МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра вычислительной техники

**ОТЧЕТ**

**ПО ЛАБОРАТОРНОЙ РАБОТЕ №4**

**ПО ДИСЦИПЛИНЕ «ТЕОРИЯ ФОРМАЛЬНЫХ ЯЗЫКОВ И КОМПИЛЯТОРОВ»**

**НА ТЕМУ**

**«Синтаксис языков программирования. Нисходящий синтаксический анализ»**

Факультет: АВТФ Преподаватель: Малявко А.А.

Группа: АВТ-709

Студент: Кузнецов А.В.

Вариант: 33312212

Новосибирск 2020 г.

**Цель работы**

Изучение основных идей и понятий нисходящих методов синтаксического анализа, выявление свойств формальных грамматик, необходимых для реализации нисходящего восстановления дерева грамматического разбора, приобретение навыков построения процедурной и различных автоматных реализаций нисходящего анализа, исследование поведения нисходящих синтаксических акцепторов.

**Задание**

* 1. Используя пакет ВебТрансЛаб:

-     расширить грамматику заданного на курсовую работу языка до полной;

-     изучить и освоить проверку принадлежности грамматики к классу **LL1**, используя в качестве проверяемых грамматики, полученные при выполнении работы №4;

-     построить конечный автомат со стековой памятью и несколькими состояниями (шаблон …SyntAsMultiFSM…), разобраться в структуре управляющей таблицы автомата, уяснить способы формирования и использования всех полей;

-     построить конечный автомат со стековой памятью и одним состоянием, управляемый входным символом и символом, снятым с верхушки стека (шаблон …SyntAsSingleFSM…), разобраться в структуре управляющей таблицы автомата, уяснить способы формирования и использования клеток таблицы;

-     построить процедурную реализацию рекурсивного спуска (шаблон …SyntAsRD…), уяснить способы формирования функций этого акцептора.

2. Выполнить трассировку процессов нисходящего синтаксического акцепта, изучить поведение всех построенных синтаксических акцепторов при разборе как правильных предложений, так и предложений с намеренно внесенными синтаксическими ошибками.

3.   Проанализировать и сравнить между собой все полученные тексты программ и результаты выполнения пункта 2.2. Оценить степень пригодности изученных вариантов реализации нисходящих синтаксических акцепторов для выполнения курсовой работы.

**Грамматика LL(1)**

Обозначение LL(1) означает, что строки разбираются слева направо (первая L) и используются самые левые выводы (вторая L), а цифра 1 – что варианты порождающих правил выбираются с помощью одного предварительного просмотренного символа.

LL(1) — [LL-анализатор](https://ru.wikipedia.org/wiki/LL-%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%82%D0%BE%D1%80), [нисходящий алгоритм синтаксического разбора](https://ru.wikipedia.org/wiki/%D0%9D%D0%B8%D1%81%D1%85%D0%BE%D0%B4%D1%8F%D1%89%D0%B8%D0%B9_%D1%81%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7). Цифра 1 говорит, что для определения пути разбора нужна всего одна [лексема](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7). Очень быстр в исполнении и имеет характерное сообщение об ошибке вида «ожидался такой-то символ».

**Для каждого**[**нетерминала**](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D1%82%D0%B5%D1%80%D0%BC%D0%B8%D0%BD%D0%B0%D0%BB)**A в**[**грамматике**](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)**генерируется множество терминалов First(A), определенное следующим образом:**

* Если в грамматике есть правило с A в левой части и правой частью, начинающейся с терминала, то данный терминал входит в First(A)
* Если в грамматике есть правило с A в левой части и правой частью, начинающейся с нетерминала (обозначим B), то First(B) строго входит в First(A)
* Никакие иные терминалы не входят в First(A)

**Множества выбора**

Множества выбора – это множество терминальных символов, с которого начинается правая часть любого правила. Для каждого правила это множество уникально, при пересечении этих множеств грамматика перестает принадлежать к LL(1) грамматикам.

При этом возможна ситуация, когда правая часть правила пуста. Тогда множеством выбора будет являться то, что может следовать после данного правила.

Также правая часть правила может начинаться с нетерминала. Множеством выбора такого правила будут терминалы из правой части этого нетерминала. Пример:

*A: X*

*X: const*

Тогда множество для *А: const*

**Ход работы**

В работе была доработана грамматика языка, предлагаемого вариантом задания на курсовую работу.

**Определение программы:**

Конструкция определения программы выглядит следующим образом:

Prog ProgName [Functions]

Где Prog – специальное слово, ProgName –некоторое название функции, соответствующее myid, а functions – любое количество функций программы.

**Определение функции:**

Конструкция определения функции выглядит следующим образом:

TYPE funcname (TYPE arg1, TYPE arg2){

<<внутренние операторы>>

};

**Вызов функции:**

func funcname (arg1, arg2)

Где funcname принимает значение main или myid (F\_gh и тд. По варианту) а func – специальное слово.

Под внутренними операторами подразумевается различные блоки выполнения программы, такие как:

Оператор присваивания;

Оператор цикла;

Оператор переключателя

Условные операторы

А также их наборы, называемые блоками.

При этом в блоках операторах могут выполняться другие блоки операторы. При обработке программы со вложенными блоками операторами, вся грамматика обрабатывается последовательно, цепляясь за синтаксический блок и следующий за ним; если блок не укладывается в условия грамматики, то выпадает ошибка. Это происходит из-за того, что программа не ожидает подобного блока в данной позиции.

Таким образом, любая программа в данном языке будет состоять из заголовка программы (Prog Progname) и набора различных функций, внутри которых находятся различные операторы.

**Флаги управления операциями:**

С каждым состоянием должны быть связаны операции управления стековой памятью (занесение адреса возврата, снятие адреса с верхушки стека и переключение в состояние возврата) и операция управления чтением следующего входного символа. Все данные операции управления могут задаваться булевскими переменными, которые и обозначены флажками:

* флаг a управляет чтением следующего входного символа;
* флаг s управляет занесением адреса точки возврата в стек;
* флаг r обеспечивает переключение автомата в состояние, номер которого снимается с верхушки стека возвратов;
* флаг e запрещает останов по ошибке, когда состояние соответствует нетерминалу из левой части и есть еще хотя бы одно правило для такого нетерминала.

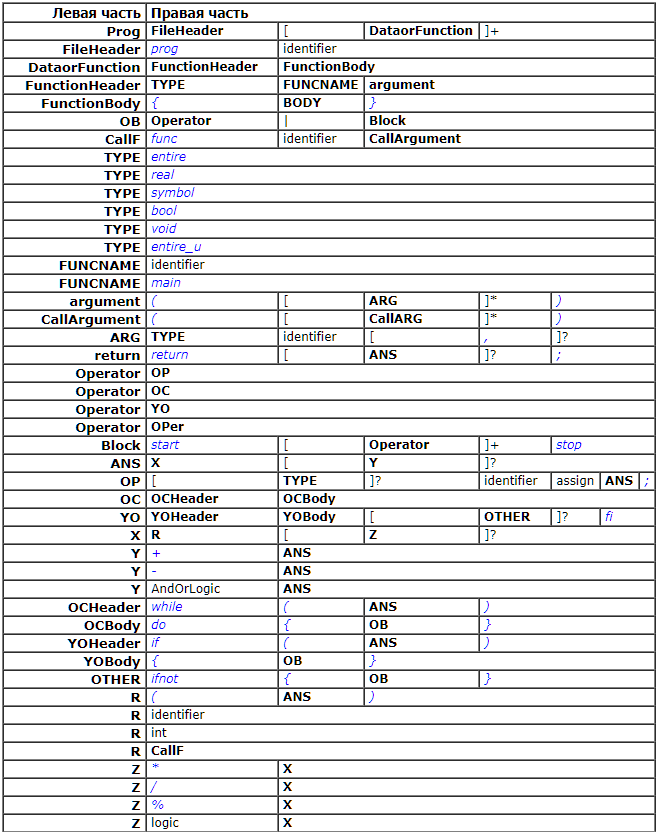


Рис. 5. Синтаксис языка.

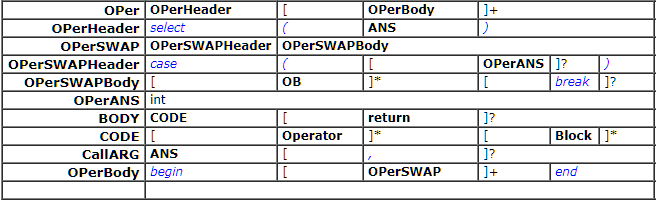


Рис. 6. Синтаксис языка.

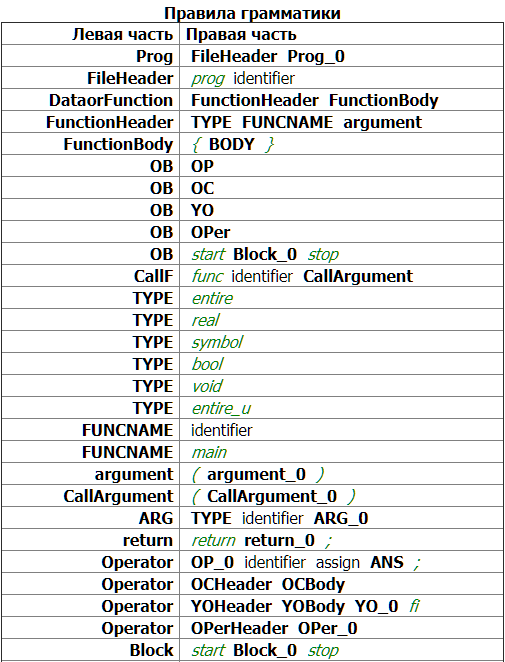


Рис. 7. Грамматика языка

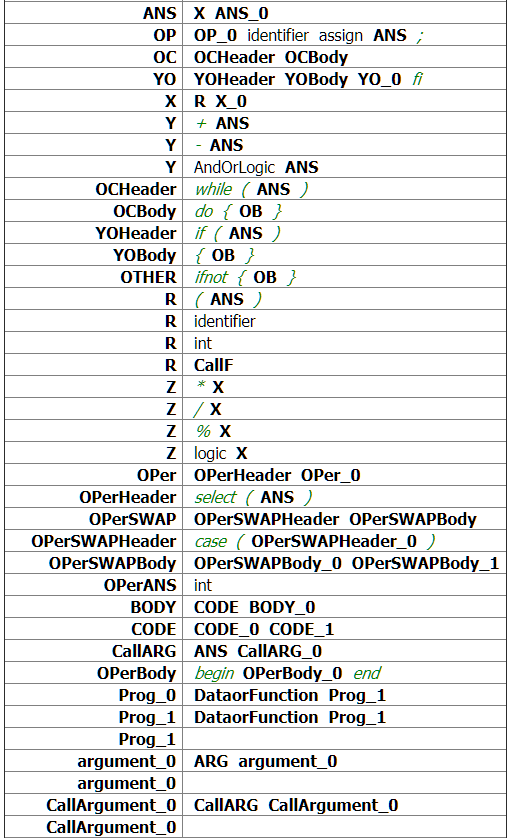


Рис. 8. Грамматика языка

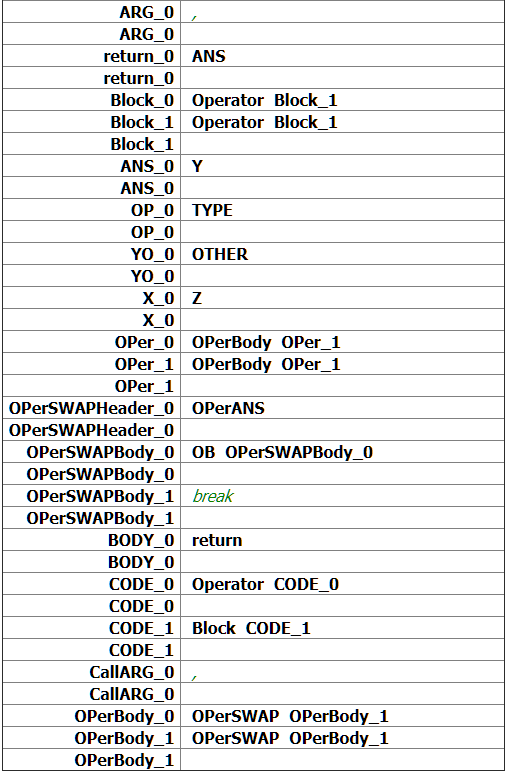


Рис.9. Грамматика языка.

Разработанная грамматика принадлежит к классу LL(1).

**Тестирование программы и шаблоны автоматов**

Для тестирования языка было подготовлено два варианта кода – верный (код, написанный в соответствии с лексическими и синтаксическими правилами языка) и ошибочный (верный код, в котором были допущены ошибки).

Верный код:

prog p6OG

entire s8MOFNUM(entire\_u i1NT) {

entire s7M = 0;

while (i1NT > 0) do { start

entire t5P = i1NT % 10;

s7M = s7M + t5P;

i1NT = i1NT / 10;

stop

}

return s7M;

}

real u2PU(entire\_u n8W) {

entire m4X = 0;

entire m4XEXP = 0;

entire i1NT = 0;

while (i1NT < n8W)

do {

start

entire s9M = func s8MOFNUM(i1NT);

entire\_u r3SULT = 0;

entire e8P = 2;

while (r3SULT < e8P)

do {start

r3SULT = s9M \* e8P;

if (i1NT == r3SULT) {start

m4X = i1NT;

m4EXP = e8P;

stop

}

fi

e8P = e8P + 1;

stop

}

i1NT = i1NT + 1;

stop

}

real t1ME = e2D - s7ART;

return t1ME;

}

entire main() {

entire\_u n5S = 5;

real c6UTIME = func u2PU(n5S);

return 0;

}

Ошибочный код:

prog p6OG

entire func s8MOFNUM(entire\_u i) {

entire sum = 0;

while (i > 0) do

entire t5P = i1NT % 10;

sum = sum + t5P;

i = i / 10;

}

return sum;

}

real u2PU(entire\_u n) {

entire max = 0;

entire maxexp = 0;

entire i = 0;

while (i < n)

do

start

entire s9M = func s8MOFNUM(i1NT);

entire\_u r3SULT = 0;

entire e8P = 2;

while (r3SULT < e8P)

do {start

r3SULT = s9M \* e8P;

if (i1NT == r3SULT) {start

m4X = i1NT;

m4EXP = e8P;

stop

}

fi

e8P = e8P + 1;

stop

}

i1NT = i1NT + 1;

stop

}

real t1ME = e2D - s7ART;

return t1ME;

}

entire main() {

entire\_u n5S = 5;

real c6UTIME = func u2PU(n5S);

return 0;

}

По шаблону lexAsTableSyntAsMultiFSM был построен конечный автомат, с помощью которого были протестированы оба варианта кода.

Результат тестирования верного варианта:

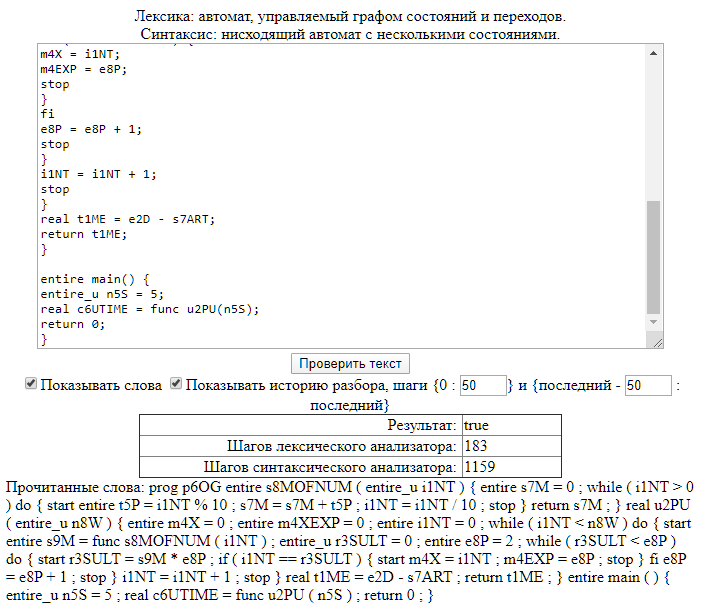


Рис. 10 – Результат тестирования верного варианта программы.

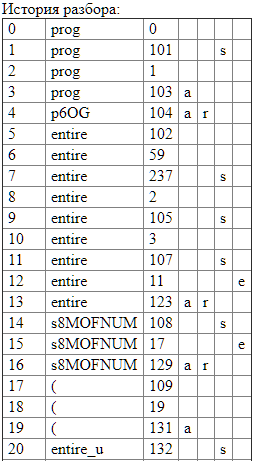


Рис. 11 – Часть истории разбора верного варианта программы.

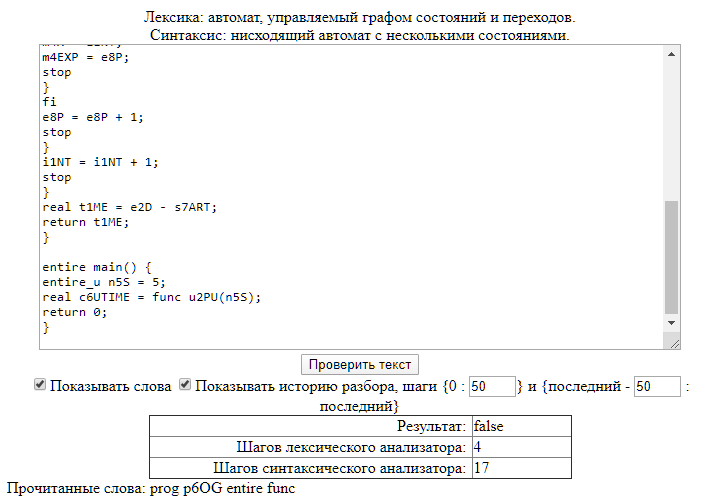


Рис. 12 – Результат тестирования ошибочного варианта программы.

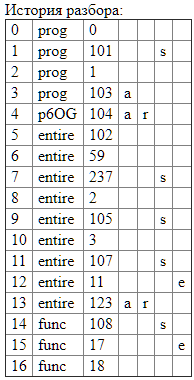


Рис. 13 – История разбора ошибочного варианта программы.

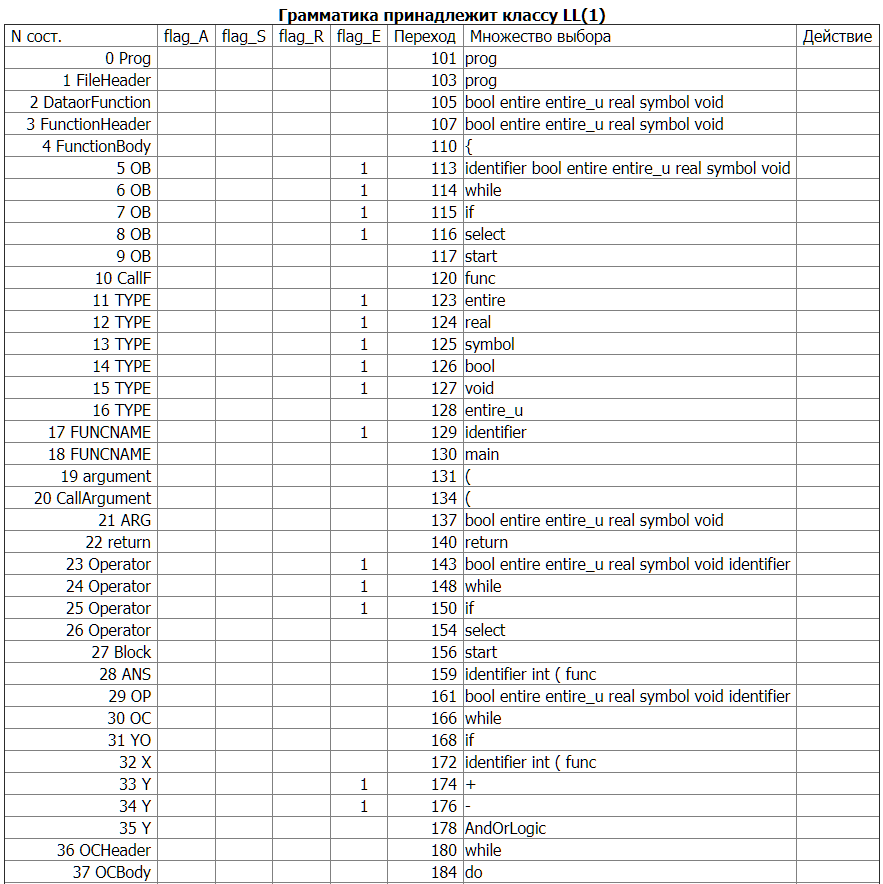


Рис. 14 – Управляющая таблица нисходящего автомата с несколькими состояниями.

В таблице, изображенной на рис. 14, показаны соотношения состояний и ожидаемые множества выбора. Если конструкция не входит в множество выбора, будет получена ошибка. В случае вхождения в множество выбора состояние изменяется.

По шаблону lexAsTableSyntAsSingleFSM был построен конечный автомат, с помощью которого были протестированы оба варианта кода.

Результат тестирования верного варианта:

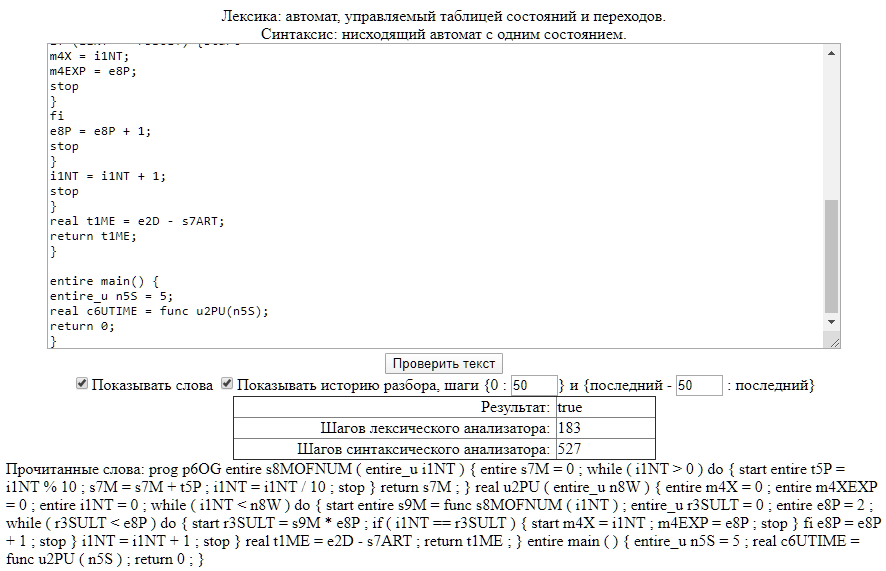


Рис. 15 – Результат тестирования верного варианта программы.

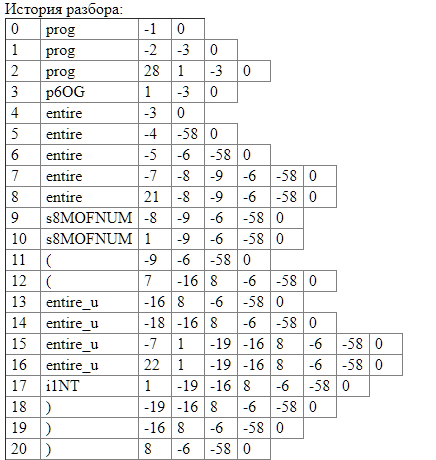


Рис. 16 – Часть истории разбора верного варианта программы.

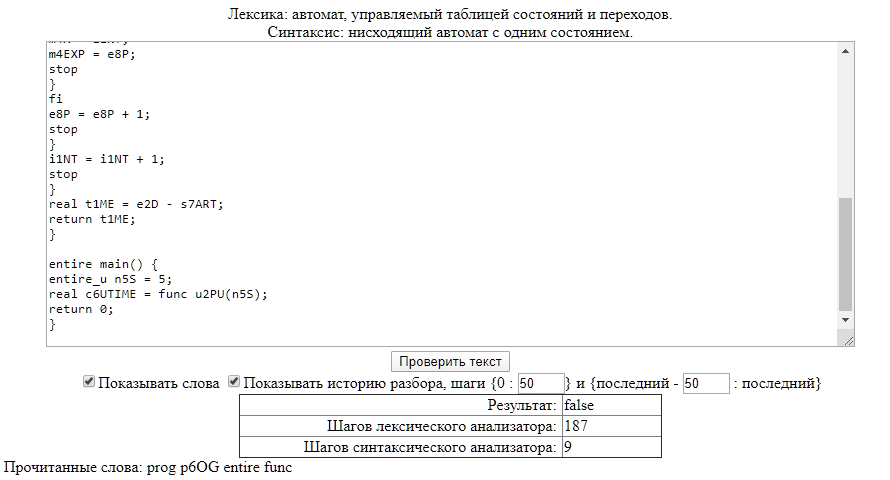


Рис. 17 – Результат тестирования ошибочного варианта программы.

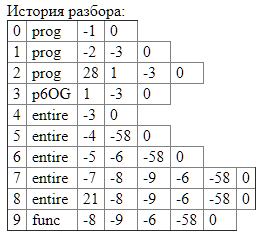


Рис. 18 – История разбора ошибочного варианта программы.

По шаблону lexAsTableSyntAsRD был построен конечный автомат, с помощью которого были протестированы оба варианта кода.

Результат тестирования верного варианта:

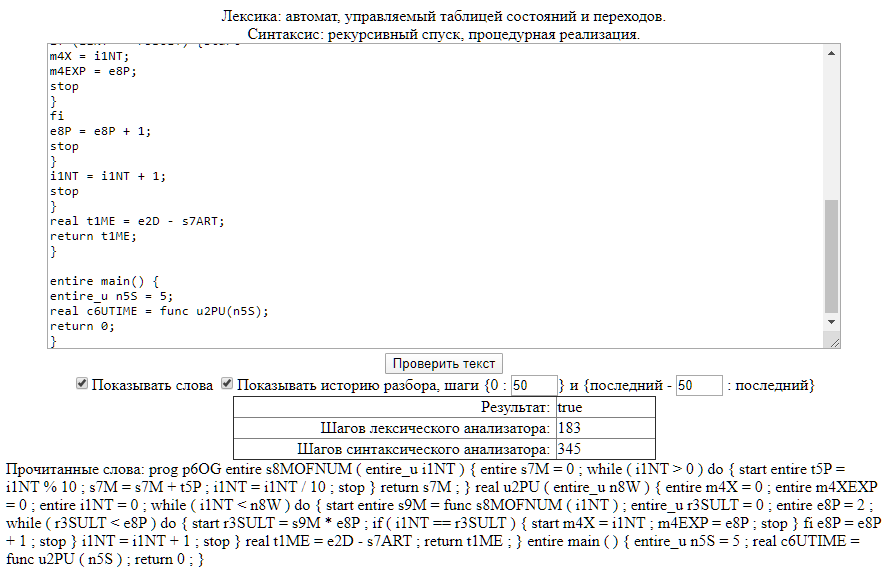


Рис. 19 – Результат тестирования верного варианта программы.

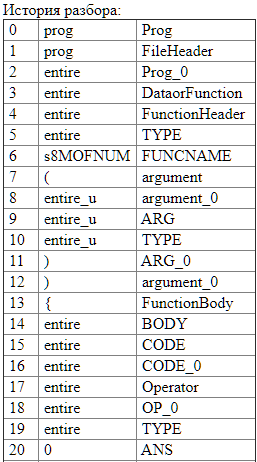


Рис. 20 – Часть истории разбора верного варианта программы.

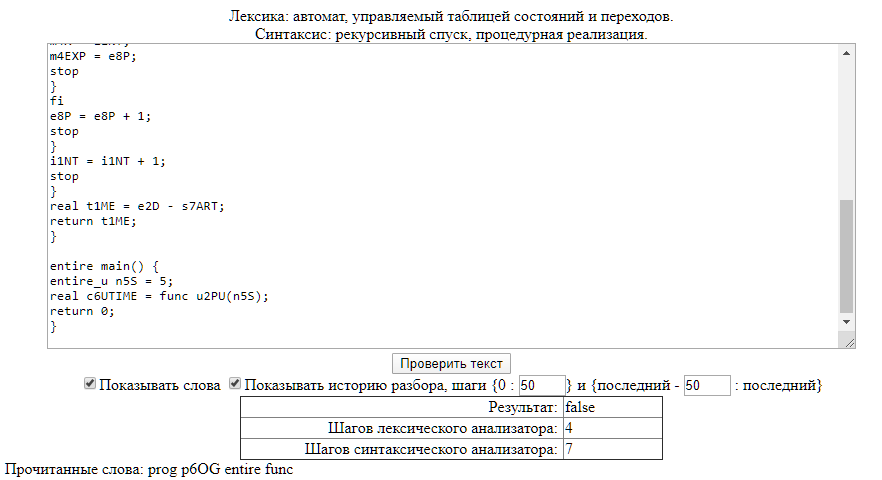


Рис. 21 – Результат тестирования ошибочного варианта программы.

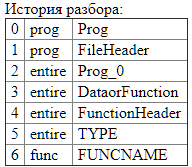


Рис. 22 – История разбора ошибочного варианта программы.

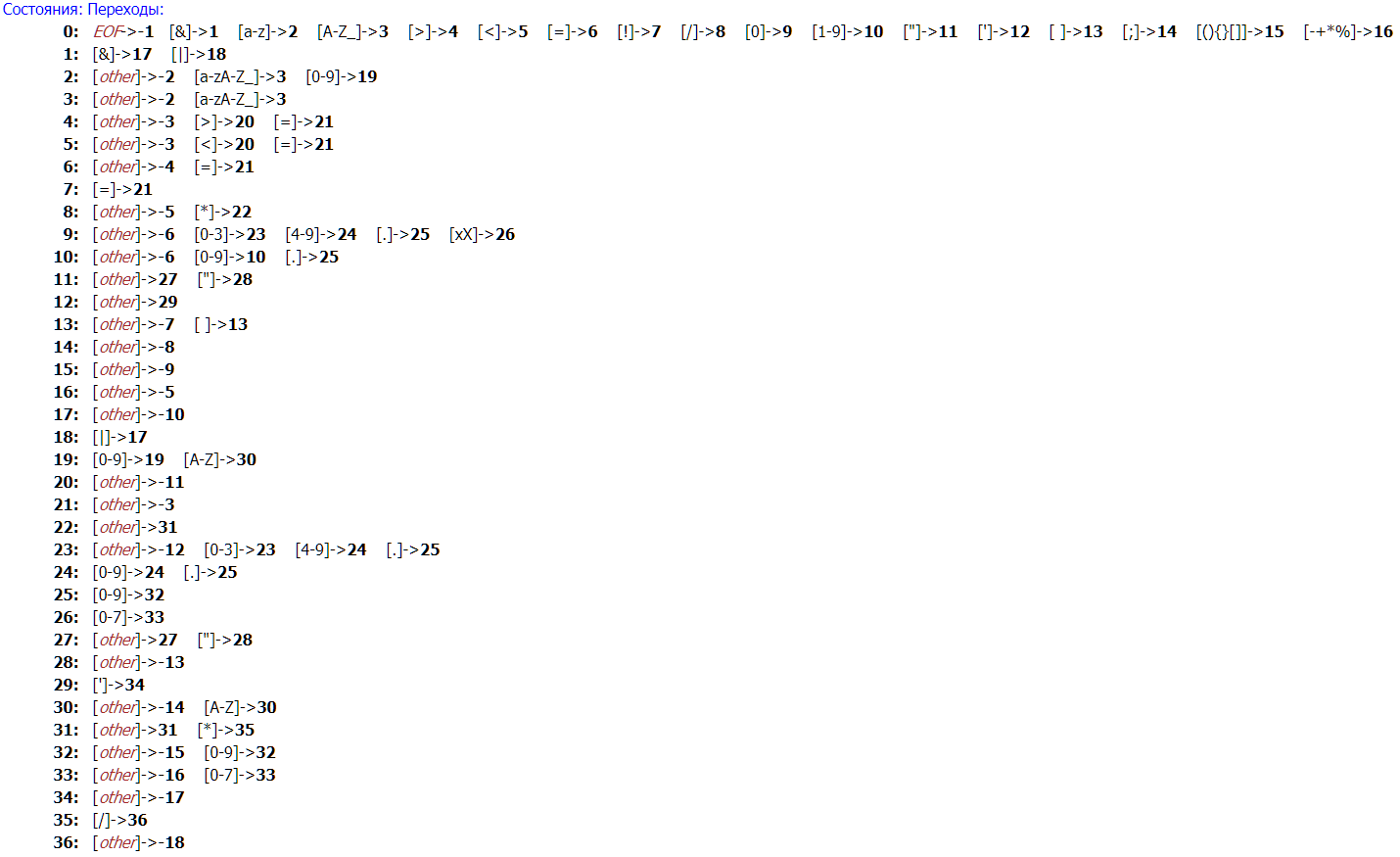
**Выводы**

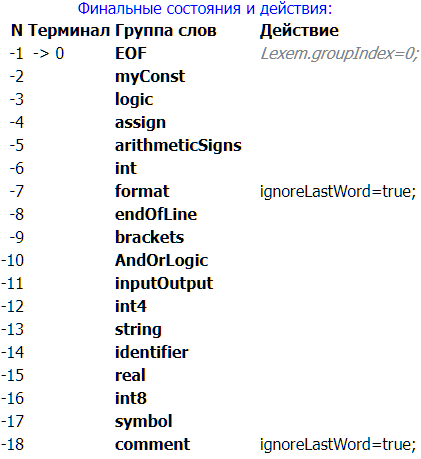
В ходе выполнения лабораторной работы были изучены основные идеи и понятия нисходящих методов синтаксического анализа, выявлены свойства формальных грамматик, необходимых для реализации нисходящего восстановления дерева грамматического разбора, приобретены навыки построения процедурной и различных автоматных реализаций нисходящего анализа, исследовано поведения нисходящих синтаксических акцепторов.

Также был сделан вывод, что автомат, построенный по шаблону lexAsTableSyntAsRD, является наиболее удобным для поиска ошибок в программе, так как таблица истории разбора вместо состояний, как это делают другие изученные автоматы, отображает типы в том виде, в котором они описаны в синтаксических правилах языка. Кроме того, синтаксический анализатор этого автомата делает наименьшее количество шагов при анализе кода, что также упрощает процесс поиска ошибок.**История работы лексических автоматов для предыдущих лабораторных работ**

История разбора обработки лексического фрагмента «entire\_u i1NT = 356;»

**Графовый лексический автомат:**

****

****

0: [e] → 2

2: [n] → 3

3: [t] → 3

3: [i] → 3

3: [r] → 3

3: [e] → 3

3: [\_] → 3

3: [u] → 3

3: [ ] → -2 – myConst

0: [ ] → 13

13: [i] → -7 – format

0: [i] → 2

2: [1] → 19

19: [N] → 30

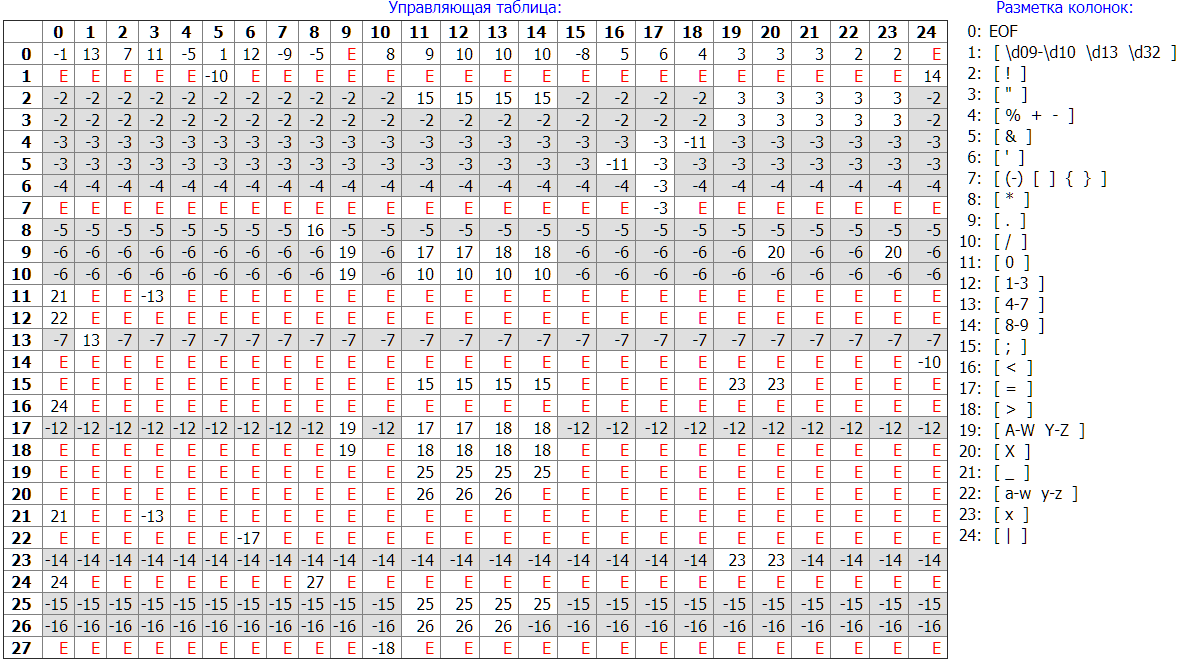
30: [T] → 30

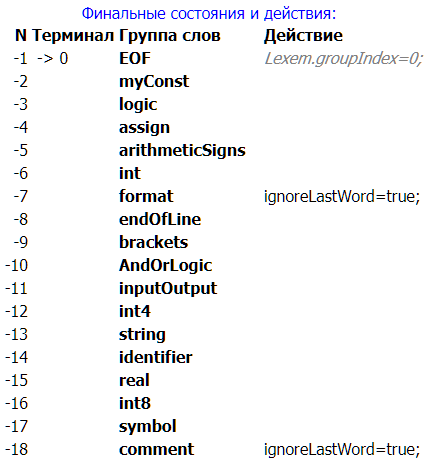
30: [;] → -14 – identifier

0: [;] → 14

14: [;] → -8 – endOfLine

**Табличный лексический автомат:**





0: [e] (22) → 2

2: [n] (22) → 3

3: [t] (22) → 3

3: [i] (22) → 3

3: [r] (22) → 3

3: [e] (22) → 3

3: [\_] (21) → 3

3: [u] (22) → 3

3: [ ] (1) → -2 – myConst

0: [ ] (1) → 13

13: [i] (22) → -7 – format

0: [i] (22) → 2

2: [1] (12) → 15

15: [N] (19) → 23

23: [T] (19) → 23

23: [;] (15) → -14 – identifier

0: [;] (15) → -8 – endOfLine