МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМ. Р.Е. АЛЕКСЕЕВА»

Институт радиоэлектроники и информационных технологий

Кафедра «Прикладная математика»

Дисциплина: «Теория компиляции»

Курсовая работа на тему:

Проектирование и реализация систем лексического и синтаксического анализа программного кода

Выполнил:

студент гр. 19-ПМ-1

Гусев Дмитрий

Проверил:

Доцент кафедры «Вычислительные системы и технологии»

Жевнерчук Д. В.

Нижний Новгород

2021

Содержание

[Введение 3](#_Toc73110661)

[Глава 1. Постановка задачи 4](#_Toc73110662)

[Глава 2. Формальная модель задачи 5](#_Toc73110663)

[Глава 3. Спецификация основных процедур и функций 7](#_Toc73110664)

[3.1 Лексический анализатор 7](#_Toc73110665)

[3.2 Синтаксический анализатор 7](#_Toc73110666)

[Глава 4. Структурная организация данных 18](#_Toc73110667)

[4.1 Спецификация входных данных 19](#_Toc73110668)

[4.2 Спецификация выходных данных 19](#_Toc73110669)

[Глава 5. Разработка алгоритма решения задачи 20](#_Toc73110670)

[5.1 Укрупненная схема алгоритма программного средства 20](#_Toc73110671)

[5.2 Детальная разработка алгоритмов отдельных подзадач 20](#_Toc73110672)

[Глава 6. Установка и эксплуатация программного средства 23](#_Toc73110673)

[Глава 7. Работа с программным средством 34](#_Toc73110674)

[Заключение 35](#_Toc73110675)

[Литературный список 36](#_Toc73110676)

[Приложение 37](#_Toc73110677)

# Введение

Компилятор – программный модуль, задачей которого является перевод программы, написанной на одном из языков программирования (исходный язык) в программу на язык ассемблера или язык машинных команд.

Большинство компиляторов переводят программу с некоторого высокоуровневого языка программирования в машинный код, который может быть непосредственно выполнен компьютером.

Целью данной курсовой работы является изучение составных частей, основных принципов построения и функционирования компиляторов, практическое освоение методов построения составных частей компилятора для заданного входного языка.

Курсовая работа заключается в создании отдельных частей компилятора заданного языка.

В первой части работы ставится задача разработать программу(Сканнер), которая получает на входе набор идентификаторов, организует таблицу и позволяет осуществить многократный поиск идентификатора в этой таблице. В дальнейшем, требуется написать код, который выполняет лексический анализ входного текста по заданной грамматике и порождает таблицу лексем с указанием их типов.

В второй части работы необходимо разработать программу(Парсер), которая получает на входе грамматику и строку. Парсер будет проверять строку на правильность с точки зрения заданной грамматики.

Глава 1. Постановка задачи

Цель курсовой работы – разработать компилятор модельного языка на языке программирования Java.

Задачи курсовой работы:

1. В соответствии с номером варианта составить формальное описание

модельного языка программирования с помощью:

1. РБНФ;
2. формальных грамматик;
3. Написать пять содержательных примеров программ, раскрывающих особенности конструкций учебного языка программирования, отразив в этих

примерах все его функциональные возможности;

1. Составить таблицы лексем и диаграмму состояний с действиями для

распознавания и формирования лексем языка;

1. По диаграмме с действиями написать функцию сканирования текста

входной программы на модельном языке;

1. Разработать программное средство, реализующее лексический анализ

текста программы на входном языке;

1. Реализовать синтаксический анализатор текста программы на модельном языке методом рекурсивного спуска;
2. Построить цепочку вывода и дерево разбора простейшей программы на

модельном языке из начального символа грамматики;

1. Распечатать пример таблиц идентификаторов и двуместных операций;
2. Показать динамику изменения содержимого стека при семантическом анализе программы на примере одного синтаксически правильного выражения;
3. Составить набор контрольных примеров, демонстрирующих все возможные типы лексических, синтаксических и семантических ошибок в программах на модельном языке.

Глава 2. Формальная модель задачи

Ниже представлены правила, операции и выражения модельного языка:

1. Операции языка:
2. Синтаксис группы операций «отношение» (в порядке следования: неравно, равно, меньше, меньше или равно, больше, больше или равно) –

<операции\_группы\_отношения>:: = < > | = | < | <= | > | >=

1. Синтаксис группы операций «сложения» (в порядке следования: сложение, вычитание, дизъюнкция) –

<операции\_группы\_сложения>:: = + | - | *or*

1. Синтаксис группы операций «умножение» (в порядке следования: умножение, деление, конъюнкция) –

<операции\_группы\_умножения>::= \* | / | *and*

1. Синтаксис унарной операции –

<унарная\_операция>::= *not*

1. Выражения языка:
2. <выражение>::= <операнд>{<операции\_группы\_отношения> <операнд>}
3. <операнд>::= <слагаемое> {<операции\_группы\_сложения> <слагаемое>}
4. <слагаемое>::= <множитель> {<операции\_группы\_умножения> <множитель>}
5. <множитель>::= <идентификатор> | <число> | <логическая\_константа> |<унарная\_операция> <множитель> | (<выражение>)
6. <число>::= {/ <цифра> /}
7. <логическая\_константа>::= true | false
8. <идентификатор>::= <буква> {<буква> | <цифра>}
9. <буква>::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
10. <цифра>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
11. Правило, определяющее структуру программы –

<программа>::= program var <описание> begin <оператор> {;<оператор> } end.

1. Правило, определяющее раздел описания переменных –

<описание>::= <тип> <идентификатор> { , <идентификатор> }

1. Правило, определяющее типы данных (в порядке следования: целый, действительный, логический) –

<тип>::= int | float | bool

1. Правило, определяющее оператор программы –

<оператор>::= <составной> | <присваивания> | <условный> |

<фиксированного\_цикла> | <условного\_цикла> | <ввода> |

<вывода>:

1. <составной>::= <оператор> { ( : | перевод строки) <оператор> }
2. <присваивания>::= <идентификатор> ass <выражение>
3. <условный>::= if <выражение> then <оператор> [ else <оператор>]
4. <фиксированного\_цикла>::= for <присваивания> to <выражение> do <оператор>
5. <условного\_цикла>::= while <выражение> do <оператор>
6. <ввода>::= read (<идентификатор> {, <идентификатор> })
7. <вывода>::= write (<выражение> {, <выражение> })
8. Многострочные комментарии в программе:
9. Признак начала комментария – {
10. Признак конца комментария – }

# Глава 3. Спецификация основных процедур и функций

## 3.1 Лексический анализатор

## Лексический анализатор (сканер) читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся лексемами.

## Лексема – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своём составе других структурных единиц языка. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т.п. На вход лексического анализатора поступает текст исходной программы, а выходная информация передаётся для дальнейшей обработки синтаксическому анализатору.

## Для каждой лексемы сканер строит выходной токен (англ. token – знак, символ) вида ‹имя\_токена, значение\_атрибута› Первый компонент токена, имя\_токена, представляет собой абстрактный символ, использующийся во время синтаксического анализа, а второй компонент, значение атрибута, указывает на запись в таблице идентификаторов, соответствующую данному токену.

Лексические анализатор в виде автоматной модели. (рис. 3.1.1).

*Рис. 3.1.1. Автоматная модель ЛА*.

## Ниже представлена таблица описания функций лексического анализатора. (Табл. 3.1.1)

|  |  |  |  |
| --- | --- | --- | --- |
| Название функции | Входные данные | Выходные данные | Описание функции |
| getLexems | String source – строка для разделения на лексемы  Int index – номер строки, необходим для более оптимального вывода результата | void | Функция разделяет строку на лексемы, ищет зарезервированные слова и выводит их. |
| getSourceLexem | ArrayList<Leksema> leksems – лист, хранящий лексемы. Класс Leksema содержит номер таблицы, номер слова в таблице и его местоположение в строке. | String result – строка для вывода изначальной строки в виде токенов. | Функция выводит изначальную строку в виде токенов. |
| getSource | List<Leksema> leksems - лист, хранящий лексемы. | String result – содержит необходимые слова вместо токенов. | Функция выводит строку, полученную путем преобразования строки из токенов в обычную строку. Т.е. вместо токенов подставляются необходимые слова или символы. |

*Табл. 3.1.1 Таблица описания функций лексического анализатора*

## Таблица зарезервированных слов и разделителей (Табл. 3.1.2)

|  |  |  |
| --- | --- | --- |
| № | Зарезервированное слово | Разделитель |
| 1 | or | <> |
| 2 | not | <= |
| 3 | program | >= |
| 4 | var | < |
| 5 | begin | > |
| 6 | end | = |
| 7 | int | + |
| 8 | float | - |
| 9 | bool | \* |
| 10 | ass | / |
| 11 | then | { |
| 12 | else | } |
| 13 | for | . |
| 14 | to | ; |
| 15 | do | , |
| 16 | while | : |
| 17 | and | \n |

## *Таблица зарезервированных слов и разделителей (Табл. 3.1.2)*

## Вывод ЛА осуществляется путем вывода двух чисел: номер таблицы, где находится лексема и номер лексемы в таблице. Рассмотрим на примере «program var name begin». Для данного примера будет построена ещё список незарезервированных слов. (Табл. 3.1.3)

|  |  |
| --- | --- |
| № | Незарезервированное слово |
| 1 | name |

*Таблица незарезервированных слов (Табл. 3.1.3)*

В итоге, лексический анализатор выведет следующую строку токенов (Табл. 3.1.4) (Примечание: мною было объединено «program var» в одно зарезервированное слово, поэтому лексем получится всего 3):

|  |  |  |  |
| --- | --- | --- | --- |
| № | 1 | 2 | 3 |
| Таблицы | 5 | 12 | 5 |
| Элемента | 0 | 0 | 1 |

*Строка с токенами(Табл.3.1.4)*

## 3.2 Синтаксический анализатор

Парсер или синтаксический анализатор, — часть программы, преобразующей входные данные (как правило, текст) в некий структурированный формат, нужный для задач последующего их (данных) анализа и использования. Технически, парсер выполняет синтаксический анализ данных (например, текста). Синтаксический анализ в информатике – процесс сопоставления линейной последовательности лексем естественного или формального языка с его формальной грамматикой. В данной курсовой работе, синтаксический анализатор будет выполнять задачи: синтаксический анализ кода, поиск ошибок и вывода их.

В данном варианте курсовой работы формальная грамматика выглядит следующим образом:

1. D->program D2 B | I1 int | I1 float | I1 bool
2. D2->var I
3. I1->I | I1,I
4. B->begin S1 { S1 end
5. S1->S | S1;S
6. S->S1{S1}|if E then S [else S] | for S to E do S | I ass S | read ({ I}) | write ({E})
7. E->E1 | E1 = E1 | E1>E1 | E1<E1 | E1<>E1 | E1 <= E1 | E1 >= E1
8. E1->T | T + E1 | T- E1 | T or E1
9. T->F | F\*T | F / T | F and T
10. F->I | N | L | not F | (E)
11. L->true | false
12. I->C | IC | IR
13. N->R | NR
14. C-> a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
15. R->0|1|2|3|4|5|6|7|8|9

Грамматика в форме Бэкуса-Наура:

<буква>::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<цифра> ::=0|1|2|3|4|5|6|7|8|9

<идентификатор>::= <буква> {<буква>|<цифра>}

<число>::={/<цифра>/}

<ключевое\_слово>::=program|var|int|bool|read|write|if|then|else|while|do|true|false|ass |for|to|float|begin|end

<разделитель>::= .|;|,|:

<программа>::= program var <описание> begin <оператор> {;<оператор> } end.

<описание>::= <тип> <идентификатор> { , <идентификатор> }

<оператор>::=<составной>|<присваивания>|<условный>|<фиксированного\_цикла> | <условного\_цикла> | <ввода> |<вывода>:

<присваивания>::= <идентификатор> ass <выражение>

<условный>::= if <выражение> then <оператор> [ else <оператор>]

<фиксированного\_цикла>::= for <присваивания> to <выражение> do <оператор>

<составной>::= <оператор> { ( : | перевод строки) <оператор> }

<ввода>::= read (<идентификатор> {, <идентификатор> })

<вывода>::= write (<выражение> {, <выражение> })

<условного\_цикла>::= while <выражение> do <оператор>

<выражение>::= <операнд>{<операции\_группы\_отношения> <операнд>}

<операнд>::= <слагаемое> {<операции\_группы\_сложения> <слагаемое>}

<слагаемое>::= <множитель> {<операции\_группы\_умножения> <множитель>}

<множитель>::=<идентификатор>|<число>|<логическая\_константа>| <унарная\_операция> <множитель> | (<выражение>)

<логическая\_константа>::= true | false

Синтаксический анализатор обрабатывает строку, проверяя может ли строка порождаться грамматикой исходного языка. Если строка не проходит проверку, то необходимо вывести сообщение об ошибке.

Таблица функции СА. (Табл. 3.2.1):

|  |  |  |  |
| --- | --- | --- | --- |
| Название функции | Входные данные | Выходные данные | Описание функции |
| Parsing | ArrayList<String> scanner\_leksems – лист, содержащий строки для обработки СА. | void | Основная работа парсера(СА). Функция считывает грамматику и обрабатывает эту грамматику. Создает First, Follow, выводит ошибки и проверяет строку на правильность, с точки зрения грамматики. |
| CreateFirst | - | void | Создает множество First |
| Create\_Follow | - | void | Создает множество Follow |
| Iteration\_String | ArrayList<String> leksem – строка, разбитые на лексемы.  ArrayList<String> stack – стек для проверки правильности строки.  int ex\_index – индекс, необходимый для проверки зацикленности рекурсии | boolean – возвращает true, если строка верная, false – неверная. | Рекурсивная функция, служащая для проверки правильности строки, с помощью перебора всех правил и использования множеств First и Follow |
| AddTerminal | - | void | Функция для разделения терминалов и нетерминалов в грамматике |

*Таблица функций синтаксического анализатора (Табл. 3.2.1)*

При построении синтаксического анализатора помогают два множества – FIRST и FOLLOW, связанные с грамматикой. Они позволяют выбрать применяемое правило на основании очередного символа входного потока.

Алгоритм построения множества FIRST (1, А) для каждого нетерминала А: 1. Первоначально внести во множество первых символов для каждого

нетерминального символа А все символы, стоящие в начале правых частей правил для этого нетерминального символа, т.е. ∀ А ∈ VN FIRST0(1, A) = {X | A→Xα ∈P, X∈ (VT∪VN), α∈ (VT∪VN)\*}.

2. Для всех А ∈ VN положить: FIRSTi+1(1, A) = FIRSTi(1, A) ∪ FIRSTi(1, B), ∀ В∈ (FIRST(1, A)⋂VN).

3. Если существует А∈VN, такой что FIRSTi+1(1, A) ≠ FIRSTi(1, A), то присвоить *i=i+1* и вернуться к шагу 2, иначе перейти к шагу 4.

4. Исключить из построенных множеств все нетерминальные символы, т.е. ∀ А ∈ VN FIRST(1, A) = FIRSTi(1, A) \ N.

Алгоритм построения множества FOLLOW (1, А) для каждого нетерминала А:

1. Первоначально внести во множество последующих символов для каждого нетерминального символа А все символы, которые в правых частях правил вывода встречаются непосредственно за символом А, т.е. ∀ А ∈ VN FOLLOW0(1, A) = {X | ∃B→αAXβ ∈ P, B ∈ VN, X ∈ (VT∪VN), α, β ∈ (VT∪VN)\*}.

2. Внести пустую строку во множество FOLLOW (1, S), т.е. FOLLOW (1, S) = FOLLOW (1, S) ∪ {ε}

3. Для всех А ∈ VN вычислить: FOLLOW’i(1, A) = FOLLOWi(1, A) ∪ FIRST (1, B), ∀ B∈ FOLLOWi(1, A) ⋂ VN

4. Для всех А ∈ VN положить: FOLLOW’’i(1, A) = FOLLOW’i(1, A) ∪ FOLLOW’i(1, B), ∀ B∈ FOLLOW’i (1, A) ⋂ VN, если ∃ правило B→ ε

5. Для всех А ∈ VN определить: FOLLOWi+1(1, A) = FOLLOW’’i(1,A) ∪ FOLLOW’’i(1,B), для всех нетерминальных символов B ∈ VN , имеющих правило вида B→αA, α ∈ (VT∪VN)\*

6. Если существует А∈ VN такой, что FOLLOWi+1(1, A) ≠ FOLLOWi(1, A), то положить i=i+1 и вернуться к шагу 3, иначе перейти к шагу 7.

7. Исключить из построенных множеств все нетерминальные символы, т.е. ∀ А∈ VN FOLLOW(1, A) = FOLLOWi(1, A) \ N.

Ниже представлены таблицы FIRST и FOLLOW (Табл. 3.2.2) для заданной грамматики, где:

C-> a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

R->0|1|2|3|4|5|6|7|8|9

|  |  |  |
| --- | --- | --- |
| Нетерминал | First | Follow |
| B | begin | - |
| C | C | ass, }, int, float, bool, “,” , \*, /, and, +, -, or, =, >, <, <>, <=, >=, then, do, ), C, R, begin |
| D | program, C | - |
| E | not, (, C, R, true, fasle | then, do, }, ) |
| F | not, (, C, R, true, fasle | \*, /, and, +, -, or, =, >, <, <>, <=, >=, then, do, }, ) |
| I1 | C | int, float, bool, “,” |
| I | C | ass, }, int, float, bool, “,” , \*, /, and, +, -, or, =, >, <, <>, <=, >=, then, do, ), C, R, begin |
| E1 | not, (, C, R, true, fasle | =, >, <, <>, <=, >=, then, do, }, ) |
| L | true, fasle | \*, /, and, +, -, or, =, >, <, <>, <=, >=, then, do, }, ) |
| D2 | var | begin |
| N | R | \*, /, and, +, -, or, =, >, <, <>, <=, >=, then, do, }, ), R, ass, C |
| R | R | ass, }, int, float, bool, “,” , \*, /, and, +, -, or, =, >, <, <>, <=, >=, then, do, ), C, R, begin |
| S | if, for, read, write, C | [, ], to, {, end, ;, } |
| T | not, (, C, R, true, fasle | \*, /, and, +, -, or, =, >, <, <>, <=, >=, then, do, }, ) |
| S1 | if, for, read, write, C | {, end, ;, } |

*Множества First и Follow (Табл. 3.2.2)*

Дальше происходит проверка строки на правильность. Во-первых, ищется необходимое правило, благодаря множествам First и Follow. Берется первая поданная лексема из списка, и если оно содержится в множестве First или Follow, то в стек добавляется нетерминал.

Во-вторых, ищется подходящее правило в грамматике для подстановки. Если такое правило находится, то нетерминал в стеке заменяется на это правило. Эти два шага повторяется до тех пор, пока в буфере ввода есть лексемы.

Для этого алгоритма мною была использована рекурсия для того, чтобы в случае ошибки программа возвращалась к предыдущим шагам и подставляла другие подходящие правила или нетерминалы.

Рассмотрим на примере строки “a int;”. Для этого примера синтаксический анализ будет выглядеть следующим образом (Табл. 3.2.3):

|  |  |  |  |
| --- | --- | --- | --- |
| № шага | Стек | Входной буфер | Действие |
| 1 | D | a int; | Свертка D->I1 int;, т.к. a ∈ FIRST (1,D) |
| 2 | I1 int; | a int; | Свертка I1->I, т.к. a ∈ First(1,I1) |
| 3 | I int; | a int; | Свертка I->a т.к. a ∈ First(1,I) |
| 4 | a int; | a int; | Выброс |
| 5 | int; | int; | Выброс |
| 6 | ε | ε | Строка корректна |

*Пример синтаксического анализа (Табл. 3.2.3)*

Цепочка вывода выглядит так: D->{I1 int;}->{I int;}->{a int;}

Дерево разбора(рис.3.2.1):

D

int;

I1

I

a

*Дерево разбора(Рис.3.2.1)*

# Глава 4. Структурная организация данных

# Спецификация входных данных

На вход сканнеру задается название файла. Из файла считывается текст для дальнейшей обработки. Для хранения синтаксиса и зарезервированных слов была использована структура данных – лист. Для каждой группы зарезервированных слов был создан отдельный лист. Данные листы помещались в общий массив. Список незарезервированных слов создается и помещается в общий массив в процессе работы сканнера.

На вход парсеру передается лист со строками. Данный лист создается при работе сканнера, а в этот лист помещены строки из текста, разделенные разделителями. Грамматика, с которой работает парсер, должна быть описана в файле. Таким образом, необходимо задать парсеру название файла, в котором содержится нужная грамматика, в коде.

## Спецификация выходных данных

Сканнер выводит:

1)Номер строки

2) Каждое найденное слово, к какой группе правил оно принадлежит, его позицию и код лексемы.

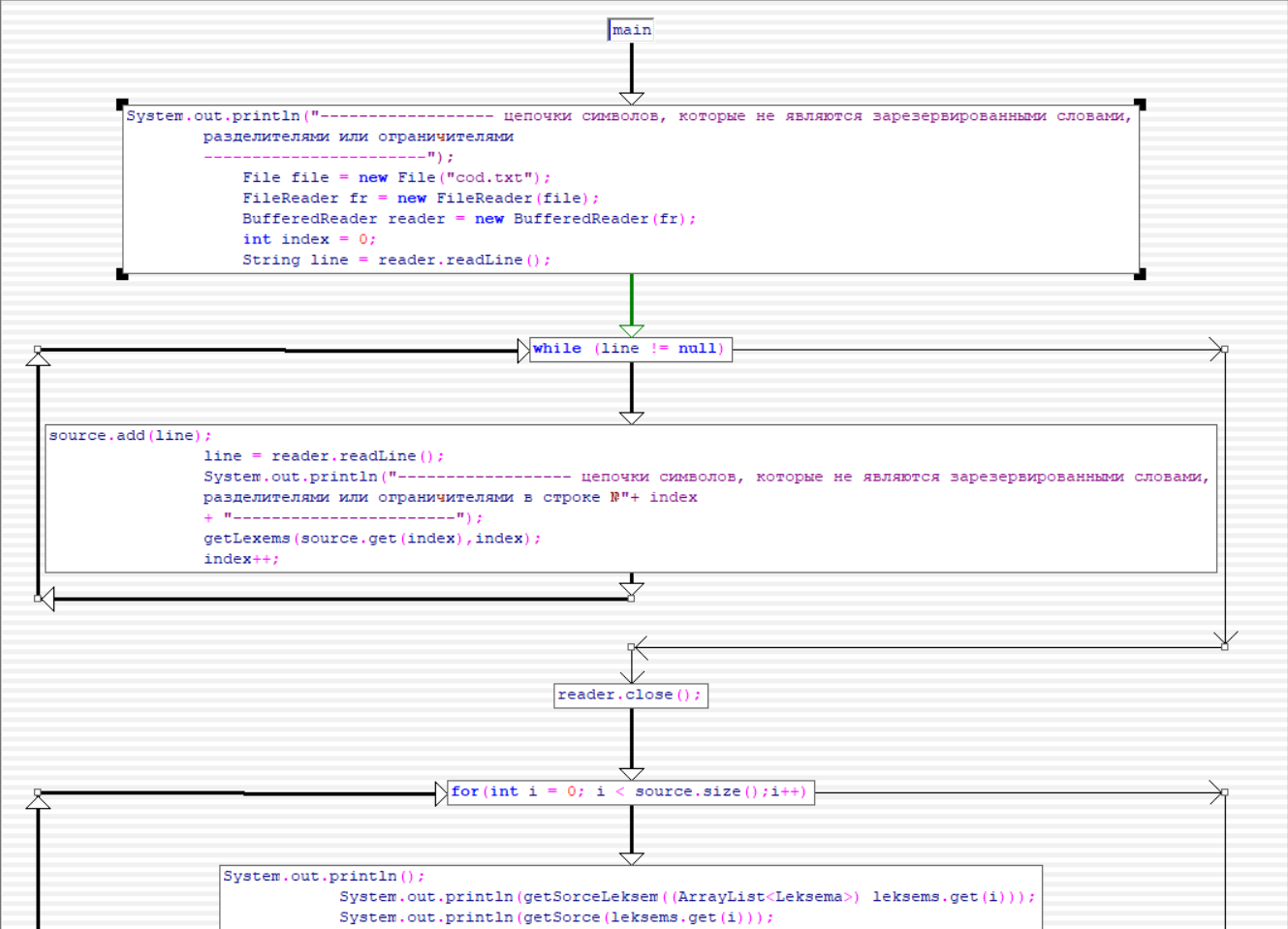
1. Строку, состоящую из кода лексем
2. По коду лексем выстраивает и выводит первоначальную строку

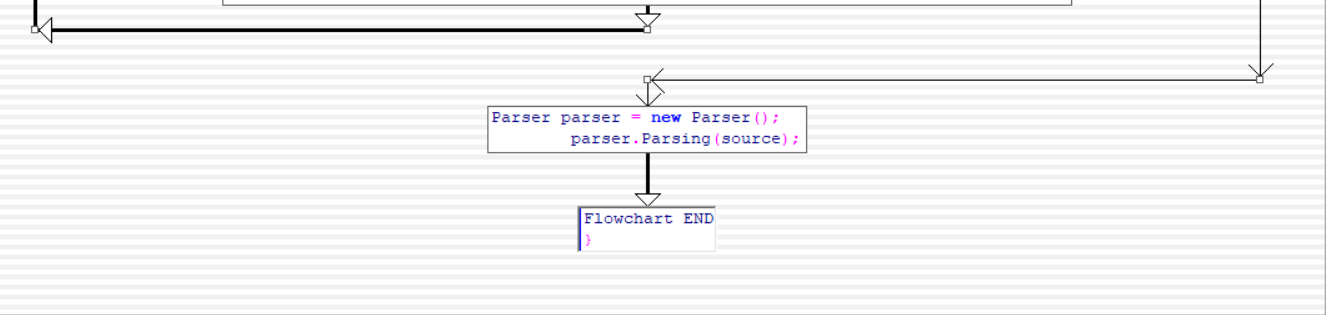
Парсер выводит сообщение правильно ли введена строка, с точки зрения данной грамматики. Если парсер определил причину ошибки, то выводит причину на экран. В ином случае, будет выведено только, что строка задана неправильно.

# Глава 5. Разработка алгоритма решения задачи

# Укрупненная схема алгоритма программного средства

Метод main в Сканнере(Блок-схема 5.1.1):





*Метод Main в виде блок схемы (Блок-схема 5.1.1)*

В функции main происходит следующие:

1) Чтение строк из файла

2) Добавление строк в Source

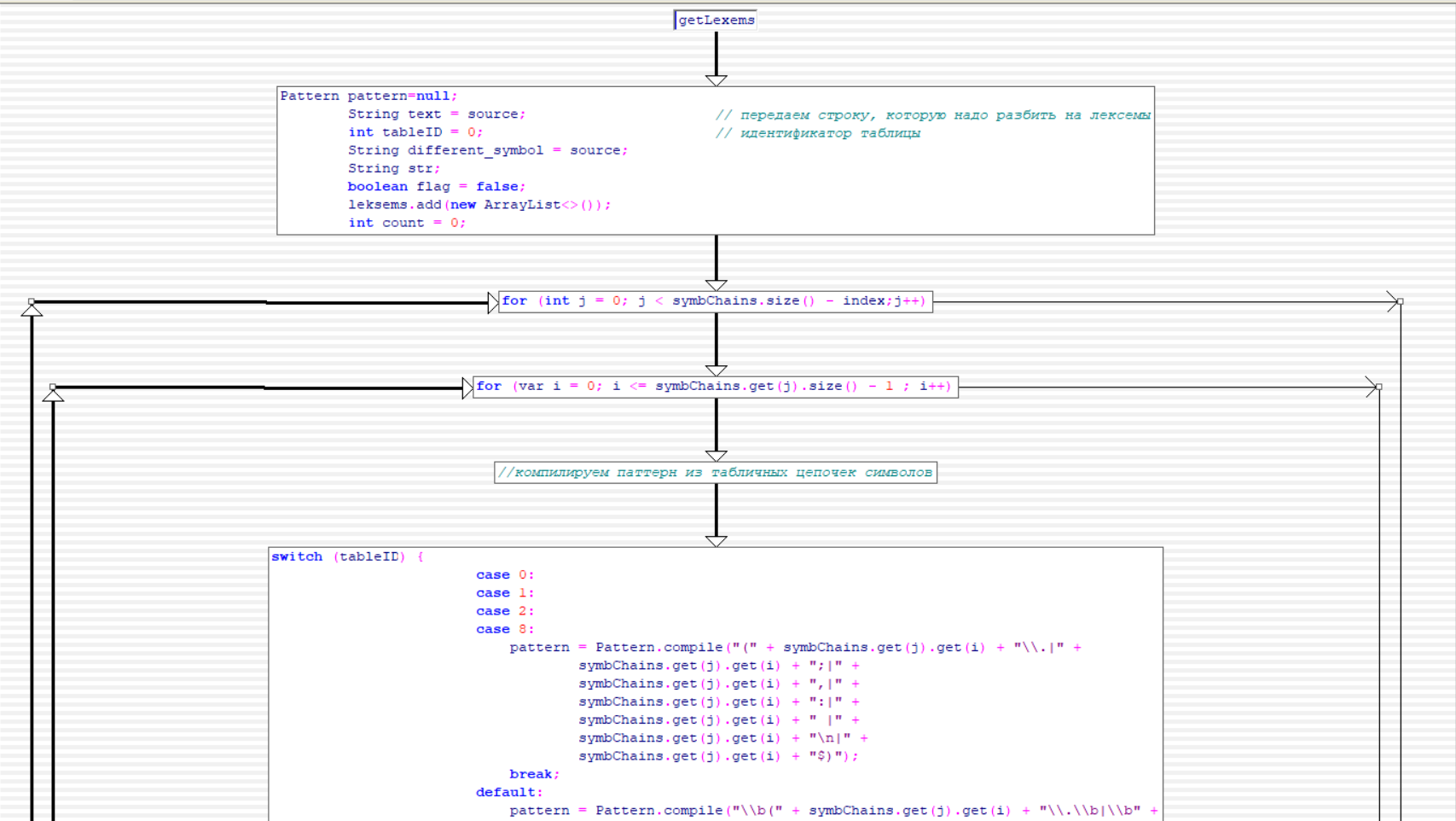
3) Передача строки в метод по разбиение на лексемы(getLexems)

4) Вывод на экран строки из токенов, с помощью метода getSourceLexem

5) Вывод на экран строки, преобразованную из строки токенов(метод – getSource)

## 5.2 Детальная разработка алгоритмов отдельных подзадач

Метод GetLexem (Блок-схема 5.2.1):



D

C

B

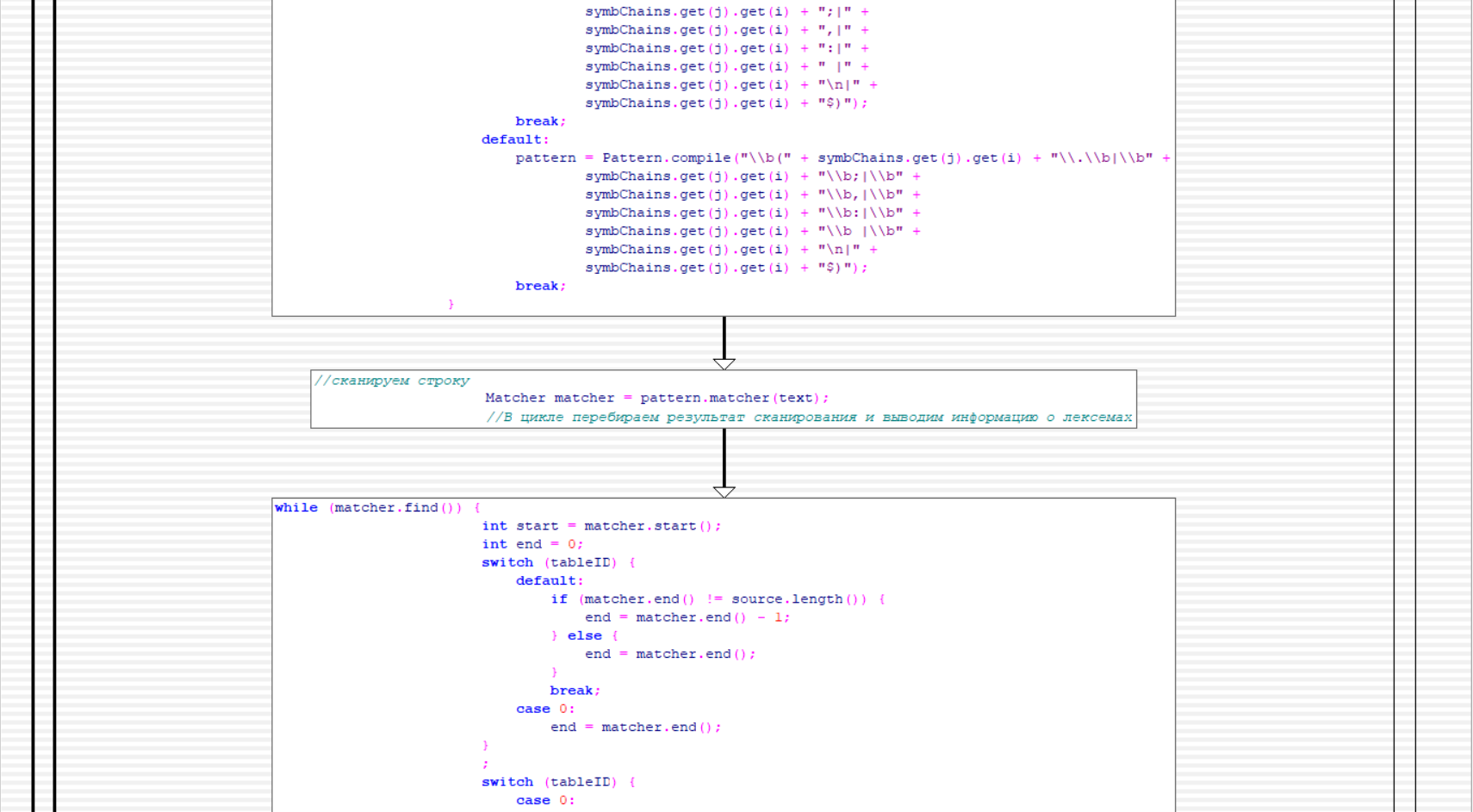
A

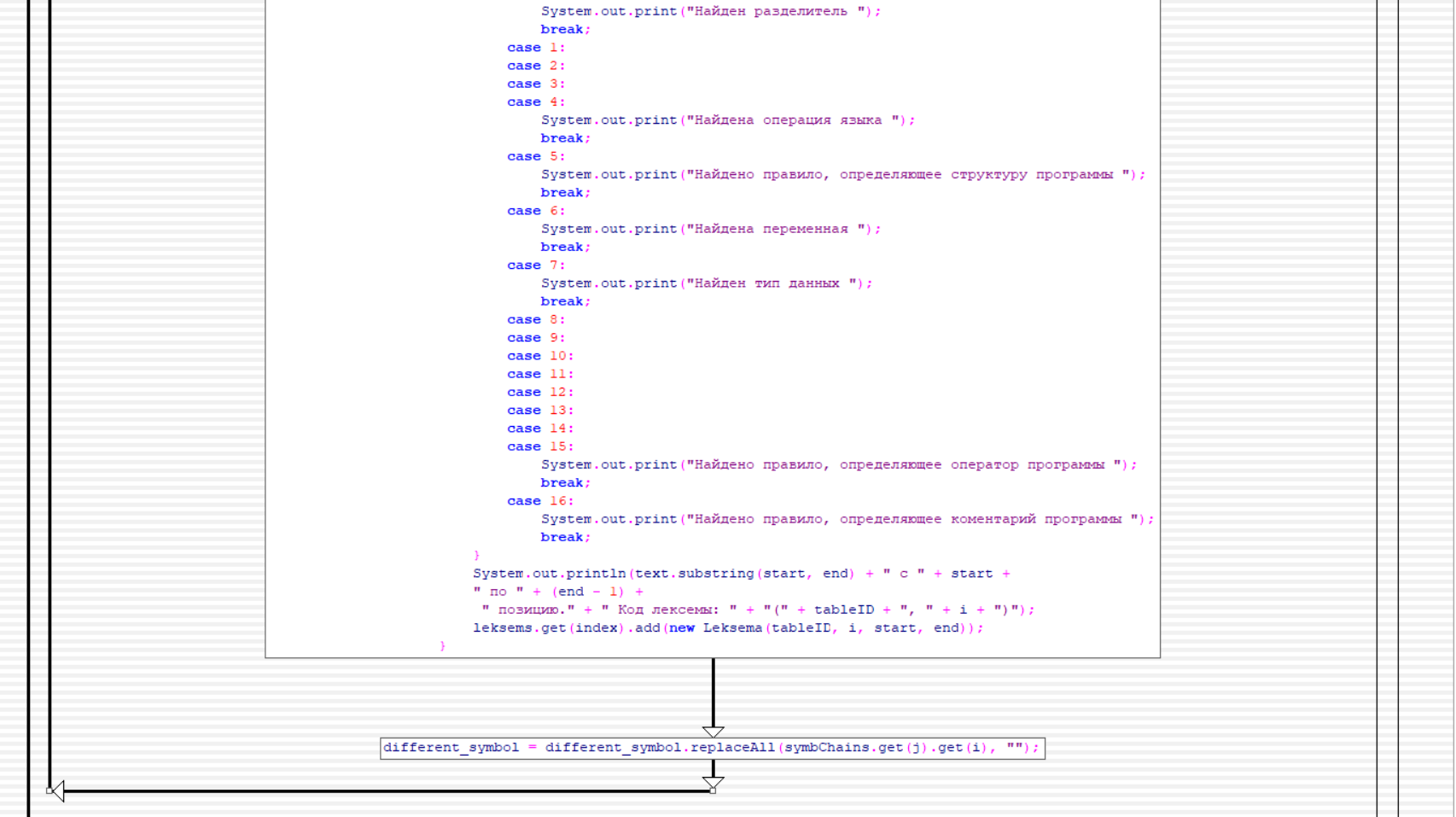
D

C

B

A





A

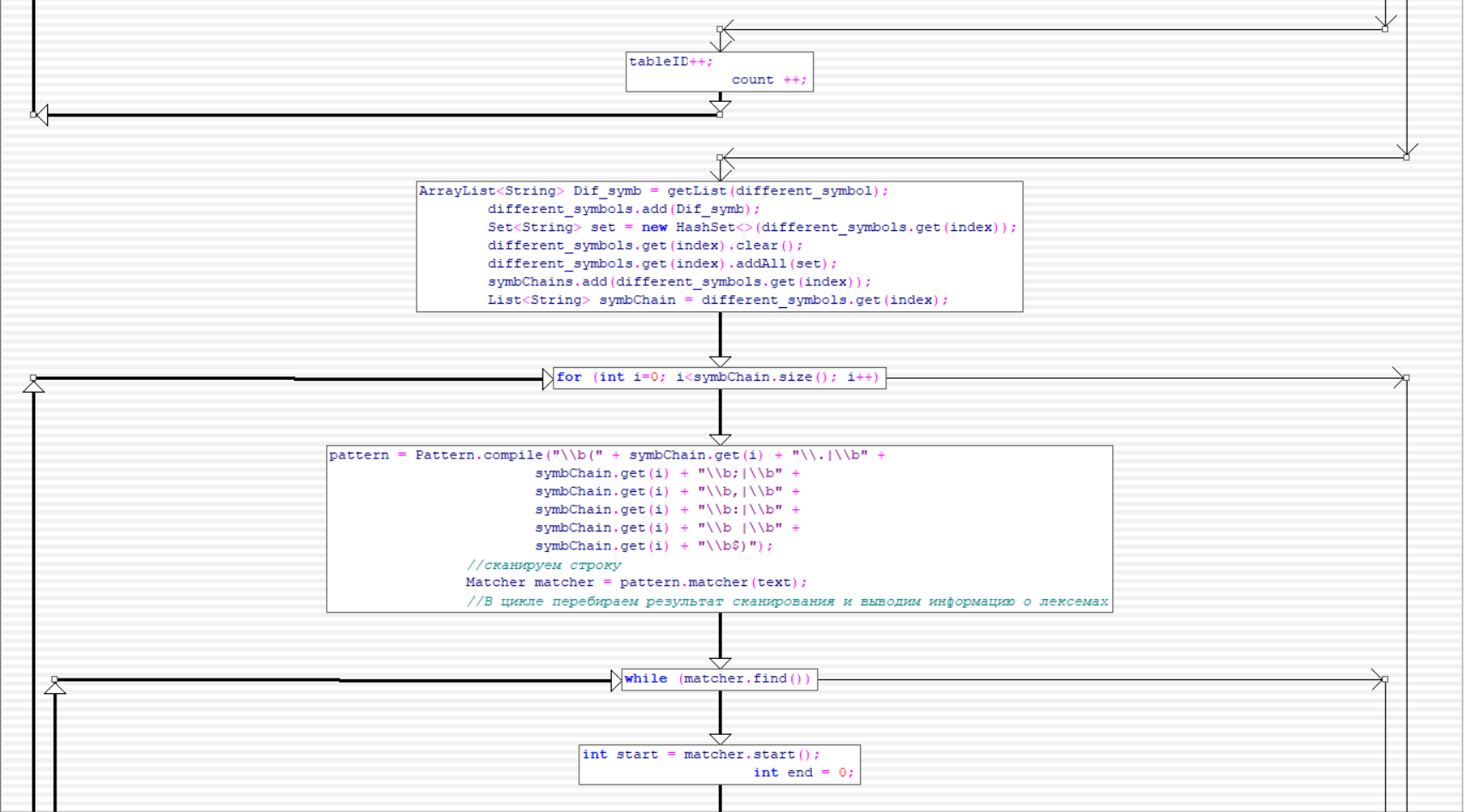
C

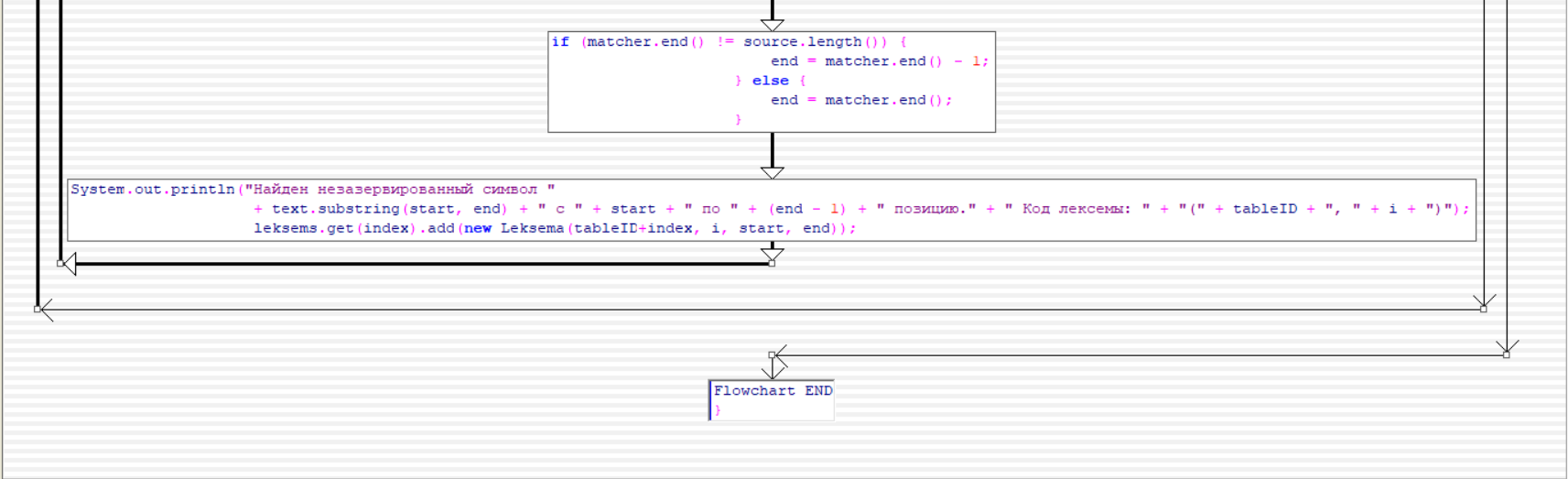
D

D

C

A



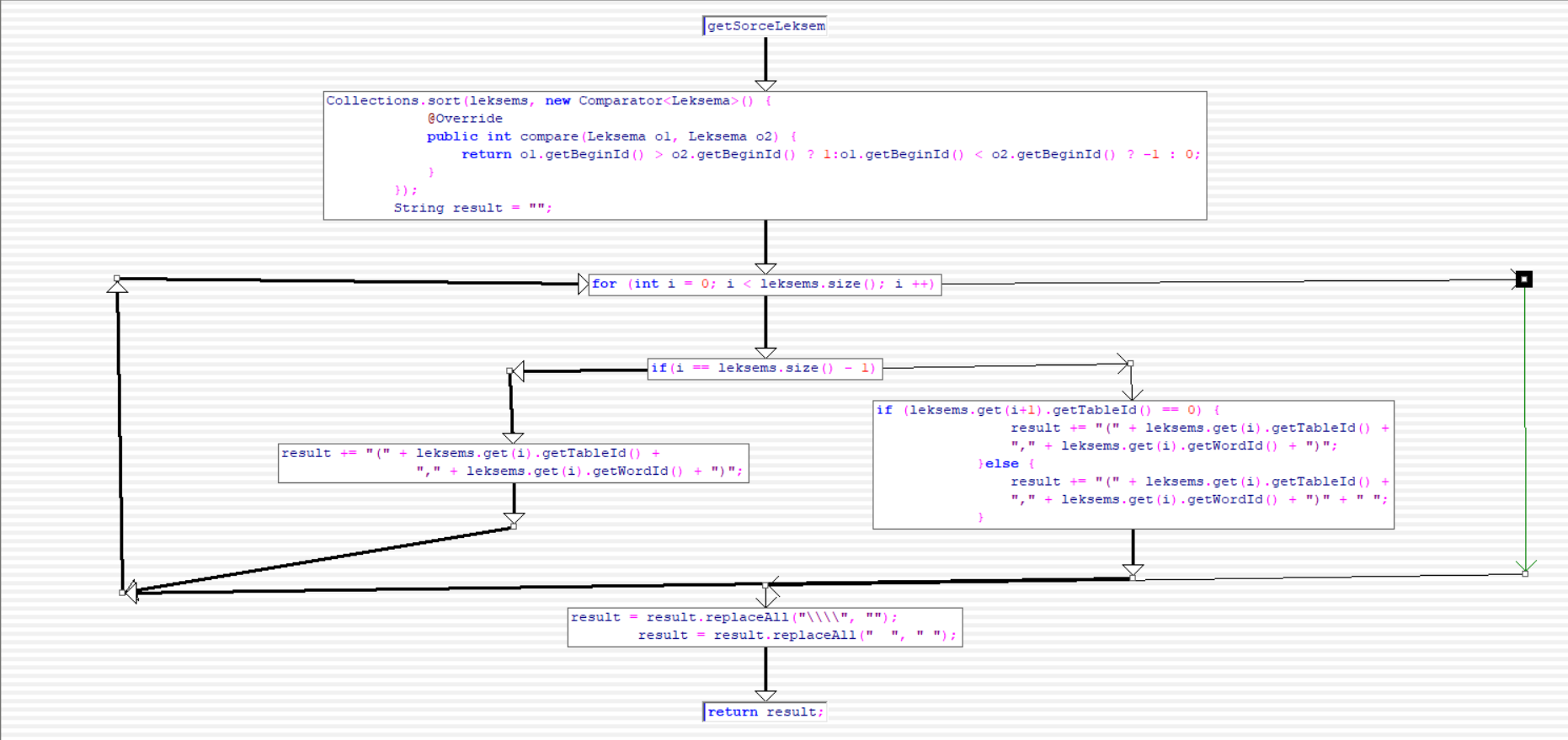


*Метод GetLexem в виде блок схемы (Блок-схема 5.2.1)*

Метод Get Lexem выполняет:

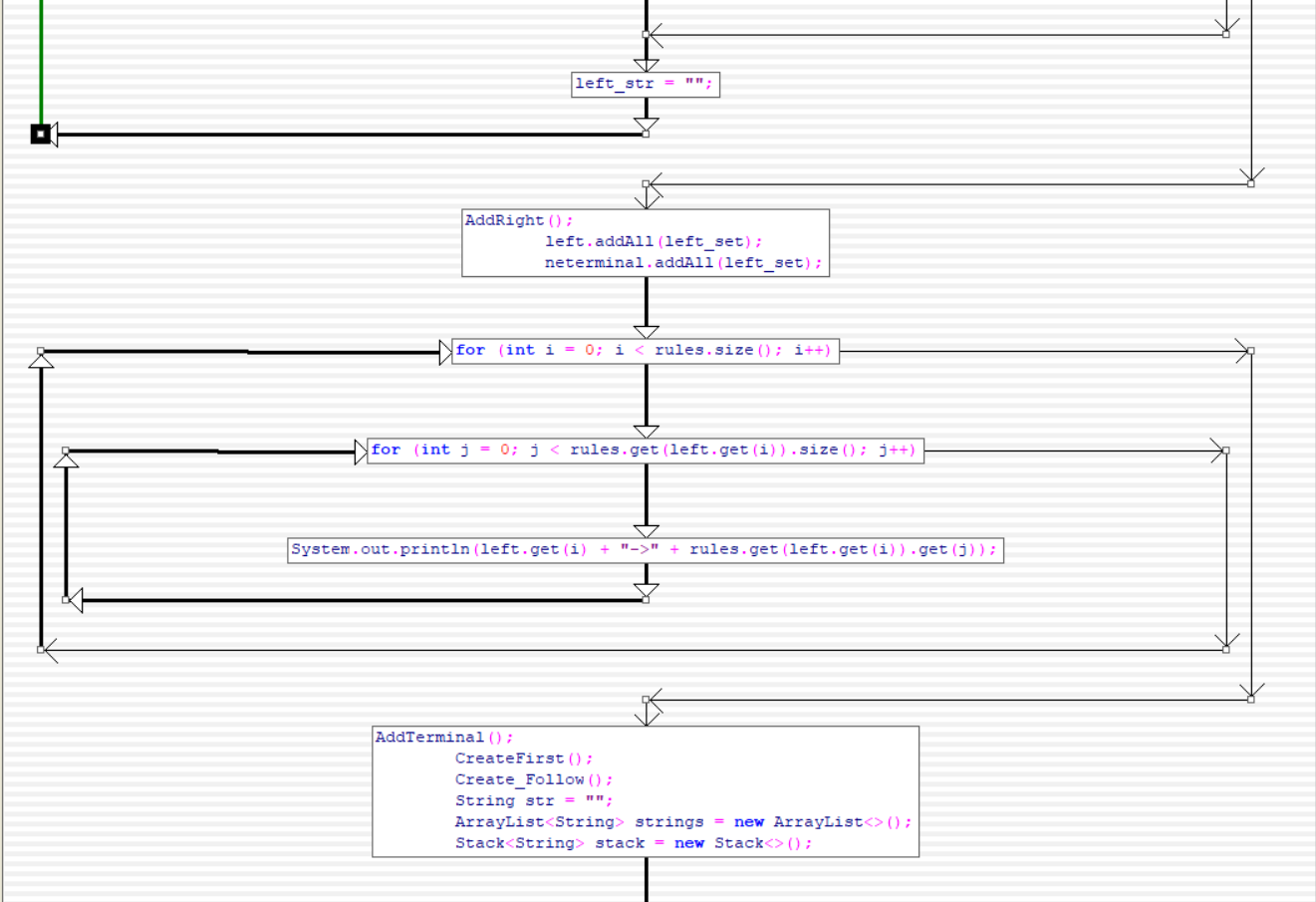
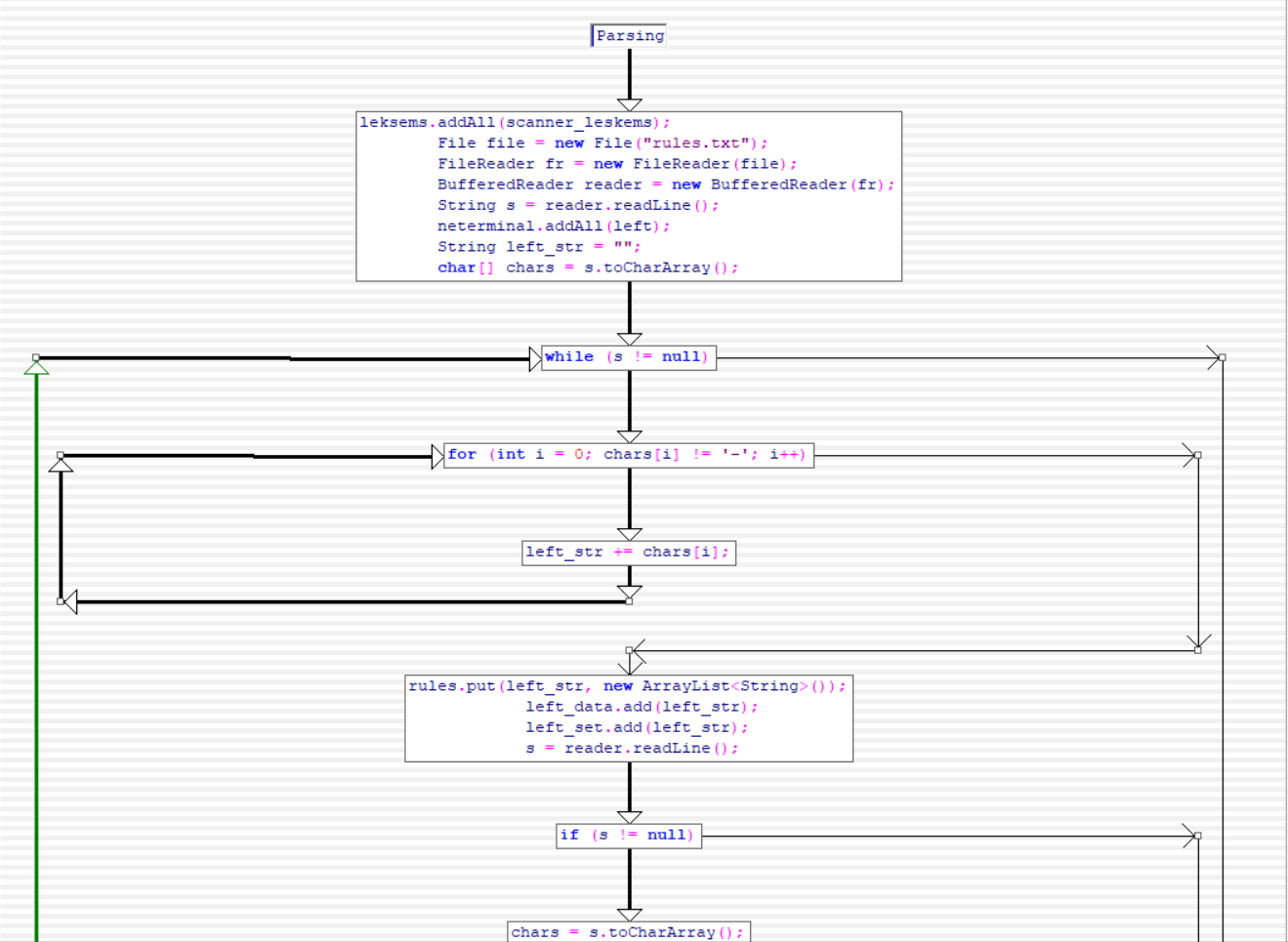
1. Создает регулярное выражение, по принципу слово из таблицы и прибавлением к нему разделители
2. Все, что будет найдено с помощью регулярного выражение обрабатывается:
   1. Смотрит на особенность номера таблицы с помощью switch, у каждой группы зарезервированных слов свой способ вывода
   2. Выводит на экран найденное слово
3. Все слова, которые не были найдены добавляются в отдельный список
4. Этот список добавляется в общую таблицу
5. Потом повторяем первые два шага для этого списка
6. Выводит нерезервированные слова

Метод GetSourceLexem(Блок-схема 5.2.2):



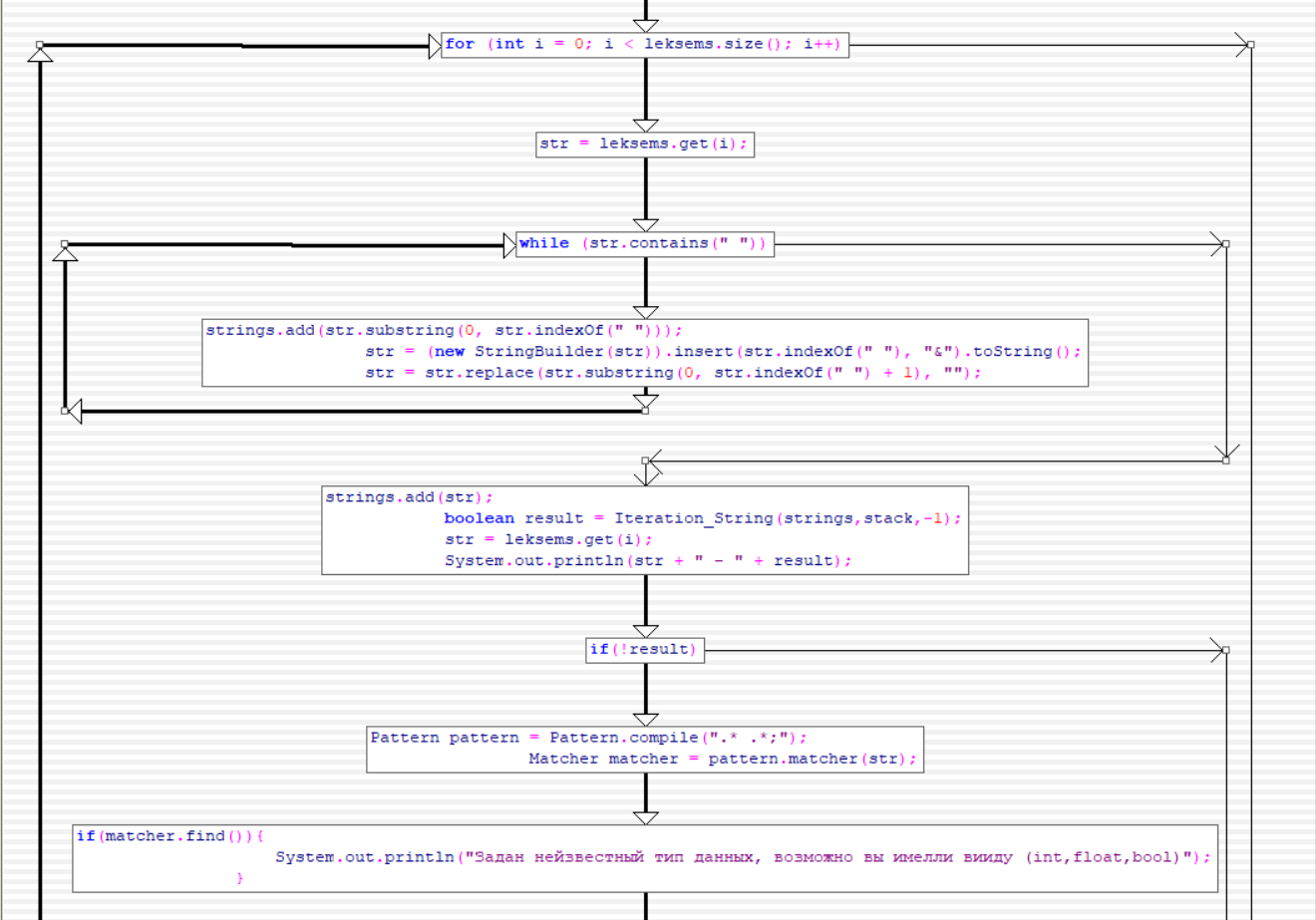
*Метод GetSourceLexem в виде блок схемы (Блок-схема 5.2.2)*

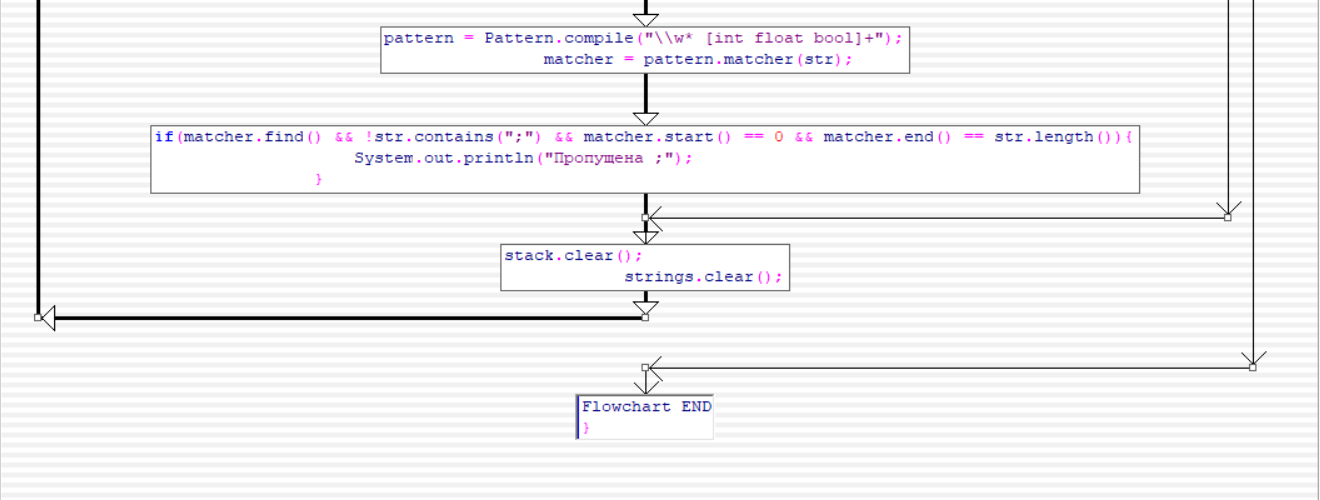
Метод GetSourceLexem отвечает за построение и за вывод на экран строку из токенов.

Метод Parsing(Блок-схема 5.2.3)

A

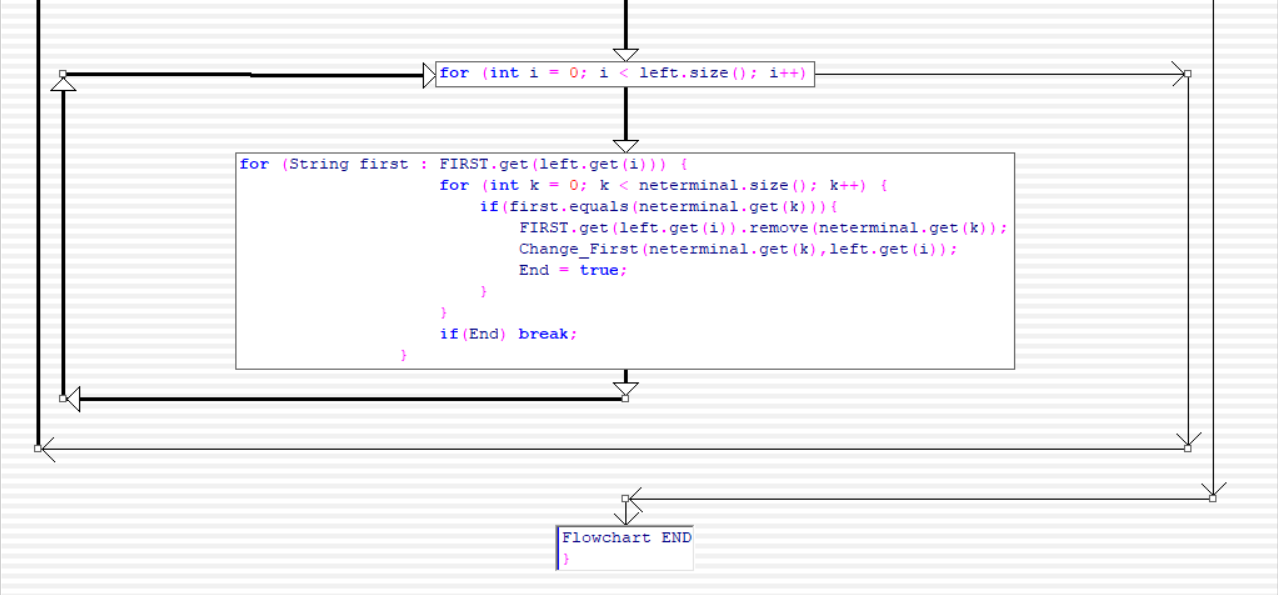
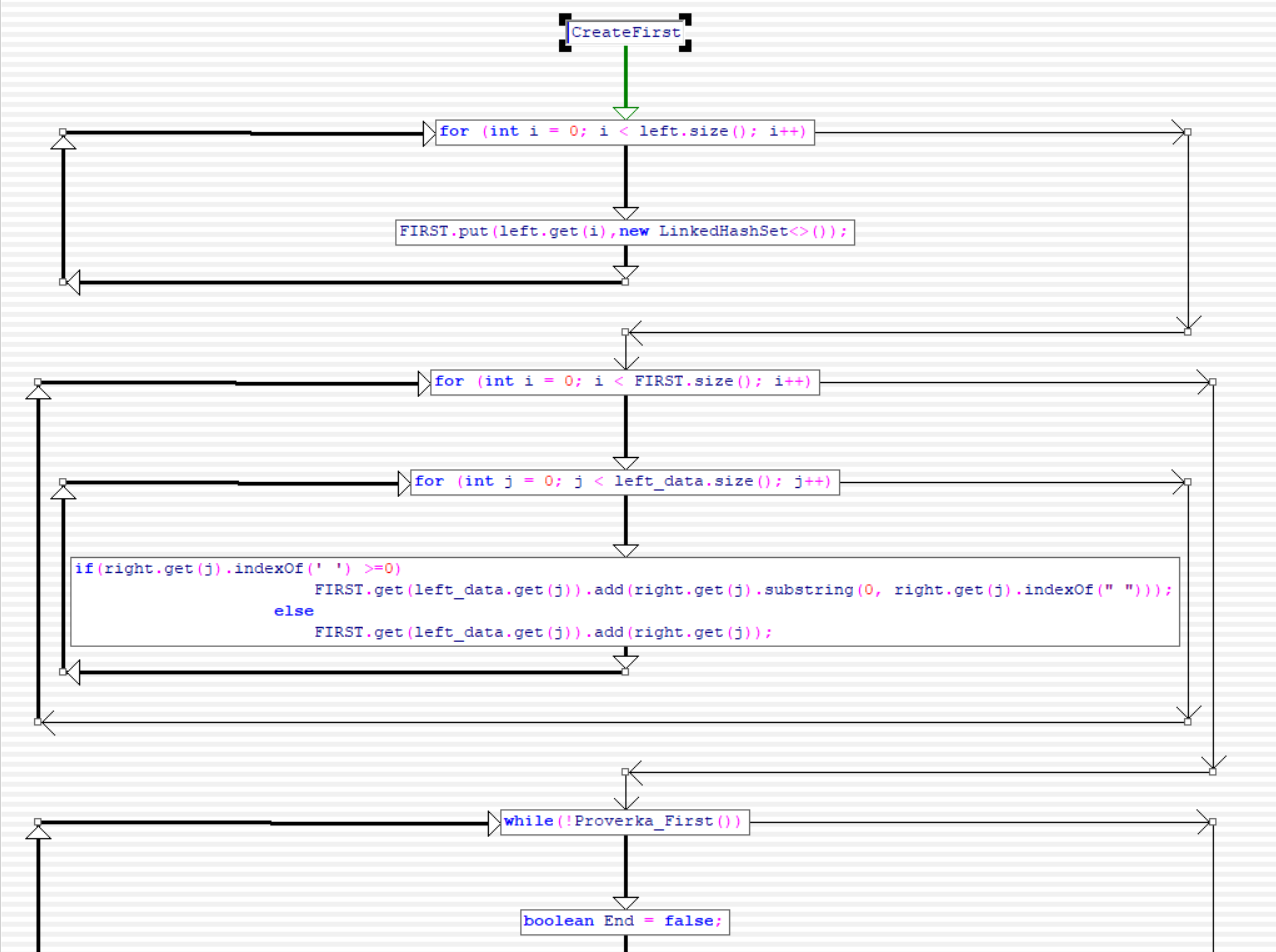
A





*Метод Parsing в виде блок схемы (Блок-схема 5.2.1)*

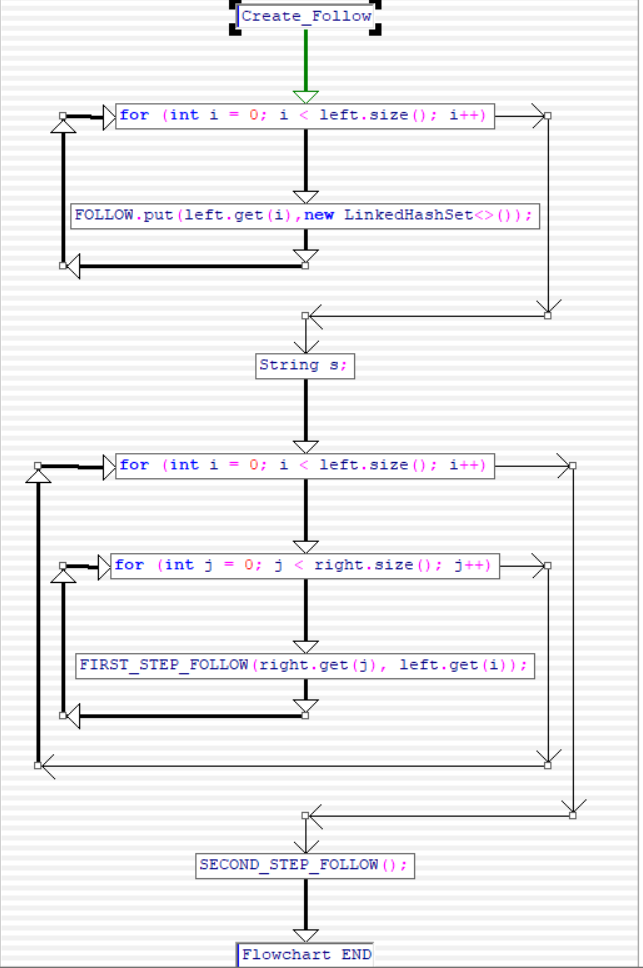
Метод CreateFirst (Блок-схема 5.2.4):



*Метод CreateFirst в виде блок схемы (Блок-схема 5.2.4)*

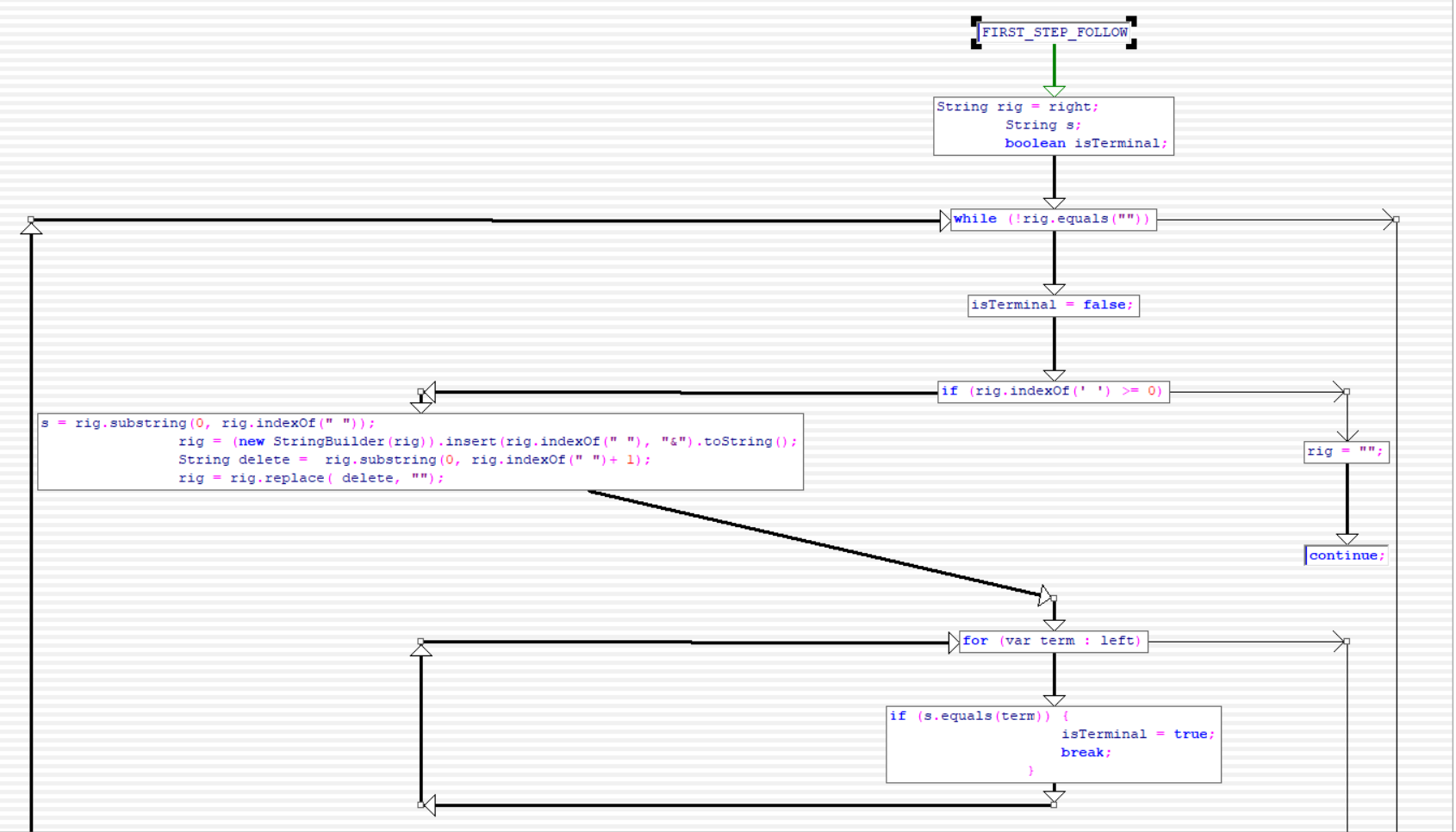
Во- первых функция добавляет первые символы либо слова в множество First

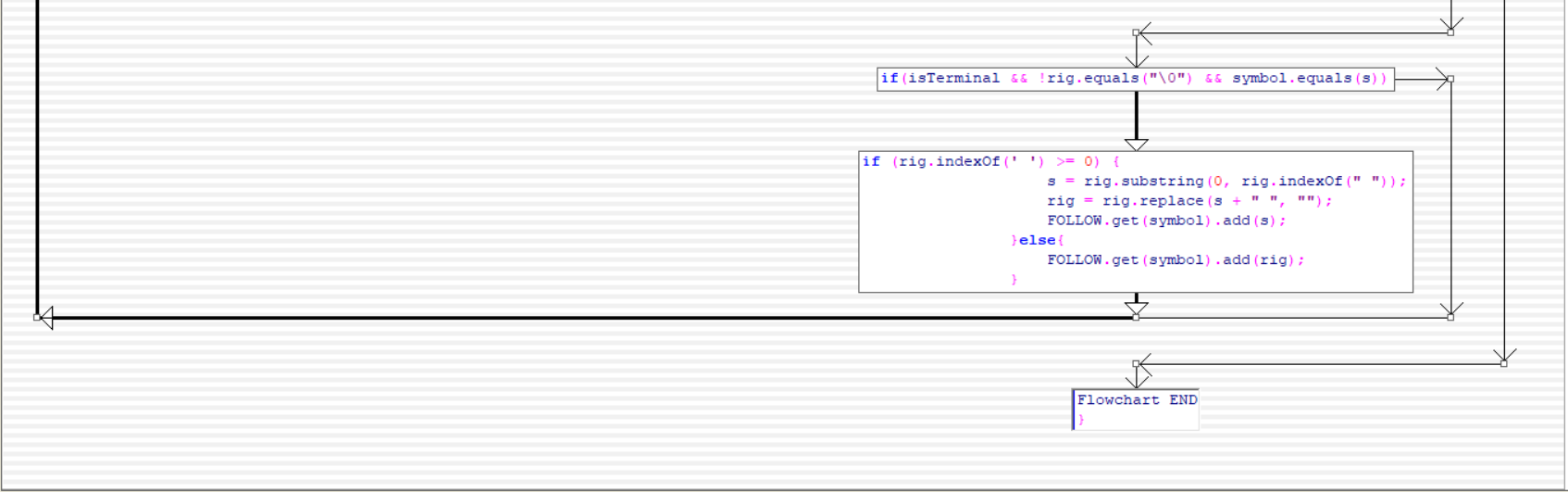
Во-вторых проверяет на наличие подстановок нетерминала в правила так, что если первый символ в правиле нетерминал, то к First этого нетерминала прибавляется First левого нетерминала.

Метод Create\_Follow(функция, создающая множество Follow)(Блок-схема 5.2.5): 

*Метод Create\_Follow в виде блок схемы (Блок-схема 5.2.6)*

Метод First\_Step\_Follow(Блок-схема 5.2.6):

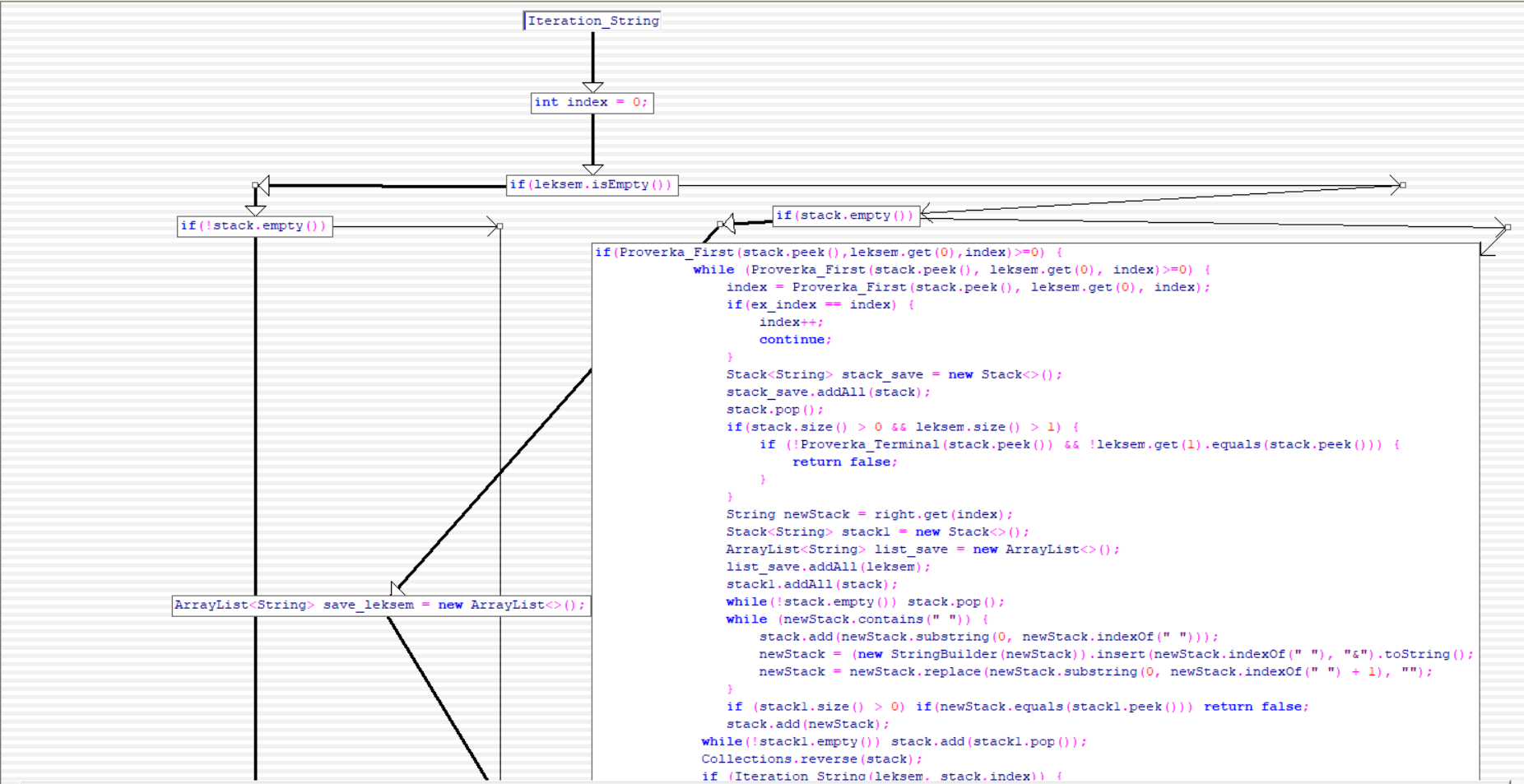


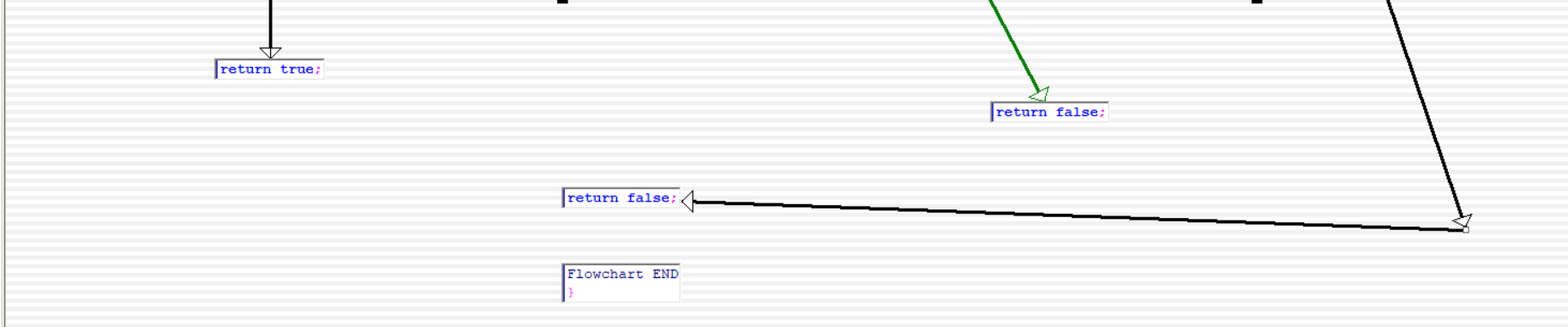
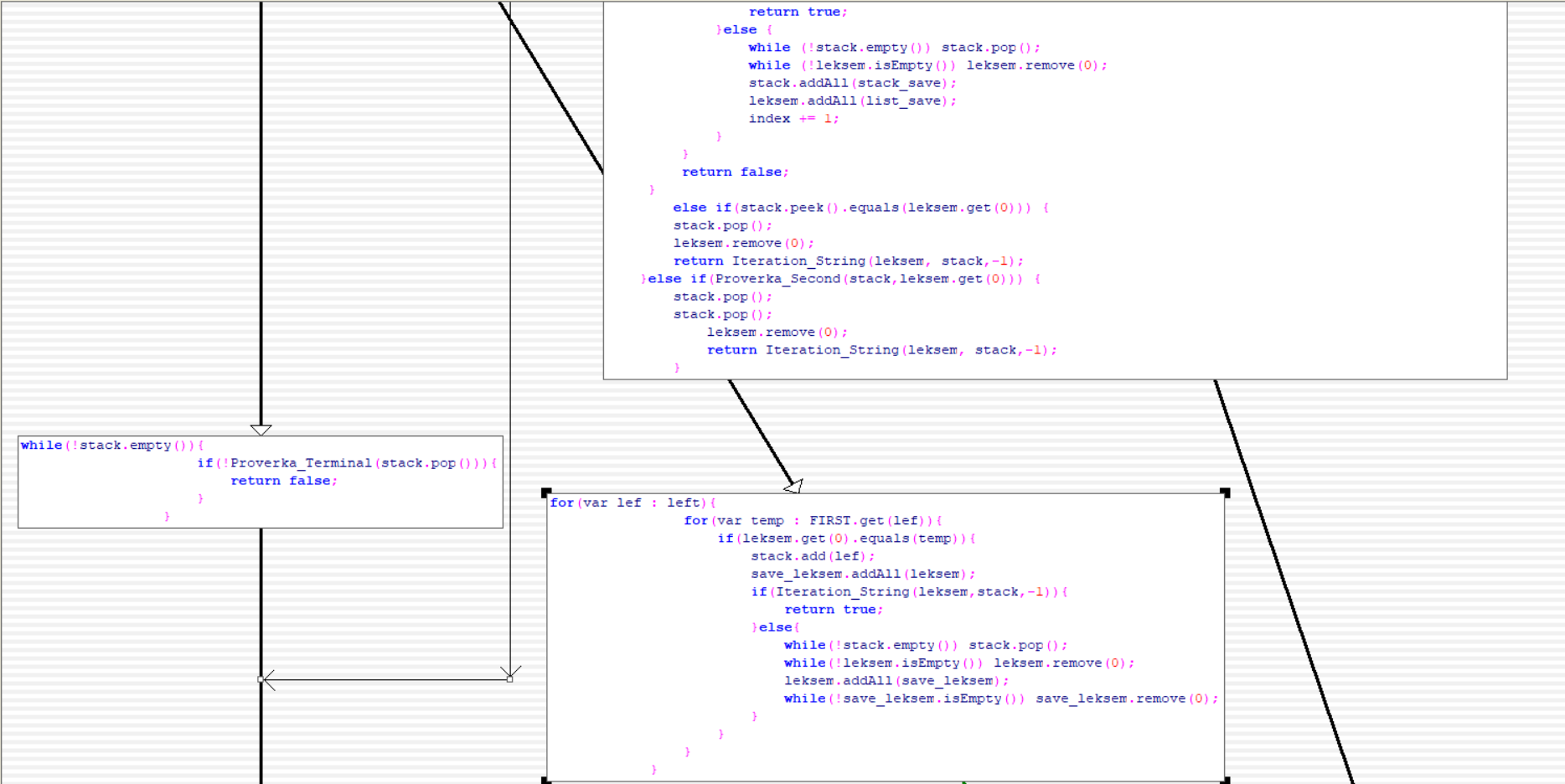


*Метод First\_Step\_Follow в виде блок схемы (Блок-схема 5.2.6)*

Функция First\_Step\_Follow выполняет первый шаг в построение множества, то есть добавляет все символы, которые идут после нетерминала во всех правилах.

Метод Iteration\_String(Блок-схема 5.2.7):



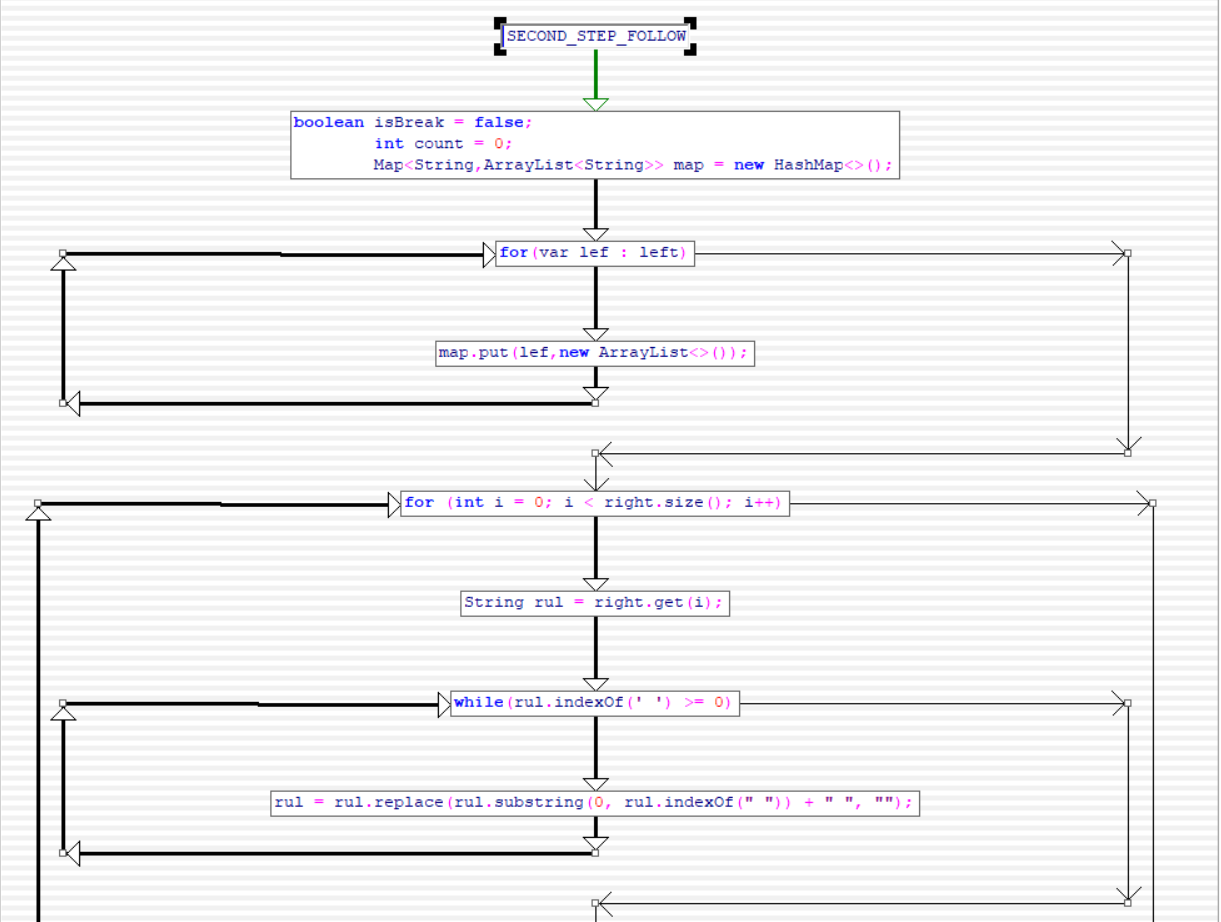


*Метод Iteration\_String в виде блок схемы (Блок-схема 5.2.7)*

Функция выполняет рекурсивный подбор необходимых правил, тем самым проверяя строку на правильность. Рекурсия происходит до момента, пока входной буфер не окажется пустым и в стеке не будет терминальных символов, то тогда возвращается true. Следующие проверки происходят следующим образом:

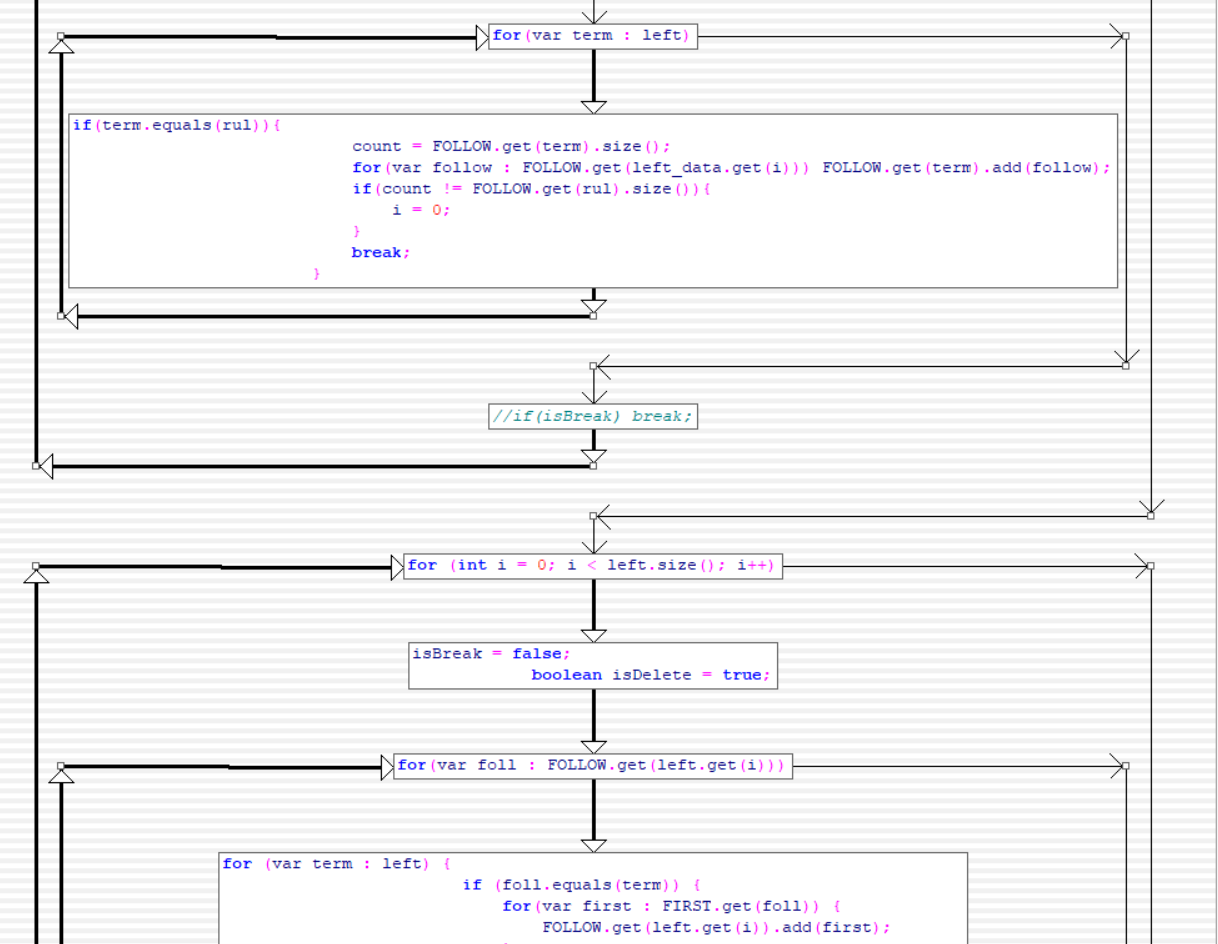
1. Проверяется есть ли символ из буфера ввода в множестве First, если есть, то добавляется в стек нетерминал
2. Проверяет есть ли в правилах грамматики, у которых в левой части находится нетерминал, лежащий на вершине стека, необходимое, если находит такое правило, то добавляет его в стек, вместо нетерминала
3. Проверяется равен ли символ в буфере символу в стеке, если да, то удаляется этот символ из буфера и стека
4. Проверяет есть ли в множестве Follow элемент в стеке, идущий после нетерминального символа, который находится на вершине стека. Если есть, то удаляется в стеке данный нетерминал и этот элемент, как в стеке, так и во входном буфере

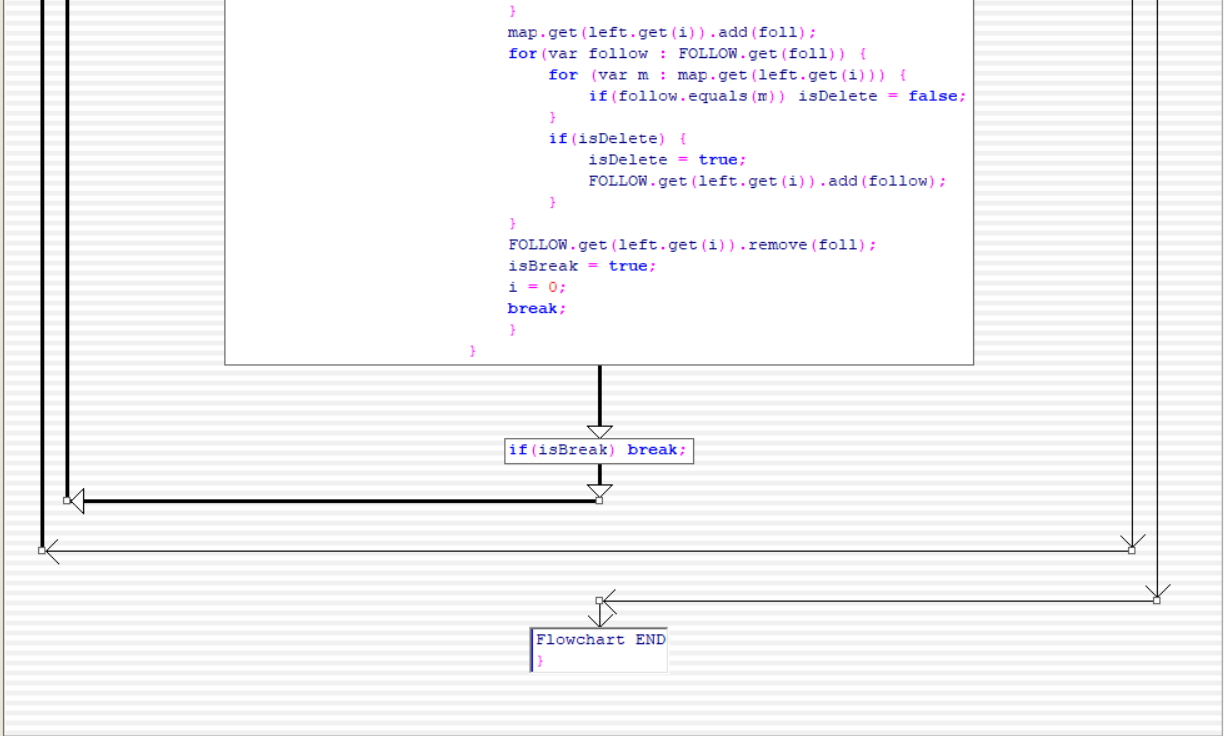
Метод Second\_Step\_Follow(Блок-схема 5.2.8):



A

A





*Метод Second\_Step\_Follow в виде блок схемы (Блок-схема 5.2.8)*

Данная функция отвечает за 3 шаг построения множества Follow: Для всех А ∈ VN вычислить: FOLLOW’i(1, A) = FOLLOWi(1, A) ∪ FIRST (1, B), ∀ B∈ FOLLOWi(1, A) ⋂ VN

Глава 6. Установка и эксплуатация программного средства

Для создания кода мною был выбран язык программирования Java и мною была использована интегрированная среда разработки(IDE) - IntelliJ IDEA Community Edition 2020.3.2 x64. Я использовал данную IDE для компиляции кода, поэтому лучше компилировать код именно в IntelliJ IDEA. Кроме этого, могут быть использованы и другие IDE, такие как NetBeans, Eclipse, JCreator и любые другие, поддерживающие язык программирования Java. Это касательно Windows, для операционной системы Linux, необходимо воспользоваться командой строкой и открыть jar-файл, с помощью команды: java -jar ./<название файла>.jar.

Программа состоит из 3 классов:

1)Leksema(размер файла 1КБ)

2)Main(размер файла 14 КБ)

3)Parser(размер файла 17 КБ)

Общий размер программы составляет – 32 КБ.

Глава 7. Работа с программным средством

В коде программы необходимо указывать имена файлов откуда будет взят текст кода и грамматика. На выводе программы получается вывод введенного текста через токены, а также после этого выведется строка построенная из этих токенов для того, чтобы проверить правильность работы кода. Следующие выводится работа синтаксического анализатора. Выводится все строки, которые анализировались и результат. Рядом со строкой будет написано true в случае правильности, в случае неправильности выводится false. После false выводит сообщение об ошибки, если ее удалось установить.

Ошибки, которые могут быть выведены:

1. Отсутствие точки с запятой
2. Неправильный тип данных

# Заключение

В данной курсовой работе был реализован компилятор, который способен выполнить синтаксический анализ, основанный на веденной пользователем грамматики правил. Кроме этого, компилятор выполняет лексический анализ, благодаря которому, изначальный текст введенного кода для анализа выводится в форме токенов, а также способен перевести строку из токенов в первоначальный вид. Итак, мною были выполнены все поставленные задачи и достигнута цель работы.

# Литературный список

1. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы. Принципы, технологии, инструментарий — М.: Вильямс, 2018. — 1184 с.
2. Альфред В. Ахо, Джеффри Д. Ульман. Теория синтаксического анализа, перевода и компиляции — М.: Мир, 1978 — 611 с.
3. Свердлов С.З. Конструирование компиляторов - Lambert Academic Publishing, 2015 – 575 с.
4. Н. Вирт. Разработка ОС и компилятора. Проект Оберон - ДМК Пресс, 2012. – 558 с.
5. Wikipedia: Компилятор. URL: <https://ru.wikipedia.org/wiki/Компилятор>. (Дата обращения 05.06.2021)
6. Wikipedia:Лексический анализатор URL: <https://ru.wikipedia.org/wiki/Лексический_анализ> (Дата обращения 05.06.2021)
7. Wikipedia Синтаксический анализатор URL: <https://ru.wikipedia.org/wiki/Синтаксический_анализатор> (Дата обращения 05.06.2021)

Приложение

Текст программы

import java.io.\*;  
import java.util.\*;  
import java.util.regex.\*;  
  
public class Main {  
 private static List<List<Leksema>> *leksems* = new ArrayList<>();  
 private static ArrayList<String> *source* = new ArrayList<>();  
  
  
 private static final List<String> *operations\_connection* = new ArrayList<>(Arrays.*asList*("< >","<=",">="," <"," >"," ="));  
 private static final List<String> *operations\_sum* = new ArrayList<>(Arrays.*asList*("\\+","\\-","or"));  
 private static final List<String> *operations\_multiplication* = new ArrayList<>(Arrays.*asList*("\\\*", "\\/", "and"));  
 private static final List<String> *operations\_unary* = new ArrayList<>(Arrays.*asList*("not"));  
  
 private static final List<String> *operations\_structure* = new ArrayList<>(Arrays.*asList*("program var","begin","end","\\{;","} end."));  
 private static final List<String> *operations\_type* = new ArrayList<>(Arrays.*asList*("int", "float", "bool"));  
 private static final List<String> *operations\_operators* = new ArrayList<>(Arrays.*asList*("ass","then","else","for","to","do","while",  
 "read","write"));  
 private static final List<String> *operations\_comments* = new ArrayList<>(Arrays.*asList*("\\{","\\}"));  
  
  
 private static final List<String> *separators* = new ArrayList(Arrays.*asList*("\\.", ";", ",", ":"));  
 private static List<List<String>> *symbChains* = new ArrayList(Arrays.*asList*( *separators*,*operations\_connection*,*operations\_sum*,  
 *operations\_multiplication*,*operations\_unary*,*operations\_structure*,*operations\_type*,*operations\_operators*,*operations\_comments*));  
  
 public static List<ArrayList<String>> *different\_symbols* = new ArrayList<>();  
  
  
 public static ArrayList<String> getList(String str){  
 String s = "";  
 ArrayList<String> Mylist = new ArrayList<>();  
 for (int i = 0; i < str.length(); i++) {  
 while(str.toCharArray()[i] != ' '){  
  
 s += str.toCharArray()[i];  
 if(i+1<str.length()){ i++; } else {break;}  
 if(str.toCharArray()[i] == '\0') break;  
 }  
 if (!s.equals("")) Mylist.add(s);  
 s = "";  
 }  
 return Mylist;  
 }  
  
  
 public static void getLexems(String source, int index){  
 Pattern pattern=null;  
 String text = source; // передаем строку, которую надо разбить на лексемы  
 int tableID = 0; // идентификатор таблицы  
 String different\_symbol = source;  
 String str;  
 boolean flag = false;  
 *leksems*.add(new ArrayList<>());  
 int count = 0;  
 for (int j = 0; j < *symbChains*.size() - index;j++) {  
  
 for (var i = 0; i <= *symbChains*.get(j).size() - 1 ; i++) {  
 //компилируем паттерн из табличных цепочек символов  
 switch (tableID) {  
 case 0:  
 case 1:  
 case 2:  
 case 8:  
 pattern = Pattern.*compile*("(" + *symbChains*.get(j).get(i) + "\\.|" +  
 *symbChains*.get(j).get(i) + ";|" +  
 *symbChains*.get(j).get(i) + ",|" +  
 *symbChains*.get(j).get(i) + ":|" +  
 *symbChains*.get(j).get(i) + " |" +  
 *symbChains*.get(j).get(i) + "\n|" +  
 *symbChains*.get(j).get(i) + "$)");  
 break;  
 default:  
 pattern = Pattern.*compile*("\\b(" + *symbChains*.get(j).get(i) + "\\.\\b|\\b" +  
 *symbChains*.get(j).get(i) + "\\b;|\\b" +  
 *symbChains*.get(j).get(i) + "\\b,|\\b" +  
 *symbChains*.get(j).get(i) + "\\b:|\\b" +  
 *symbChains*.get(j).get(i) + "\\b |\\b" +  
 *symbChains*.get(j).get(i) + "\n|" +  
 *symbChains*.get(j).get(i) + "$)");  
 break;  
 }  
  
 //сканируем строку  
 Matcher matcher = pattern.matcher(text);  
 //В цикле перебираем результат сканирования и выводим информацию о лексемах  
 while (matcher.find()) {  
 int start = matcher.start();  
 int end = 0;  
  
 switch (tableID) {  
 default:  
 if (matcher.end() != source.length()) {  
 end = matcher.end() - 1;  
 } else {  
 end = matcher.end();  
 }  
 break;  
 case 0:  
 end = matcher.end();  
 }  
 ;  
 switch (tableID) {  
 case 0:  
 System.*out*.print("Найден разделитель ");  
 break;  
 case 1:  
 case 2:  
 case 3:  
 case 4:  
 System.*out*.print("Найдена операция языка ");  
 break;  
  
 case 5:  
 System.*out*.print("Найдено правило, определяющее структуру программы ");  
 break;  
  
 case 6:  
 System.*out*.print("Найдена переменная ");  
 break;  
  
 case 7:  
 System.*out*.print("Найден тип данных ");  
 break;  
  
 case 8:  
 case 9:  
 case 10:  
 case 11:  
 case 12:  
 case 13:  
 case 14:  
 case 15:  
 System.*out*.print("Найдено правило, определяющее оператор программы ");  
 break;  
 case 16:  
 System.*out*.print("Найдено правило, определяющее коментарий программы ");  
 break;  
 }  
  
 System.*out*.println(text.substring(start, end) + " с " + start + " по " + (end - 1) + " позицию." + " Код лексемы: " + "(" + tableID + ", " + i + ")");  
 *leksems*.get(index).add(new Leksema(tableID, i, start, end));  
 }  
 different\_symbol = different\_symbol.replaceAll(*symbChains*.get(j).get(i), "");  
 }  
 tableID++; //Увеличиваем идентификатор таблицы корректных цепочек символов на 1  
  
 count ++;  
 }  
 ArrayList<String> Dif\_symb = *getList*(different\_symbol);  
 *different\_symbols*.add(Dif\_symb);  
  
 Set<String> set = new HashSet<>(*different\_symbols*.get(index));  
 *different\_symbols*.get(index).clear();  
 *different\_symbols*.get(index).addAll(set);  
 *symbChains*.add(*different\_symbols*.get(index));  
  
  
 List<String> symbChain = *different\_symbols*.get(index);  
 for (int i=0; i<symbChain.size(); i++) {  
  
 //компилируем паттерн из табличных цепочек символов  
 pattern = Pattern.*compile*("\\b(" + symbChain.get(i) + "\\.|\\b" +  
 symbChain.get(i) + "\\b;|\\b" +  
 symbChain.get(i) + "\\b,|\\b" +  
 symbChain.get(i) + "\\b:|\\b" +  
 symbChain.get(i) + "\\b |\\b" +  
 symbChain.get(i) + "\\b$)");  
 //сканируем строку  
 Matcher matcher = pattern.matcher(text);  
 //В цикле перебираем результат сканирования и выводим информацию о лексемах  
 while (matcher.find()) {  
 int start = matcher.start();  
 int end = 0;  
 if (matcher.end() != source.length()) {  
 end = matcher.end() - 1;  
 } else {  
 end = matcher.end();  
 }  
  
  
 System.*out*.println("Найден незазервированный символ " + text.substring(start, end) + " с " + start + " по " + (end - 1) + " позицию." + " Код лексемы: " + "(" + tableID + ", " + i + ")");  
  
 *leksems*.get(index).add(new Leksema(tableID+index, i, start, end));  
 }  
 }  
  
 }  
  
public static String getSorce(List<Leksema> leksems){  
// for (int i = 0; i < leksems.size(); i++) {  
// for (int j = i; j < leksems.size(); j++) {  
// if(leksems.get(i).getBeginId() < leksems.get(j).getBeginId()){  
// Leksema leksema = leksems.get(i);  
//  
// leksems.add(i,leksems.get(j));  
// leksems.remove(i);  
//  
// leksems.add(j, leksema);  
// leksems.remove(j);  
// }  
// }  
  
  
 // }  
  
 Collections.*sort*(leksems, new Comparator<Leksema>() {  
 @Override  
 public int compare(Leksema o1, Leksema o2) {  
 return o1.getBeginId() > o2.getBeginId() ? 1:o1.getBeginId() < o2.getBeginId() ? -1 : 0;  
 }  
 });  
  
 String result = "";  
 for (int i = 0; i < leksems.size(); i ++){  
 if(i == leksems.size() - 1){  
  
 result += *symbChains*.get(leksems.get(i).getTableId()).get(leksems.get(i).getWordId());  
  
  
 }else if (leksems.get(i+1).getTableId() == 0) {  
  
 result += *symbChains*.get(leksems.get(i).getTableId()).get(leksems.get(i).getWordId());  
  
 }else {  
  
 result += *symbChains*.get(leksems.get(i).getTableId()).get(leksems.get(i).getWordId())+" ";  
  
 }  
 }  
  
 result = result.replaceAll("\\\\", "");  
 result = result.replaceAll(" ", " ");  
 return result;  
}  
  
 public static String getSorceLeksem(ArrayList<Leksema> leksems){  
  
  
 Collections.*sort*(leksems, new Comparator<Leksema>() {  
 @Override  
 public int compare(Leksema o1, Leksema o2) {  
 return o1.getBeginId() > o2.getBeginId() ? 1:o1.getBeginId() < o2.getBeginId() ? -1 : 0;  
 }  
 });  
  
 String result = "";  
 for (int i = 0; i < leksems.size(); i ++){  
 if(i == leksems.size() - 1){  
 result += "(" + leksems.get(i).getTableId() + "," + leksems.get(i).getWordId() + ")";  
 }else if (leksems.get(i+1).getTableId() == 0) {  
 result += "(" + leksems.get(i).getTableId() + "," + leksems.get(i).getWordId() + ")";  
 }else {  
 result += "(" + leksems.get(i).getTableId() + "," + leksems.get(i).getWordId() + ")" + " ";  
 }  
 }  
  
 result = result.replaceAll("\\\\", "");  
 result = result.replaceAll(" ", " ");  
 return result;  
 }  
  
 public static void main(String[] args) throws IOException {  
 System.*out*.println("------------------ цепочки символов, которые не являются зарезервированными словами,разделителями или ограничителями -----------------------");  
  
 File file = new File("cod.txt");  
 FileReader fr = new FileReader(file);  
 BufferedReader reader = new BufferedReader(fr);  
 int index = 0;  
 String line = reader.readLine();  
  
 while (line != null) {  
 *source*.add(line);  
 line = reader.readLine();  
 System.*out*.println("------------------ цепочки символов, которые не являются зарезервированными словами,разделителями или ограничителями в строке №"+ index + "-----------------------");  
  
 *getLexems*(*source*.get(index),index);  
 index++;  
 }  
 reader.close();  
  
 for(int i = 0; i < *source*.size();i++) {  
 System.*out*.println();  
 System.*out*.println(*getSorceLeksem*((ArrayList<Leksema>) *leksems*.get(i)));  
 System.*out*.println(*getSorce*(*leksems*.get(i)));  
 }  
  
  
 Parser parser = new Parser();  
 parser.Parsing(*source*);  
  
 }  
}

public class Test {  
 public static void main(String[] args) {  
 Stack<String> stack = new Stack<>();  
 stack.add("Dimas");  
 stack.add("St");  
 stack.add("23");  
 Collections.reverse(stack);  
 while(!stack.empty()){  
 System.out.println(stack.pop());  
 }  
 }  
}

public class Parser {  
 public static Map<String, ArrayList<String>> *rules* = new HashMap<>();  
 public static Set<String> *left\_set* = new LinkedHashSet<>();  
 public static ArrayList<String> *left\_data* = new ArrayList<>();  
 public static ArrayList<String> *left* = new ArrayList<>();  
 public static ArrayList<String> *right* = new ArrayList<>();  
 public static ArrayList<String> *terminal* = new ArrayList<>();  
 public static ArrayList<String> *neterminal* = new ArrayList<>();  
 public static Map<String, Set<String>> *FIRST* = new HashMap<>();  
 public static Map<String, Set<String>> *FOLLOW* = new HashMap<>();  
 public static ArrayList<String> *leksems* = new ArrayList<>();  
  
  
 public void Parser(ArrayList<String> list){  
 *leksems*.addAll(list);  
 }  
 public void Parsing(ArrayList<String> scanner\_leskems) throws IOException {  
 *leksems*.addAll(scanner\_leskems);  
 File file = new File("rules.txt");  
 FileReader fr = new FileReader(file);  
 BufferedReader reader = new BufferedReader(fr);  
 String s = reader.readLine();  
 *neterminal*.addAll(*left*);  
 String left\_str = "";  
 char[] chars = s.toCharArray();  
 while (s != null) {  
  
 for (int i = 0; chars[i] != '-'; i++) {  
 left\_str += chars[i];  
 }  
  
 *rules*.put(left\_str, new ArrayList<String>());  
 *left\_data*.add(left\_str);  
 *left\_set*.add(left\_str);  
 s = reader.readLine();  
 if (s != null) chars = s.toCharArray();  
 left\_str = "";  
 }  
 *AddRight*();  
 *left*.addAll(*left\_set*);  
 *neterminal*.addAll(*left\_set*);  
 for (int i = 0; i < *rules*.size(); i++) {  
 for (int j = 0; j < *rules*.get(*left*.get(i)).size(); j++) {  
 System.*out*.println(*left*.get(i) + "->" + *rules*.get(*left*.get(i)).get(j));  
 }  
 }  
 *AddTerminal*();  
 *CreateFirst*();  
 *Create\_Follow*();  
 String str = "";  
 ArrayList<String> strings = new ArrayList<>();  
  
 Stack<String> stack = new Stack<>();  
 for (int i = 0; i < *leksems*.size(); i++) {  
 str = *leksems*.get(i);  
 while (str.contains(" ")) {  
 strings.add(str.substring(0, str.indexOf(" ")));  
 str = (new StringBuilder(str)).insert(str.indexOf(" "), "&").toString();  
 str = str.replace(str.substring(0, str.indexOf(" ") + 1), "");  
 }  
 strings.add(str);  
 boolean result = *Iteration\_String*(strings,stack,-1);  
 str = *leksems*.get(i);  
 System.*out*.println(str + " - " + result);  
 if(!result){  
  
 Pattern pattern = Pattern.*compile*(".\* .\*;");  
 Matcher matcher = pattern.matcher(str);  
 if(matcher.find()){  
 System.*out*.println("Задан нейзвестный тип данных, возможно вы имелли вииду (int,float,bool)");  
 }  
 pattern = Pattern.*compile*("\\w\* [int float bool]+");  
 matcher = pattern.matcher(str);  
 if(matcher.find() && !str.contains(";") && matcher.start() == 0 && matcher.end() == str.length()){  
 System.*out*.println("Пропущена ;");  
 }  
 }  
 stack.clear();  
 strings.clear();  
 }  
  
  
  
  
 }  
  
 public static void CreateFirst() {  
  
 for (int i = 0; i < *left*.size(); i++) {  
 *FIRST*.put(*left*.get(i),new LinkedHashSet<>());  
 }  
  
  
  
 for (int i = 0; i < *FIRST*.size(); i++) {  
 for (int j = 0; j < *left\_data*.size(); j++) {  
  
 if(*right*.get(j).indexOf(' ') >=0)  
 *FIRST*.get(*left\_data*.get(j)).add(*right*.get(j).substring(0, *right*.get(j).indexOf(" ")));  
 else  
 *FIRST*.get(*left\_data*.get(j)).add(*right*.get(j));  
  
 }  
 }  
  
 while(!*Proverka\_First*()){  
 boolean End = false;  
 for (int i = 0; i < *left*.size(); i++) {  
 for (String first : *FIRST*.get(*left*.get(i))) {  
 for (int k = 0; k < *neterminal*.size(); k++) {  
 if(first.equals(*neterminal*.get(k))){  
 *FIRST*.get(*left*.get(i)).remove(*neterminal*.get(k));  
 *Change\_First*(*neterminal*.get(k),*left*.get(i));  
 End = true;  
 }  
 }  
 if(End) break;  
 }  
 }  
 }  
 }  
  
  
 static void Create\_Follow(){  
 for (int i = 0; i < *left*.size(); i++) {  
 *FOLLOW*.put(*left*.get(i),new LinkedHashSet<>());  
 }  
 String s;  
  
 for (int i = 0; i < *left*.size(); i++) {  
 for (int j = 0; j < *right*.size(); j++) {  
 *FIRST\_STEP\_FOLLOW*(*right*.get(j), *left*.get(i));  
 }  
 }  
  
 *SECOND\_STEP\_FOLLOW*();  
 }  
 static boolean Iteration\_String( ArrayList<String> leksem, Stack<String> stack,int ex\_index){  
 int index = 0;  
  
// if(stack.size() > 50){  
// System.out.println("false");  
// System.exit(0);  
// }  
  
 if(leksem.isEmpty()){  
 if(!stack.empty()){  
 while(!stack.empty()){  
 if(!*Proverka\_Terminal*(stack.pop())){  
 return false;  
 }  
 }  
 }  
 return true;  
  
  
  
 }else if(stack.empty()){  
 ArrayList<String> save\_leksem = new ArrayList<>();  
 for(var lef : *left*){  
 for(var temp : *FIRST*.get(lef)){  
 if(leksem.get(0).equals(temp)){  
 stack.add(lef);  
 save\_leksem.addAll(leksem);  
 if(*Iteration\_String*(leksem,stack,-1)){  
 return true;  
 }else{  
 while(!stack.empty()) stack.pop();  
 while(!leksem.isEmpty()) leksem.remove(0);  
  
 leksem.addAll(save\_leksem);  
 while(!save\_leksem.isEmpty()) save\_leksem.remove(0);  
 }  
 }  
 }  
 }  
 return false;  
  
  
 }else if(*Proverka\_First*(stack.peek(),leksem.get(0),index)>=0) {  
 while (*Proverka\_First*(stack.peek(), leksem.get(0), index)>=0) {  
 index = *Proverka\_First*(stack.peek(), leksem.get(0), index);  
 if(ex\_index == index) {  
 index++;  
 continue;  
 }  
  
 Stack<String> stack\_save = new Stack<>();  
 stack\_save.addAll(stack);  
 stack.pop();  
 if(stack.size() > 0 && leksem.size() > 1) {  
 if (!*Proverka\_Terminal*(stack.peek()) && !leksem.get(1).equals(stack.peek())) {  
 return false;  
 }  
 }  
 String newStack = *right*.get(index);  
 Stack<String> stack1 = new Stack<>();  
  
 ArrayList<String> list\_save = new ArrayList<>();  
 list\_save.addAll(leksem);  
  
 stack1.addAll(stack);  
 while(!stack.empty()) stack.pop();  
  
 while (newStack.contains(" ")) {  
 stack.add(newStack.substring(0, newStack.indexOf(" ")));  
 newStack = (new StringBuilder(newStack)).insert(newStack.indexOf(" "), "&").toString();  
 newStack = newStack.replace(newStack.substring(0, newStack.indexOf(" ") + 1), "");  
 }  
 if (stack1.size() > 0) if(newStack.equals(stack1.peek())) return false;  
 stack.add(newStack);  
 while(!stack1.empty()) stack.add(stack1.pop());  
 Collections.*reverse*(stack);  
  
  
  
 if (*Iteration\_String*(leksem, stack,index)) {  
 return true;  
 }else {  
 while (!stack.empty()) stack.pop();  
 while (!leksem.isEmpty()) leksem.remove(0);  
 stack.addAll(stack\_save);  
 leksem.addAll(list\_save);  
 index += 1;  
 }  
 }  
 return false;  
 }  
  
  
  
 else if(stack.peek().equals(leksem.get(0))) {  
 stack.pop();  
 leksem.remove(0);  
 return *Iteration\_String*(leksem, stack,-1);  
  
  
 }else if(*Proverka\_Second*(stack,leksem.get(0))) {  
 stack.pop();  
 stack.pop();  
 leksem.remove(0);  
 return *Iteration\_String*(leksem, stack,-1);  
 }  
 return false;  
 }  
  
 static boolean Proverka\_Second(Stack<String> stack, String leksema){  
 Stack<String> stack1 = new Stack<>();  
 stack1.addAll(stack);  
 String first\_element = stack1.pop();  
  
 if(*Proverka\_Terminal*(first\_element)){  
 if(stack1.empty()) return false;  
 for(var temp : *FOLLOW*.get(first\_element)){  
 if(stack1.peek().equals(temp) && leksema.equals(stack1.peek())){  
 return true;  
 }  
  
 }  
 }  
 return false;  
  
 }  
  
 static boolean Proverka\_Terminal(String s){  
 for (var lef : *left*){  
 if(lef.equals(s)) return true;  
 }  
 return false;  
 }  
  
  
  
 static int Proverka\_First(String s, String leksema, int index){  
 String righ;  
 for(int i = index; i < *left\_data*.size();i++){  
 if(*right*.get(i).contains(" ")){  
 righ = *right*.get(i).substring(0,*right*.get(i).indexOf(" "));  
 }else{  
 righ = *right*.get(i);  
 }  
 if(s.equals(*left\_data*.get(i)) && righ.equals(leksema)){  
 index = i;  
 return index;  
 }else if(s.equals(*left\_data*.get(i)) && *Proverka\_Terminal*(righ)){  
 for (var temp : *FIRST*.get(righ)) {  
 if(temp.equals(leksema)) {  
 index = i;  
 return index;  
 }  
 }  
 }  
 }  
  
  
 return -1;  
 }  
  
 static void SECOND\_STEP\_FOLLOW(){  
 boolean isBreak = false;  
 int count = 0;  
 Map<String,ArrayList<String>> map = new HashMap<>();  
 for(var lef : *left*) map.put(lef,new ArrayList<>());  
 for (int i = 0; i < *right*.size(); i++) {  
 String rul = *right*.get(i);  
 while(rul.indexOf(' ') >= 0) {  
 rul = rul.replace(rul.substring(0, rul.indexOf(" ")) + " ", "");  
 }  
  
 for(var term : *left*){  
 if(term.equals(rul)){  
 count = *FOLLOW*.get(term).size();  
 for(var follow : *FOLLOW*.get(*left\_data*.get(i))) *FOLLOW*.get(term).add(follow);  
 if(count != *FOLLOW*.get(rul).size()){  
 i = 0;  
 }  
 break;  
 }  
 }  
 //if(isBreak) break;  
 }  
  
  
 for (int i = 0; i < *left*.size(); i++) {  
 isBreak = false;  
 boolean isDelete = true;  
 for(var foll : *FOLLOW*.get(*left*.get(i))) {  
 for (var term : *left*) {  
 if (foll.equals(term)) {  
 for(var first : *FIRST*.get(foll)) {  
 *FOLLOW*.get(*left*.get(i)).add(first);  
 }  
 map.get(*left*.get(i)).add(foll);  
 for(var follow : *FOLLOW*.get(foll)) {  
 for (var m : map.get(*left*.get(i))) {  
 if(follow.equals(m)) isDelete = false;  
 }  
 if(isDelete) {  
 isDelete = true;  
 *FOLLOW*.get(*left*.get(i)).add(follow);  
 }  
 }  
 *FOLLOW*.get(*left*.get(i)).remove(foll);  
 isBreak = true;  
 i = 0;  
 break;  
 }  
 }  
 if(isBreak) break;  
 }  
 }  
 }  
  
  
 static void FIRST\_STEP\_FOLLOW(String right, String symbol) {  
 String rig = right;  
 String s;  
 boolean isTerminal;  
 while (!rig.equals("")) {  
 isTerminal = false;  
 if (rig.indexOf(' ') >= 0) {  
 s = rig.substring(0, rig.indexOf(" "));  
 rig = (new StringBuilder(rig)).insert(rig.indexOf(" "), "&").toString();  
 String delete = rig.substring(0, rig.indexOf(" ")+ 1);  
 rig = rig.replace( delete, "");  
 } else {  
 rig = "";  
 continue;  
 }  
 for (var term : *left*) {  
 if (s.equals(term)) {  
 isTerminal = true;  
 break;  
 }  
 }  
  
 if(isTerminal && !rig.equals("\0") && symbol.equals(s)){  
 if (rig.indexOf(' ') >= 0) {  
 s = rig.substring(0, rig.indexOf(" "));  
 rig = rig.replace(s + " ", "");  
 *FOLLOW*.get(symbol).add(s);  
 }else{  
 *FOLLOW*.get(symbol).add(rig);  
 }  
 }  
  
 }  
 }  
  
 static void Change\_First(String neterminal, String left){  
  
 for(String first : *FIRST*.get(neterminal)) {  
 if(!neterminal.equals(first)) {  
 *FIRST*.get(left).add(first);  
 }  
 }  
  
 }  
  
 static boolean Proverka\_First(){  
  
 for (int i = 0; i < *left*.size(); i++) {  
 for(String first : *FIRST*.get(*left*.get(i))) {  
 for (int k = 0; k < *neterminal*.size(); k++) {  
 if(first.equals(*neterminal*.get(k))) return false;  
 }  
 }  
  
 }  
 return true;  
 }  
  
 public static void AddTerminal() {  
 boolean isTerminal;  
 int index;  
 for (var righ : *right*) {  
 String s;  
 String rig;  
 rig = righ;  
 while (!rig.equals("")) {  
 isTerminal = true;  
 if (rig.indexOf(' ') >= 0) {  
 s = rig.substring(0, rig.indexOf(" "));  
 rig = rig.replace(s + " ", "");  
 } else {  
 s = rig;  
 rig = rig.replace(s, "");  
 }  
  
 for (var term : *left*) {  
 if (s.equals(term)) {  
 isTerminal = false;  
 break;  
 }  
 }  
 if (isTerminal) *terminal*.add(s);  
 }  
 }  
 }  
  
  
 public static void AddRight() throws IOException {  
 File file = new File("rules.txt");  
 FileReader fr = new FileReader(file);  
 BufferedReader reader = new BufferedReader(fr);  
 String s = reader.readLine();  
 char[] chars = s.toCharArray();  
 String left\_str = "";  
 while (s != null) {  
 for (int i = 0; chars[i] != '-'; i++) {  
 left\_str += chars[i];  
 }  
 String right\_line = "";  
 for (int i = left\_str.length() + 2; i < s.length(); i++) {  
 right\_line += chars[i];  
 }  
 String str;  
 String rig = right\_line;  
 while (!rig.equals("")) {  
  
 if (rig.indexOf(' ') >= 0) {  
 str = rig.substring(0, rig.indexOf(" "));  
 rig = rig.replace(str + " ", "");  
 } else {  
 str = rig;  
 rig = rig.replace(str, "");  
 }  
  
 *rules*.get(left\_str).add(str);  
 }  
 *right*.add(right\_line);  
 s = reader.readLine();  
 if (s != null) chars = s.toCharArray();  
 left\_str = "";  
 }  
 }  
}

Контрольный пример

1. Входная строка «a int» - false

Ошибка: Пропущена «;»

2)Входная строка «b boole;» - false

Ошибка: Задан неизвестный тип данных, возможно вы имели виду (int,float,bool)

3) Входная строка «float;» - true

4)Входная строка «program var n begin» - false (Примечание: false, потому что отсутствует продолжение правила, то есть строка введена не до конца, но ошибку на экран синтаксический анализатор не выведет)

Скриншот работы

