# ИНФОРМАЦИОННЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ

УДК 681.51
DOI: 10.18101/2304-5728-2018-1-66-82

## ПОСТРОЕНИЕ РАСПОЗНАВАТЕЛЯ ЯЗЫКА YARD ПО СИНТАКСИЧЕСКОЙ ГРАФ-СХЕМЕ

**© Федорченко Людмила Николаевна**
кандидат технических наук, доцент
старший научный сотрудник,
Санкт-Петербургский институт информатики и автоматизации Российской академии наук
Россия, 199178, г. Санкт-Петербург, 14 Линия В.О., 39
E-mail:LNF@iias.spb.su

В статье представлена схема автоматического построения распознавателей языков класса LL(1) с использованием синтаксических граф-схем. В качестве примера рассмотрен формальный язык Yard, моделирующий поведение многослойных искусственных нейронных сетей на принципах машины с динамической архитектурой и в силу этого имеющий ряд специфических языковых конструкций.

***Ключевые слова***: синтаксическая граф-схема; КСР-грамматика; распознаватель формального языка.

### Введение

В настоящее время языковые технологии активно включаются в различные сферы нашей жизни, например, в комплексные системы производственной сферы [1], при сопоставлении образцов в анализе ДНК, в задачи компьютерной безопасности [2] и во многие другие, что привело к развитию современных транслирующих систем, использующих принцип синтаксического управления для производства программ обработки данных и ориентированных на разнообразный набор вычислительных устройств.

Обработка данных по такой технологии управляется синтаксической структурой этих данных, которые традиционно описываются формальными грамматиками и чаще всего контекстно-свободными грамматиками и их обобщениями.

Множество современных искусственных языков, языки программирования (Си, C++, языки семейства Microsoft.Net (C#, F#, Visual C++, Visual C#, Visual F#), Python2, Python3, Java, Haskel, Rust, Kotlin, Go, Perl, PHP, R, JavaScript, Swift, Objective-C, Pascal, Basic, D и другие), языки разметки графов (DOT, Trivial Graph Format, GraphML и другие) порождаются контекстно-свободными грамматиками в регулярной форме (КСР-грамматиками). Возникает проблема синтаксического анализа, разбора подобных языков и построения распознавателей для эффективного опре-

деления принадлежности цепочек (предложений) языку.

В статье рассматривается задача построения распознавателя для произвольной КСР-грамматики на примере формальной грамматики языка программирования Yard с определенными функциями, реализация которого предполагает в дальнейшем создание оптимального генератора кода для специальных нестандартных чипов.

## 1. Современное состояние дел

Для решения задачи построения трансляции используются различные технологии разработки трансляторов, наиболее распространенными из которых являются средства, основанные на применении различных подклассов контекстно-свободных (КС) грамматик для описания входного языка [3–6]:

- YACC (Yet Another Compiler-Compiler) — 1975 — это a parser generator: находит иерархическую структуру программы.
- Lex — 1978 — это a lexical analyzer generator: разбивает исходный файл на лексемы или другие лексические единицы.
- Lex &YACC — 366 страниц, 2-е издание. (1992).
- Flex (Fast Lexical Analyzer), (1987) — генератор сканера, release 2.6.4, May 6, 2017 — TeX manual (287K).
- GNU Bison (1985), YACC (Yet Another Compiler Compilers) — совместимый генератор анализатора — ASCII (548K).
- ANTLR (ANother Tool for Language Recognition). — Руководство по ANTLR 4–328 страниц. 2nd ed. 2013.

Очевидно, что для построения современных трансляторов использование стандартных инструментов является утомительным и трудоемким делом.

В качестве перспективной формальной базы для построения трансляторов в СПИИРАН разработана методика, основанная на использовании грамматик с обобщенными регулярными выражениями и атрибутами в виде семантик для представления контекстных ограничений и синтаксических граф-схем. Эта методика поддержана пакетом программных приложений, реализованных на платформе Microsoft.NET. В качестве основы взят код инструментального средства SynGT (SynGT Graph Transformation), разрабатываемого в СПИИРАН с 2000 г. [8].

- *Формальной моделью* методики построения языковых процессоров, описанной в [7] и реализованной в SynGT, являются регулярные выражения как средство описания языка и конечный автомат в качестве адекватного средства его обработки.
- *Теоретическим обоснованием* этой модели являются теорема С. Клини и ее следствия о том, что класс регулярных множеств является минимальным классом, содержащим все конечные множества, замкнутым относительно операций объединения, конкатенации и замыкания [9].

• *Регулярные множества* распознаются конечными автоматами и представляются регулярными выражениями.

Использование этой модели ограничено ее применением лишь к регулярным языкам, поэтому модель обобщена до класса КСР-языков с использованием специальных магазинных автоматов [10; 8].

КСР-язык порождается КСР-грамматикой — обобщением КС-грамматики.

Основные определения были даны в работах [8; 11]. Напомним их неформально.

Как известно [9], формальным языком является набор цепочек языковых токенов (лексем или терминалов) в соответствии с грамматикой языка. Грамматика языка состоит из контекстно-свободной части в виде КС-правил и контекстно-зависимой части (ограничений).

КС-правила представлены в форме с обобщенными регулярными выражениями (КСР-правила):

*Нетерминал* : *обобщенное регулярное выражение*.

Регулярные выражения состоят из токенов (лексем), нетерминалов, регулярных операций (конкатенация, альтернативный выбор и обобщенная итерация), пустой цепочки ("") и скобок ("(" и ")"):

– Конкатенация (“,”): A,B = AB.
– Альтернативный выбор (“;”): A;B = либо A, либо B.
– Обобщенная итерация ( “#“): A#B = A; ABA; ABABA; ABABABA;
– Унарная итерация Клини (“*“): A* =…; A; AA; AAA;…
– Возможно пустое множество цепочек X: [ X ] = ( X; ).

## 2. Обобщение регулярной модели

Обобщение конечно автоматной модели обработки языков сводится к следующему:

• вводится итерация с разделителем (#) или обобщенная итерация, или итерация Г. С. Цейтина, которая не расширяет множество регулярных слов и может быть определена через традиционную (одноместную) операцию Клина (*) как (P#Q)=P,(Q,P)*. Она удобна при работе со стеком и частично решает задачу минимизации регулярного выражения по числу вхождений символов из объединенного алфавита всех символов грамматики;

• КСР-грамматика — обобщение КС-грамматики. Класс языков не расширяет, но снимает лишнюю структурированность языка;

• синтаксическая граф-схема (СГС) — графический аналог КСР-грамматики, стартовый объект для синтеза распознавателя (анализатора) языка, в терминах вершин которой удобно формулировать свойство детерминированности языка и возможные типы конфликтов (Shift/Reduce). СГС является промежуточной моделью между порождающей структурой, грамматикой и распознающей машиной, МП-автоматом.

## 3. Синтаксическая граф-схема (СГС)

*Синтаксические граф-схемы* (СГС) — это аналог диаграмм Вирта. Каждому нетерминалу соответствует одна компонента такой диаграммы или одно правило КСР-грамматики, правая часть которого является обобщенным регулярным выражением относительно символов объединенного алфавита грамматики. В объединенный алфавит грамматики входят алфавиты терминалов, нетерминалов, семантик и возможно предикатов.

СГС создается рекурсивно из графов для элементарных регулярных выражений и бинарных операций (рис. 1).

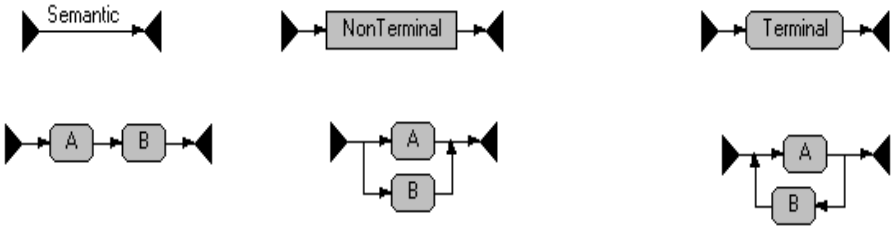


Рис.1. Графы для элементарных регулярных выражений

При разработке транслятора необходимо решать следующие *задачи:*

- *упрощение грамматики;*
- *регуляризация грамматики;*
- *разрешение конфликтных ситуаций,* возникающих в процессе построения распознавателя и анализатора и связанных с языковой неоднозначностью (как синтаксической, так и семантической) и недетерминированностью магазинного автомата.

*Фундаментальные работы* таких авторов, как:

- Н. Хомский — иерархия грамматик;
- С. К. Клини — регулярные множества и регулярные выражения;
- Г. С. Цейтин, А. П. Ершов, С. С. Лавров, Б. К. Мартыненко [10] — технология программирования;
- Д. Кнут, А. В. Ахо, Д. Ульман — теория языков и автоматов;
- Грегори Розенберг и Арто Саломаа — Handbook of Formal Languages составляют основу современной теории построения трансляторов языка, которую необходимо расширять в соответствии с требованиями практики во всех областях применения наличием инструментария и методов для автоматизации реализуемого языка в условиях жестких временных ограничений.

## 4. Инструментальная система SynGT

Основные модули:

- текстовый редактор:
  - проверяет корректность регулярных выражений (конечно-автоматная модель);

– формирует приведенную (well-formed) грамматику (удаление недостижимых нетерминалов, нетерминалов с пустым порождением, цепные правила и т.д.);
– осуществляет редактирование словарей символов и др.);
• графический редактор:
– осуществляет пользовательский интерфейс для обработки грамматики;
• эквивалентные синтаксические преобразования:
– формируют грамматику для построения детерминированного процессора;
• генератор тестов:
– генерирует набор прототестов для получения их замыкания [12].

## 5. SynGT: базовые функции и решаемые задачи

Алгоритмы, реализованные и постоянно пополняемые в SynGT, выполняют следующие эквивалентные преобразования над регулярными выражениями и КСР-правилами:
– подстановка вместо нетерминала его порождения;
– исключения лево- и праворекурсивных нетерминалов из грамматики с обобщенной итерацией;
– объединение общих префиксов в графе для нетерминала;
– удаление повторяющихся альтернатив для нетерминала;
– удаление крайних рекурсий для самовложенных нетерминалов;
– свертка регулярного подвыражения в новый нетерминал и создание правила в грамматике для нового нетерминала с регулярным подвыражением в его правой части;
– удаление лишних правил.

В результате должны быть решены следующие задачи:

1. После обработки текста КСР-грамматики получаем правильную запись регулярных выражений. Все входящие нетерминалы являются продуктивными, т. е. порождают непустые цепочки, и достижимыми, то есть участвуют в выводе цепочек языка.

2. Преобразование текстового представления регулярного выражения в графическое (задача размещения граф-схемы на экране).

3. Реализация стандартных редакторских функций и функции подстановки вместо нетерминала его регулярного выражения, а также сведения подвыражения в новый нетерминал и добавление его в список нетерминалов грамматики.

4. Обратное преобразование граф-схемы в текстовое регулярное выражение.

5. Эквивалентные преобразования КСР-грамматики и синтаксической граф-схемы (удаление рекурсий).

6. Регуляризация КСР-грамматики.

7. Синтез (построение) конечной таблицы всех состояний МП-автомата, распознающего формальный язык.

8. Тестирование. Построение списка прототестов и их замыкание для получения множества «хороших» и «плохих» тестов [12].

Набор функций, реализованных в SynGT, служит основной цели — автоматизированное преобразование КС-грамматики для ее упрощения и максимальной регуляризации, то есть превращение синтаксической граф-схемы грамматики в минимальный набор конечных автоматов для построения анализатора языка. Если КС-грамматика порождает регулярный язык, то синтаксическая граф-схема вырождается в один граф, представляющий эквивалентное регулярное выражение. Алгоритм и примеры даны в работе [8].

## 6. Yard — пример формального языка

Рассмотрим технику построения распознавателя формального языка Yard с использованием пакета модулей SynGT.

Язык Yard содержит:

**54 лексемы:**

- generics: '<char>' '<number>' '<operation>' '<real>' '<string>' '<tag>';
- зарезервированные слова: 'all' 'array' 'begin' 'bitrow' 'bool' 'byte' 'char' 'const' 'do' 'double' 'dword' 'else' 'end' 'endif' 'for' 'if' 'in' 'inout' 'input' 'int' 'longint' 'node' 'of' 'out' 'output' 'pointer' 'program' 'real' 'shortint' 'static' 'string' 'then' 'to' 'word';
- индиканты: '(' ')' '*' ',' '.' '..' ':' ':=' ';' '@' '[' ']' '^' '=';

**19 нетерминалов**: ArraySpecification CompoundStatement Constant ConstantDeclaration Declaration FormalBounds FormalComponent ForStatement LinkList NetExpression NodeDeclaration Operand PointerSpecification PrimitiveResource Program SliceConstructor Specification SubscriptExpression VariableDeclaration.

Начальный нетерминал: Program.

**38 (semanitcs**), разбросанных по грамматическим правилам. Каждой семантике предшествует знак «$»

## 7. Контекстно-свободные правила в регулярной форме

**1.** ArraySpecification : 'array' FormalBounds FormalComponent ; '<tag>' [ FormalBounds ] [ FormalComponent ] .

**2.** Constant : '<number>' ; '<real>' ; '<string>' ; '<char>' .

**3.** CompoundStatement : 'begin' ( NetExpression )*( ';' ) 'end' ; 'if' NetExpression 'then' ( NetExpression )*( ';' ) [ 'else' ( NetExpression )*( ';' ) ] 'endif' .

**4.** ConstantDeclaration : 'const' ( '<tag>' '=' Operand )*( ';' ) .

**5.** Declaration : ConstantDeclaration ; VariableDeclaration ; NodeDeclaration .

**6.** FormalBounds : '[' [ ( '<tag>' ; Constant [ '..' Constant ] )*( ',' ) ] ']' .

**7.** FormalComponent : 'of' Specification .

**8.** ForStatement : 'for' 'all' '<tag>' 'do' ( NetExpression ; CompoundStatement

) .

**9.** LinkList : ( ( 'in' ; 'out' ; 'inout' ) ( ( '<tag>' )*( ',' ) ':' Specification )*( ';' ) )*( ';' ) .

**10.** NetExpression : ( *( '<operation>' ) Operand [ PrimitiveResource ] )*( '<operation>' ; '=' ) ; Operand [ PrimitiveResource ] ':=' NetExpression ; ForStatement .

**11.** NodeDeclaration : 'node' VariableDeclaration ; [ 'static' ] 'node' '<tag>' [ '(' ( '<tag>' )*( ',' ) ')' ] ; CompoundStatement .

**12.** Operand : '<tag>' *( SliceConstructor ; '^' ) ; '(' NetExpression ')' ; '<tag>' [ '(' ( '<tag>' )*( ',' ) ')' ] ; Constant .

**13.** PointerSpecification : 'pointer' [ [ 'to' ] '<tag>' ] .

**14.** PrimitiveResource : '@' ( 'input' ; 'output' ; '<tag>' ) .

**15.** Program : [ 'static' ] 'program' '<tag>' '(' ( 'input' ; 'output' ; '<tag>' )*( ',' ) ')' *( ';' Declaration ) ';' CompoundStatement '.' .

**16.** SliceConstructor : '[' ( SubscriptExpression )*( ',' ) ']' .

**17.** Specification : ( 'bool' ; 'byte' ; 'char' ; 'dword' ; 'int' ; 'longint' ; 'shortint' ; 'word' ; 'bitrow' ; 'double' ; 'pointer' ; 'real' ; 'string' ) [ '*' '<number>' ] ; PointerSpecification ; ArraySpecification .

**18.** SubscriptExpression : '<tag>' ; 'all' ; NetExpression ; '(' ( SubscriptExpression )*( ',' ) ')' .

**19.** VariableDeclaration : ( ( '<tag>' )*( ',' ) ':' Specification )*( ';' ) .

## 8. Контекстные условия (семантики)

**38 semantics:** $array1 $array2 $bound1 $bound2 $constant $constdecl $do $dyadic $else $endif $forall $if $link $monadic $netexpr $node $operand $operand1 $operand2 $pointer1 $pointer2 $primitive1 $primitive2 $program1 $program2 $program3 $slice1 $slice2 $slice3 $spec1 $spec2 $static $subscript1 $subscript2 $then $up $vardecl1 $vardecl2 расставлены внутри правил, например, семантика $vardecl1 проверяет, что каждая переменная описана в программе.

Семантики могут вставляться в текст правила для нетерминала либо непосредственно на дугах в граф-схеме.

Например, проверка на единственность описания каждого тэга.

VariableDeclaration : ( ($vardecl1 '<tag>' )*( ',' ) ':' Specification $vardecl2)*( ';' ) .

Operand : $operand1 ( '<tag>' *( SliceConstructor ; '^' ) ; '(' NetExpression ')' ; '<tag>' [ '(' ( '<tag>' )*( ',' ) ')' ] ; Constant ) $operand2 .

## 9. Проверка грамматики Yard в SynGT

Первоначально с помощью SynGT проверяется корректность записи регулярных выражений, удаляются лишние правила (например, нетерминал LinkList оказался недостижимым) (рис. 2–4).

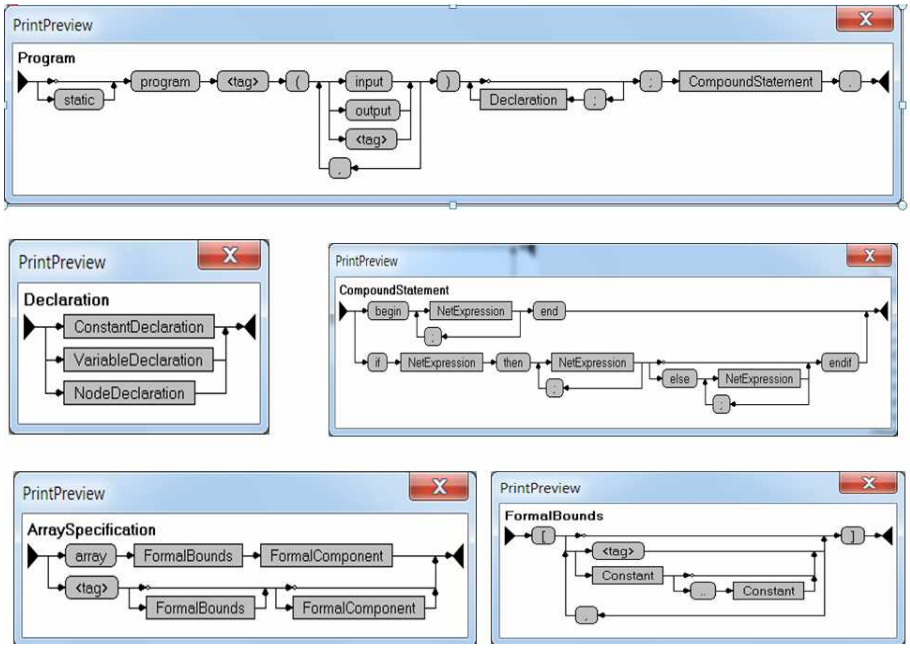


Рис. 2. SynGT-1. Графы для нетерминалов грамматики Yard

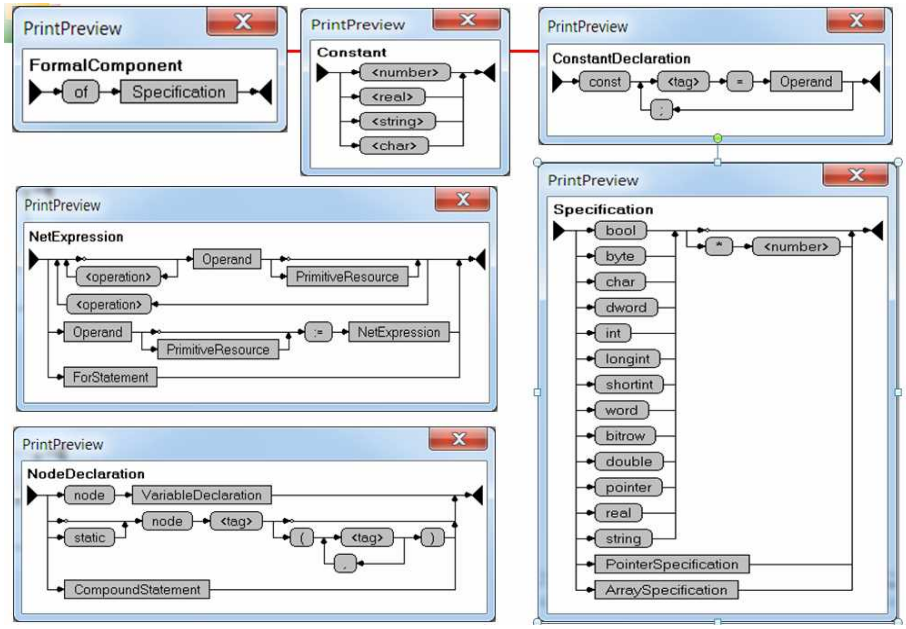


Рис. 3. SynGT-2. Графы для нетерминалов грамматики Yard

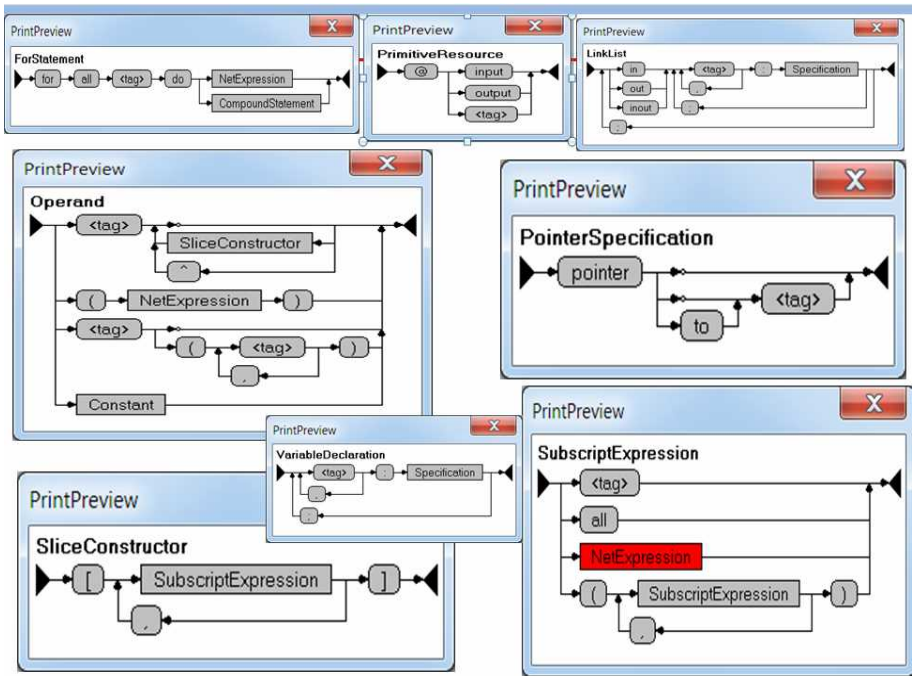


Рис. 4. SynGT-3. Графы для нетерминалов грамматики Yard

Далее проводится анализ на рекурсивности (рис. 5), удаляются леворекурсивные нетерминалы грамматики, грамматика тестируется на ее принадлежность к классу $LL(k)$. Если $k > 1$, то выполняется эквивалентное преобразование для приведения грамматики в класс $LL(1)$. Затем если необходимо, то выполняется ее регуляризация (сведение к минимальному числу правил) (рис. 5).

| Nonterminal | Value | Left recurs | Recursion | Right Rec |
|---|---|---|---|---|
| Program | ('','static'),'program','<tag>',('(','input','output','<tag>')#',')','#(',',Declaration),' | | | |
| Declaration | ConstantDeclaration;VariableDeclaration;NodeDeclaration. | | | |
| CompoundStatement | 'begin',NetExpression#',','end','if',NetExpression,'then',NetExpression#',',('; | | indirect | indirect |
| ArraySpecification | 'array',FormalBounds,FormalComponent,'<tag>',('',FormalBounds),('',Form | | indirect | indirect |
| FormalBounds | '[',('',('<tag>',Constant,('','..',Constant))#','),']'. | | | |
| FormalComponent | 'of',Specification. | | indirect | indirect |
| NetExpression | ('#'<operation>',Operand,('',PrimitiveResource))#'<operation>',Operand,('', | indirect | direct | direct |
| Constant | '<number>','<real>','<string>','<char>'. | | | |
| ConstantDeclaration | 'const',('<tag>','=',Operand)#';'. | | | |
| Operand | '<tag>','#(SliceConstructor,'^');'(',NetExpression,')','<tag>',('',('(','<tag>'#',')'); | indirect | indirect | indirect |
| VariableDeclaration | ('<tag>'#',',':',Specification)#';'. | | | |
| NodeDeclaration | 'node',VariableDeclaration;('','static'),'node','<tag>',('',('(','<tag>'#',')');Comp | | | |
| Specification | ('bool','byte','char','dword','int','longint','shortint','word','bitrow','double','point | | indirect | indirect |
| ForStatement | 'for','all','<tag>','do',(NetExpression;CompoundStatement). | | indirect | indirect |
| LinkList | (('in','out','inout'),('<tag>'#',',':',Specification)#',')#';'. | | | |
| PrimitiveResource | '@',('input','output','<tag>'). | | | |
| SliceConstructor | '[',SubscriptExpression#',',']'. | | indirect | |
| PointerSpecification | 'pointer',('',('','to'),'<tag>'). | | | |
| SubscriptExpression | '<tag>','all',NetExpression;'(',SubscriptExpression#',',')'. | direct | direct | |
| EOGram | ''. | | | |

Рис. 5. Анализ на рекурсивность в SynGT

## 10.Реализация алгоритма построения распознавателя языка Yard

Формальные определения состояний в граф-схеме и основные алгоритмы построения распознавателя формального языка даны в [11]. В данном разделе кратко опишем реализацию алгоритмов построения распознавателя, выполненную студентами 3-го курса СПбГУ *Кравченко Евгением* и *Ерзиковой Юлией*.

Алгоритм построения распознавателя языка Yard состоит из трех основных этапов:

**Этап 1.** Приведенную грамматику Yard, полученную из SynGT, преобразовать в граф-схему в линейном представлении согласно заданным шаблонам [11].

**Этап 2.** Построить стек состояний, удалив эквивалентные состояния.

**Этап 3.** Построить программу-распознаватель и провести несколько экспериментов.

Для построения синтаксической граф-схемы были использованы следующие программные элементы, которые опишем далее.

С помощью перечисления зададим всевозможные типы элементов, содержащихся в строке: символьная строка, операция, адрес перехода и выход из правила.

Класс **TableElement** описывает элемент линейного представления граф-схемы. Для данного элемента будем хранить его тип и само значение элемента. При этом если это адрес перехода, то для него поле lexeme является пустым, а поле numb заполняется числовым значением. Для остальных типов элементов поле numb = -1, а lexeme инициализируется соответствующей строкой.

```
enum element_type {String, Operator, Address_link, Ex};

class TableElement
{
   element_type kind;
   string lexeme;
   int numb;
  public:
   TableElement (element_type x, string y="", int z=-1):
kind(x), lexeme(y), numb(z) {}
}
```

Класс LinearGraphScheme, являющийся дружественным с классом TableElement, ответственен за построение итоговой граф-схемы в линейной форме по КСР-грамматике. Основным методом является FormResultTable, который организует обработку правил входной КСР-грамматики по принципу, описанному в разделе «Описание структуры программы и представление данных в ней».

Шаг 1. В ходе анализа каждого КСР-правила для него вход по нетерминалу и разветвление путей заносится в очередь. При достижении конца правила достаем из очереди первый элемент и для него выполняем рекурсивный вызов метода **FormResultTable**. Следующие два метода используются для определения принадлежности текущей лексемы к

определенному типу(element_type) и развертки операции по заданным шаблонам.

```
class LinearGraphScheme
{private:
    vector<string> Input_CFR_Rules;
    vector<TableElement> ResultTable;
   public:
    vector< TableElement> FormResultTable(string current_st,
int len);
    element_type GetElementType(string s);
    string template_operator(string A, string B, Operator C)
    TableElement GetValue(int TableIndex);
    element_type GetType(int TableIndex);
    TableElement Start_of_Rule(string nonterm);
    TableElement End_of_Rule(string nonterm);
    friend class TableElement;
}
```

По заданным индексам может быть найдено значение и тип элемента в массиве, также по нетерминалу можно восстановить элементы начала и конца, задаваемого правила.

Для построения стека состояний были использованы следующие программные элементы:

Класс **FormStackOfState** отвечает за построение стека состоянии распознавателя КСР-цепочек. Основным методом является **FormStateStack**, который с помощью построенной ранее линейной граф-схемы заполняет stack States по алгоритму, подробно описанному в разделе «Описание структуры программы и представление данных в ней».

Шаг 2. Попутно в вектор **State_Address** помещаются адреса всех состояний из стека. Дополнительно введенные методы **Get_State** и **Get Address** предназначены для сопоставления состояния граф-схемы и адреса элемента в стеке.

Данный класс является фундаментом для осуществления последующей проверки на принадлежность цепочек символов КСР-грамматике.

Для построения последовательности состояний анализатора были использованы следующие программные элементы:

Класс **CheckCharacterStrings** отвечает за распознавание входных цепочек символов.

Конструктор этого класса инициализирует и получает построенный на предыдущем шаге стек состояний. Многомерный контейнер **Trajectory** отвечает за хранение путей в процессе передвижения по состояниям, т. к. таких путей может быть не один, а несколько.

```
class FormStackOfState
{
private:
    stack<int> States;
    vector<int> State_Address;
    LinearGraphScheme grapth_scheme;
public:
    stack<int> FormStateStack(LinearGraphScheme
GrapthScheme);
    int Get_State (int Address);
    int Get_ Address(int State);
}
```

Также класс CheckCharacterStrings обладает методом **Check**, который принимает на вход цепочку символов и осуществляет ее проверку на принадлежность данной КСР-грамматике.

Для того чтобы можно было отследить до какого момента (состояния) рассматриваемая цепочка выводима в заданной КСР-грамматике, был добавлен метод **NextState**, при каждом вызове которого объект класса переходит на одно состояние вперед. В свою очередь, за вывод построенного для цепочки пути отвечает метод **WritePath**.

Эффективность данного способа организации распознавания цепочек достигается за счет того, что построенный стек состояний инициализируется один раз при создании объекта класса и хранится в течение всего периода работы программы. Поэтому проверка входной цепочки вынесена в отдельный метод класса.

```
Class CheckCharacterStrings
{
private:
    FormStackOfState Stack_of_state;
    vector<vector<int>> Trajectory;
    int NumberOfPaths;
    int CurrentState;
public:
    CheckCharacterStrings(FormStackOfState StackOfState);
    bool Check(string CharacterString);
    int NextState();
    void WritePath();
}
```

Реализация граф-схемы в линейной записи основана на рекурсивном сведении более сложных случаев к базисным элементам, которые представлены следующими шаблонами:

Таблица 1

Шаблоны для представления операций в граф-схеме

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1) a,b. | Адрес в массиве | 1 | 2 | 3 | | | | |
| | значение | a | b | . | | | | |
| 2) a;b. | Адрес в массиве | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | значение | < | 6 | a | ↓ | 7 | b | . |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3) a#b. | Адрес в массиве | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | значение | a | < | 7 | b | ↓ | 1 | . |

В результате граф-схема грамматики Yard представлена на рис. 6 в компактном двумерном виде, где самый левый столбец вместе с верхней строкой формирует адрес в массиве — машинном представлении граф-схемы.

| | 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 | . | |

Рис. 6. Линейное представление граф-схемы Yard

```
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 }
```

Рис. 7. Состояния распознавателя

Всего 427 состояний, которые после минимизации автомата свелись к 68 состояниям (уменьшение на 84%).

Проверим построенный распознаватель на следующих двух тестах, показанных в экранном варианте на рисунках 8, 9.

```
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.
```

Рис. 8. Пример входной цепочки

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.
```

Рис. 9. Тест с ошибкой

Текст после ошибки подсвечен красным цветом.

## Заключение

В результате проделанной работы была построена МП-таблица состояний распознавателя языка Yard, которая может быть использована далее при построении анализатора. Цель на будущее — обеспечить конкурентное преимущество для новых чипов, снабдив их компилятором и библиотекой программных решений, разработанных на основе инструментальной системы SynGT.

## Литература

1. Лукьянова Л. М., Федорченко Л. Н. Средства формализации целей и проблем сложных систем производственной сферы // Вестник Бурятского государственного университета. 2012. № 9. С. 42–48.

2. Исследование и выбор криптографических стандартов на основе интеллектуального анализа документов / В. И. Воробьев [и др.] // Труды СПИИРАН. 2016. Вып. 48. С. 69–87.

3. M. E. Lesk, E. Schmidt. Lex – A Lexical Analyzer Generator. URL: http://dinosaur.compilertools.net/lex/ (дата обращения 25.02.2018).

4. Win flex-bison. URL: http://sourceforge.net/projects/winflexbison/ (дата обращения 25.02.2018).

5. GNU Bison. URL: http://www.gnu.org/software/bison/ (дата обращения 25.02.2018).

6. Terence Parr. ANTLR (ANother Tool for Language Recognition). URL: http://www.antlr.org/ (дата обращения 25.02.2018).

7. Fedorchenko L. Regularization of Context-Free Grammars. LAP LAMBERT Academic Publishing, Saarbrucken, 2011. 188 p.

8. Fedorchenko L. and Baranov S. Equivalent Transformations and Regularization in Context–Free Grammars // Bulgarian Academy of Sciences / Cybernetics and Information Technologies (CIT). Sofia, 2015. Vol. 14, No 4, P. 11–28.

9. A. Aho, R. Sethi, J. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986. 796 p.

10. B. K. Martynenko. Regular Languages and CF Grammars // In Computer Tools in Education. 2012. 1. P. 14–20.

11. Федорченко Л. Н. Алгоритмы построения состояний анализатора для КСР-языка // Вестник Бурятского государственного университета. Математика, информатика. 2016. № 4. С. 23–33.

12. Федорченко Л. Н. Генерация тестов в системе SynGT // Вестник Бурятского государственного университета. Математика, информатика. 2017. № 2. С. 33–39.

CONSTRUCTION OF THE YARD LANGUAGE RECOGNIZER USING SYNTAX GRAPH-SCHEME

*Ludmila N. Fedorchenko*
Cand. Sci. (Engineering), A/Prof., Senior Researcher,
St Petersburg Institute for Informatics and Automation of Russian Academy of Sciences (SPIIRAS).
39, 14th Liniya V. O., St Petersburg 199178, Russia
E-mail: lnf@iias.spb.su

The article presents the scheme for automatic construction of language recognizers of LL (1) class using the syntactic graph-schemes. As an example we have considered YARD formal language, modeling the behavior of multilayered artificial neural networks based on the principles of a machine with a dynamic architecture, and therefore having a number of specific language constructs.

*Keywords:* syntactic graph-scheme; context-free array grammar; formal language recognizer.

*References*

1. Luk'yanova L. M., Fedorchenko L. N. Sredstva formalizatsii tselei i problem slozhnykh sistem proizvodstvennoi sfery [Means of Formal Characterization of Objectives and Problems of Complex Production Systems]. *Vestnik Buryatskogo gosudarstvennogo universiteta — Bulletin of Buryat State University.* 2012. No. 9. Pp. 42–48.

2. Vorob'ev V. I., Evnevich E. L., Levonevskii D. K., Fatkieva R. R., Fedorchenko L. N. Issledovanie i vybor kriptograficheskikh standartov na osnove intellektual'nogo analiza dokumentov [Investigation and Selection of Cryptographic Standards Based on Intellectual Analysis of Documents]. *Trudy SPIIRAN — SPIIRAS Proceedings.* 2016. V. 48. Pp. 69–87.

3. Lesk M. E., Schmidt E. *Lex – A Lexical Analyzer Generator.* Available at: http://dinosaur.compilertools.net/lex/ (accessed 25.02.2018).

4. *Win Flex-Bison.* Available at: http://sourceforge.net/projects/winflexbison/ (accessed 25.02.2018).

5. *GNU Bison.* Available at: http://www.gnu.org/software/bison/ (accessed 25.02.2018).

6. Parr T. *ANTLR (ANother Tool for Language Recognition).* Available at: http://www.antlr.org/ (accessed 25.02.2018).

7. Fedorchenko L. *Regularization of Context-Free Grammars.* Saarbrucken: LAP LAMBERT Academic Publishing, 2011. 188 p.

8. Fedorchenko L., Baranov S. Equivalent Transformations and Regularization in Context–Free Grammars. *Cybernetics and Information Technologies (CIT).* V. 14, No. 4, Pp. 11–28. 2015.

9. Aho A., Sethi R., Ullman J. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986. 796 p.

10. Martynenko B. K. Regular Languages and CF Grammars. *Computer Tools in Education.* 2012. No. 1. Pp. 14–20 (In Russian).

11. Fedorchenko L. N. Algoritmy postroeniya sostoyanii analizatora dlya KSR-yazyka [Algorithms for Constructing the Analyzer States for Context-Free Array Language]. *Vestnik Buryatskogo gosudarstvennogo universiteta. Matematika, informatika — Bulletin of Buryat State University. Mathematics, Computer Science.* 2016. No. 4. Pp. 23–33.

12. Fedorchenko L. N. Generatsiya testov v sisteme SynGT [Generation of Tests in SynGT System]. Vestnik Buryatskogo gosudarstvennogo universiteta. Matematika, informatika — *Bulletin of Buryat State University. Mathematics, Computer Science.* 2017. No. 2. Pp. 33–39.