tag>' - 44 } , { 'array' - 37 } } , ArraySpecification
    - 427 ) , ( { 'pointer' - 279 } , PointerSpecification - 427 ) , { 'string'
    - 374 } , { 'real' - 374 } , { 'pointer' - 374 } , { 'double' - 374 } ,
    { 'bitrow' - 374 } , { 'word' - 374 } , { 'shortint' - 374 } , { 'longint'
    - 374 } , { 'int' - 374 } , { 'dword' - 374 } , { 'char' - 374 } , {
    'byte' - 374 } , { 'bool' - 374 } } , Specification - 427 )
S_159 = { 'all' - 160 }
S_160 = { '<tag>' - 161 }
S_161 = { 'do' - 162 }
S_162 = { ( { { 'if' - 67 } , { 'begin' - 62 } } , CompoundStatement - 427 )
    , ( { ( { 'for' - 159 } , ForStatement - 427 ) , ( { ( { { '<char>'
    - 427 } , { '<string>' - 427 } , { '<real>' - 427 } , { '<number>' - 427 }
    } , Constant - 427 ) , { '<tag>' - 263 } , { '(' - 257 } , { '<tag>' - 253 }
    } , Operand - 195 ) , ( { ( { { '<char>' - 427 } , { '<string>' - 427 }
    , { '<real>' - 427 } , { '<number>' - 427 } } , Constant - 427 ) , { '<tag>'
    - 263 } , { '(' - 257 } , { '<tag>' - 253 } } , Operand - 182 ) , {
    '<operation>' - 190 } } , NetExpression - 427 ) }
S_182 = { { F } , { '<operation>' - 190 } , ( { '@' - 287 } ,
    PrimitiveResource - 186 ) }
S_186 = { { F } , { '<operation>' - 190 } }

```

S_190 = { ({ ({ { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 } , { '<number>' - 427 } } , Constant - 427) , { '<tag>' - 263 } , { '(' - 257 } , { '<tag>' - 253 } } , Operand - 182) , { '<operation>' - 190 } }
S_195 = { { ':' - 200 } , ({ '@' - 287 } , PrimitiveResource - 199) }
S_199 = { ':' - 200 }
S_200 = ({ ({ 'for' - 159 } , ForStatement - 427) , ({ ({ { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 } , { '<number>' - 427 } } , Constant - 427) , { '<tag>' - 263 } , { '(' - 257 } , { '<tag>' - 253 } } , Operand - 195) , ({ ({ { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 } , { '<number>' - 427 } } , Constant - 427) , { '<tag>' - 263 } , { '(' - 257 } , { '<tag>' - 253 } } , Operand - 182) , { '<operation>' - 190 } } , NetExpression - 427)
S_212 = ({ '<tag>' - 414 } , VariableDeclaration - 427)
S_219 = { 'node' - 220 }
S_220 = { '<tag>' - 221 }
S_221 = { { F } , { '(' - 229 } }
S_225 = { { ')' - 427 } , { ',' - 229 } }
S_229 = { '<tag>' - 225 }
S_253 = { { F } , { '^' - 253 } , ({ '[' - 306 } , SliceConstructor - 253) }
S_257 = ({ ({ 'for' - 159 } , ForStatement - 427) , ({ ({ { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 } , { '<number>' - 427 } } , Constant - 427) , { '<tag>' - 263 } , { '(' - 257 } , { '<tag>' - 253 } } , Operand - 195) , ({ ({ { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 } , { '<number>' - 427 } } , Constant - 427) , { '<tag>' - 263 } , { '(' - 257 } , { '<tag>' - 253 } } , Operand - 182) , { '<operation>' - 190 } } , NetExpression - 259)
S_259 = { ')' - 427 }
S_263 = { { F } , { '(' - 271 } }
S_267 = { { ')' - 427 } , { ',' - 271 } }
S_271 = { '<tag>' - 267 }
S_279 = { { F } , { '<tag>' - 427 } , { 'to' - 284 } }
S_284 = { '<tag>' - 427 }
S_287 = { { '<tag>' - 427 } , { 'output' - 427 } , { 'input' - 427 } }
S_302 = { { ']' - 427 } , { ',' - 306 } }
S_306 = ({ { '(' - 410 } , ({ ({ 'for' - 159 } , ForStatement - 427) , ({ ({ { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 } , { '<number>' - 427 } } , Constant - 427) , { '<tag>' - 263 } , { '(' - 257 } , { '<tag>' - 253 } } , Operand - 195) , ({ ({ { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 } , { '<number>' - 427 } } , Constant - 427) , { '<tag>' - 263 } , { '(' - 257 } , { '<tag>' - 253 } } , Operand - 182) , { '<operation>' - 190 } } , NetExpression - 427) , { 'all' - 427 } , { '<tag>' - 427 } } , SubscriptExpression - 302)
S_374 = { { F } , { '*' - 377 } }
S_377 = { '<number>' - 427 }
S_406 = { { ')' - 427 } , { ',' - 410 } }
S_410 = ({ { '(' - 410 } , ({ ({ 'for' - 159 } , ForStatement - 427) , ({ ({ { '<char>' - 427 } , { '<string>' - 427 } , { '<real>' - 427 }


```

, { '<number>' - 427 } } , Constant - 427 ) , { '<tag>' - 263 } , { '(' - 257 }
, { '<tag>' - 253 } } , Operand - 195 ) , ( { ( { { '<char>' - 427 }
, { '<string>' - 427 } , { '<real>' - 427 } , { '<number>' - 427 } }
, Constant - 427 ) , { '<tag>' - 263 } , { '(' - 257 } , { '<tag>' - 253 }
} , Operand - 182 ) , { '<operation>' - 190 } } , NetExpression - 427 )
, { 'all' - 427 } , { '<tag>' - 427 } } , SubscriptExpression - 406 )
S_414 = { { ':' - 420 } , { ',' - 426 } }
S_420 = ( { ( { { '<tag>' - 44 } , { 'array' - 37 } } , ArraySpecification
- 427 ) , ( { 'pointer' - 279 } , PointerSpecification - 427 ) , { 'string'
- 374 } , { 'real' - 374 } , { 'pointer' - 374 } , { 'double' - 374 } ,
{ 'bitrow' - 374 } , { 'word' - 374 } , { 'shortint' - 374 } , { 'longint'
- 374 } , { 'int' - 374 } , { 'dword' - 374 } , { 'char' - 374 } , {
'byte' - 374 } , { 'bool' - 374 } } , Specification - 422 )
S_422 = { { F } , { ';' - 426 } }
S_426 = { '<tag>' - 414 }
S_427 = { F }

```

Всего получилось 68 состояний (изначально - 427, минимизация уменьшила это количество на 84%).

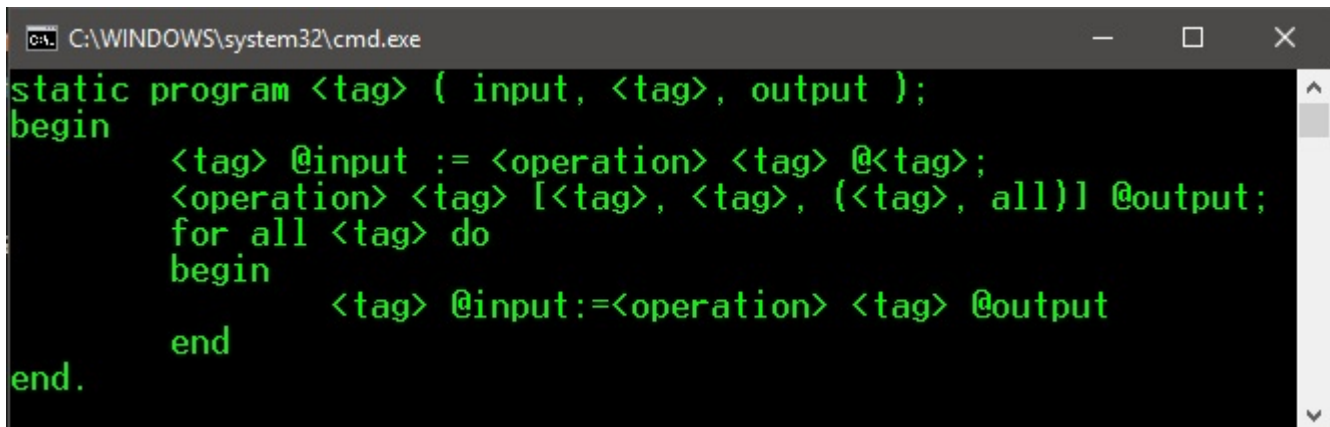
Проверим построенный распознаватель на следующем примере:

```

static program <tag> ( input, <tag>, output );
begin
    <tag> @input := <operation> <tag> @<tag>;
    <operation> <tag> [<tag>, <tag>, (<tag>, all)] @output;
    for all <tag> do
    begin
        <tag> @input:=<operation> <tag> @output
    end
end.

```

Результат работы программы следующий:



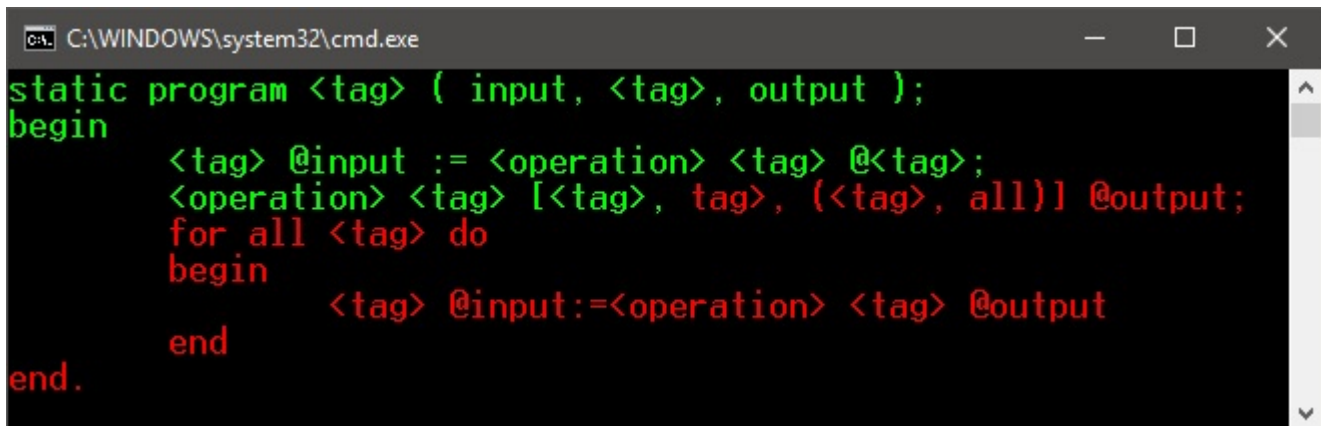
```

C:\WINDOWS\system32\cmd.exe
static program <tag> ( input, <tag>, output );
begin
    <tag> @input := <operation> <tag> @<tag>;
    <operation> <tag> [<tag>, <tag>, (<tag>, all)] @output;
    for all <tag> do
    begin
        <tag> @input:=<operation> <tag> @output
    end
end.

```

Зеленый цвет означает то, что данная последовательность символов принадлежит грамматике.

Теперь внесем в данный тест ошибку (например, удалим "<" из середины цепочки). Результат будет следующим:

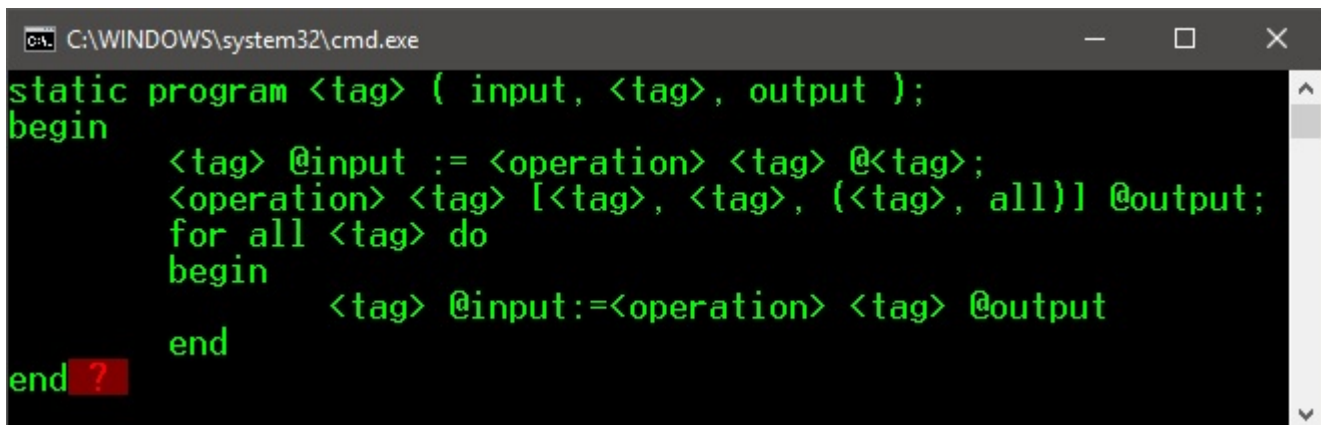


```
C:\WINDOWS\system32\cmd.exe

static program <tag> ( input, <tag>, output );
begin
    <tag> @input := <operation> <tag> @<tag>;
    <operation> <tag> [<tag>, tag>, (<tag>, all)] @output;
    for all <tag> do
    begin
        <tag> @input:=<operation> <tag> @output
    end
end.
```

Теперь половина цепочки подсвечена красным. Это значит, что на этой позиции распознаватель не смог перейти в какое-либо состояние, то есть зеленая часть является корректной, а в красной есть ошибка.

Рассмотрим еще один вариант ошибки: что если распознаватель прочел всю цепочку, но так и не попал в конечное состояние? Для этого удалим завершающий символ "." из первого тестового примера:



```
C:\WINDOWS\system32\cmd.exe

static program <tag> ( input, <tag>, output );
begin
    <tag> @input := <operation> <tag> @<tag>;
    <operation> <tag> [<tag>, <tag>, (<tag>, all)] @output;
    for all <tag> do
    begin
        <tag> @input:=<operation> <tag> @output
    end
end ?
```

Вся входная цепочка зеленая, однако в конце вывода присутствует красный вопросительный знак, который сигнализирует то, что цепочка закончилась, но конечное состояние достигнуто не было (т.е. возможно ввод не был прерван).