**Поволжский Государственный Университет Телекоммуникаций и Информатики**

Кафедра «ИВТ»

Сдана на проверку

« » 2024 г.

Допустить к защите

« » 2024 г. Защищена с оценкой

« » 2024 г.

# КУРСОВАЯ РАБОТА

По дисциплине: «Теория автоматов и формальных языков» На тему: «Разработка компилятора модельного языка» Вариант 106

Пояснительная записка

Студент группы ИВТ-11 Козин К.А.

(подпись) (ФИО)

Руководитель Бахарева Н.Ф.

(подпись) (ФИО)

210234

(№ зачетной книжки)

Самара, 2024

# Рецензия

Оглавление

[Рецензия 2](#_bookmark0)

[Введение 4](#_bookmark1)

1. [Постановка задачи 5](#_bookmark2)
2. [Формальная модель задачи 7](#_bookmark3)
   1. [Формальные грамматики 7](#_bookmark4)
   2. [Формы Бэкуса-Наура 7](#_bookmark5)
   3. [Расширенные формы Бэкуса-Наура 8](#_bookmark6)
   4. [Диаграммы Вирта 8](#_bookmark7)
3. [Спецификация основных процедур и функций 19](#_bookmark8)
   1. [Лексический анализатор 19](#_bookmark9)
   2. [Диаграмма состояний с действиями 21](#_bookmark10)
   3. [Синтаксический анализатор 25](#_bookmark11)
   4. [Семантический анализ 26](#_bookmark12)
4. [Структурная организация данных 28](#_bookmark13)
   1. [Спецификация входных данных 28](#_bookmark14)
   2. [Спецификация выходных данных 28](#_bookmark15)
5. [Разработка алгоритма решения задач 29](#_bookmark16)
   1. [Укрупненная схема алгоритма программного средства 29](#_bookmark17)
   2. [Детальная разработка алгоритмов отдельных подзадач 30](#_bookmark18)
      1. [Блок-схема лексического анализатора 30](#_bookmark19)
      2. [Блок-схема синтаксического анализатора 32](#_bookmark20)
      3. [Блок-схема семантического анализатора 35](#_bookmark21)
6. [Работа с программным средством 37](#_bookmark22)
   1. [Описание интерфейса 37](#_bookmark23)
   2. [Перечень ошибок 37](#_bookmark24)
   3. [Пример работы компилятора 38](#_bookmark25)
   4. [Генерация объектного файла 47](#_bookmark26)

[Заключение 51](#_bookmark27)

[Список использованных источников 52](#_bookmark28)

[Приложение А – Текст программы 53](#_bookmark29)

[Приложение Б – Генерация объектного файла 60](#_bookmark30)

# Введение

Компилятор – это программа, которая создаётся для перевода программы, написанной на высокоуровневом языке программирования в машинный код, который в последствие будет исполняться процессором компьютера.

В большинстве случаев компиляция программы происходит полностью. Компилятор целиком считывает программу, проводит её пошаговый анализ (лексический, синтаксический, семантический), оптимизирует её, очищая от излишних конструкций, но сохраняя исходный смысл операций, и также целиком переводит её в машинный код.

Цель курсовой работы:

* закрепление теоретических знаний в области теории формальных языков, грамматик, автоматов и методов трансляции;
* формирование практических умений и навыков разработки собственного компилятора модельного языка программирования;
* закрепление практических навыков самостоятельного решения инженерных задач, развитие творческих способностей студентов и умений пользоваться технической, нормативной и справочной литературой.

В настоящее время в мире появляются более новые языки программирования и не каждый из ныне существующих трансляторов могут прочитать программы, написанный на новом языке, и перевести его в другой язык. Поэтому сейчас разрабатываются новые трансляторы, в этом и заключается актуальность данной курсовой работы.

# Постановка задачи

Разработать компилятор модельного языка, выполнив следующие действия.

1. В соответствии с номером варианта составить формальное описание модельного языка программирования с помощью:

а) РБНФ;

б) диаграмм Вирта;

в) формальных грамматик.

1. Написать пять содержательных примеров программ, раскрывающих особенности конструкций учебного языка программирования, отразив в этих примерах все его функциональные возможности.
2. Составить таблицы лексем и диаграмму состояний с действиями для распознавания и формирования лексем языка.
3. По диаграмме с действиями написать функцию сканирования текста входной программы на модельном языке.
4. Разработать программное средство, реализующее лексический анализ текста программы на входном языке.
5. Реализовать синтаксический анализатор текста программы на модельном языке методом рекурсивного спуска.
6. Построить цепочку вывода и дерево разбора простейшей программы на модельном языке из начального символа грамматики.
7. Дополнить синтаксический анализатор процедурами проверки семантической правильности программы на модельном языке в соответствии с контекстными условиями вашего варианта.
8. Распечатать пример таблиц идентификаторов и двуместных операций.
9. Показать динамику изменения содержимого стека при семантическом анализе программы на примере одного синтаксически правильного выражения.
10. Записать правила вывода грамматики с действиями по переводу в ПОЛИЗ программы на модельном языке.
11. Пополнить разработанное программное средство процедурами, реализующими генерацию внутреннего представления введенной программы в форме ПОЛИЗа.
12. Разработать интерпретатор ПОЛИЗа программы на модельном языке.
13. Составить набор контрольных примеров, демонстрирующих: а) все возможные типы лексических, синтаксических и

семантических ошибок в программах на модельном языке; б) перевод в ПОЛИЗ различных конструкций языка;

в) представить ход интерпретации синтаксически и семантически правильной программы с помощью таблицы.

# Формальная модель задачи

Существуют три основных метода описания синтаксиса языков программирования: формальные грамматики, формы Бэкуса-Наура и диаграммы Вирта.

# Формальные грамматики

**Определение.** Формальной грамматикой называется четверка вида:

*G*  (*VT* , *VN* , *P*, *S* ), (1.1)

где *VN* - конечное множество нетерминальных символов грамматики (обычно прописные латинские буквы);

*VT* - множество терминальных символов грамматики (обычно строчные латинские буквы, цифры, и т.п.), *VT* *VN =*;

*Р* – множество правил вывода грамматики, являющееся конечным подмножеством множества (*VT* *VN*)*+*  (*VT* *VN*)*\**; элемент (*,* ) множества *Р -* называется правилом вывода и записывается в виде  (читается: «из цепочки  выводится цепочка »);

*S* – начальный символ грамматики, *S* *VN*.

Для записи правил вывода с одинаковыми левыми частями вида

  1,   2,…,  *n* используется сокращенная форма записи

  1 | 2 |…| *n*.

# Формы Бэкуса-Наура

Метаязык, предложенный Бэкусом и Науром, использует следующие обозначения:

* символ «::=» отделяет левую часть правила от правой (читается:

«определяется как»);

* нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки «<» и «>»;
* терминалы — это символы, используемые в описываемом языке;
* правило может определять порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты «|» (читается: «или»).

# Расширенные формы Бэкуса-Наура

Для повышения удобства и компактности описаний, в РБНФ вводятся следующие дополнительные конструкции (метасимволы):

* квадратные скобки «[» и «]» означают, что заключенная в них синтаксическая конструкция может отсутствовать;
* фигурные скобки «{» и «}» означают повторение заключенной в них синтаксической конструкции ноль или более раз;
* сочетание фигурных скобок и косой черты «{/» и «/}» используется для обозначения повторения один и более раз;
* круглые скобки «(» и «)» используются для ограничения альтернативных конструкций.

# Диаграммы Вирта

## Вариант: **311311**

В метаязыке диаграмм Вирта используются графические примитивы. При построении диаграмм учитывают следующие правила:

* каждый графический элемент, соответствующий терминалу или нетерминалу, имеет по одному входу и выходу, которые обычно изображаются на противоположных сторонах;
* каждому правилу соответствует своя графическая диаграмма, на которой терминалы и нетерминалы соединяются посредством дуг;
* альтернативы в правилах задаются ветвлением дуг, а итерации - их слиянием;
* должна быть одна входная дуга (располагается обычно слева или сверху), задающая начало правила и помеченная именем определяемого нетерминала, и одна выходная, задающая его конец (обычно располагается справа и снизу);
* стрелки на дугах диаграмм обычно не ставятся, а направления связей отслеживаются движением от начальной дуги в соответствии с плавными изгибами промежуточных дуг и ветвлений.

Таблица 5.1 - Операции группы «отношение»

|  |  |
| --- | --- |
| Номер | Синтаксис группы операций  (в порядке следования: неравно, равно, меньше, меньше или равно, больше, больше или равно) |
| 3 | <операции\_группы\_отношения>::=NE | EQ | LT | LE | GT | GE |

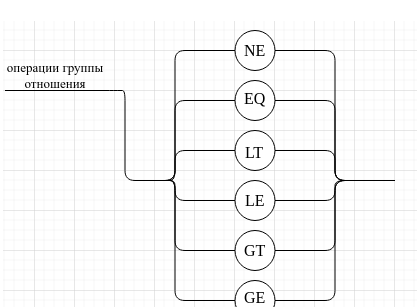


Таблица 5.2 - Операции группы «сложение»

|  |  |
| --- | --- |
| Номер | Синтаксис группы операций  (в порядке следования: сложение, вычитание, дизъюнкция) |
| 3 | <операции\_группы\_сложения>:: = *plus* | *min* | or |

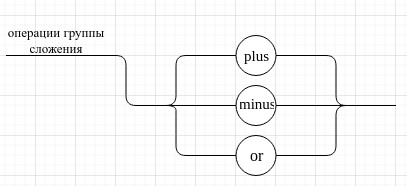


Таблица 5.3 - Операции группы «умножение»

|  |  |
| --- | --- |
| Номер | Синтаксис группы операций  (в порядке следования: умножение, деление, конъюнкция) |
| 3 | <операции\_группы\_умножения>::= mult | div | and |

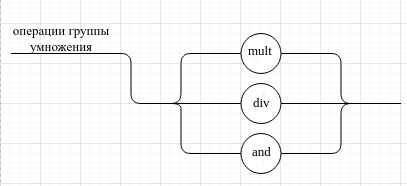
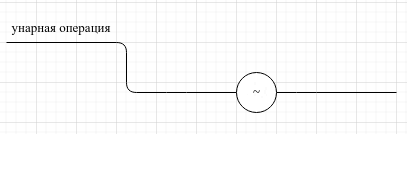


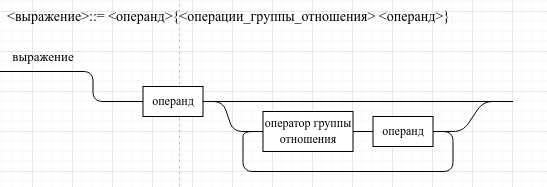
Таблица 5.4 - Унарная операция

|  |  |
| --- | --- |
| Номер | Синтаксис операции |
| 3 | <унарная\_операция>::= ~ |

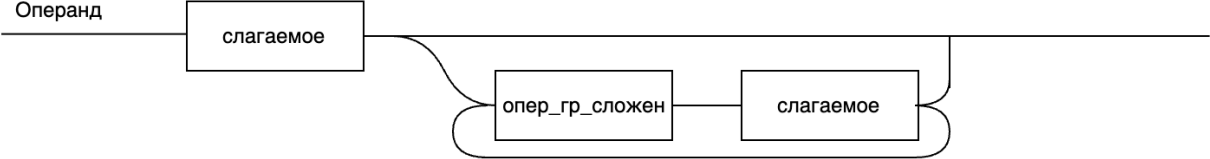


Выражения языка задаются правилами:

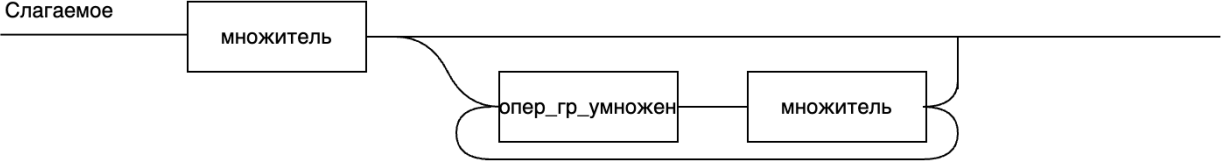
<выражение>::= <операнд>{<операции\_группы\_отношения> <операнд>}



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

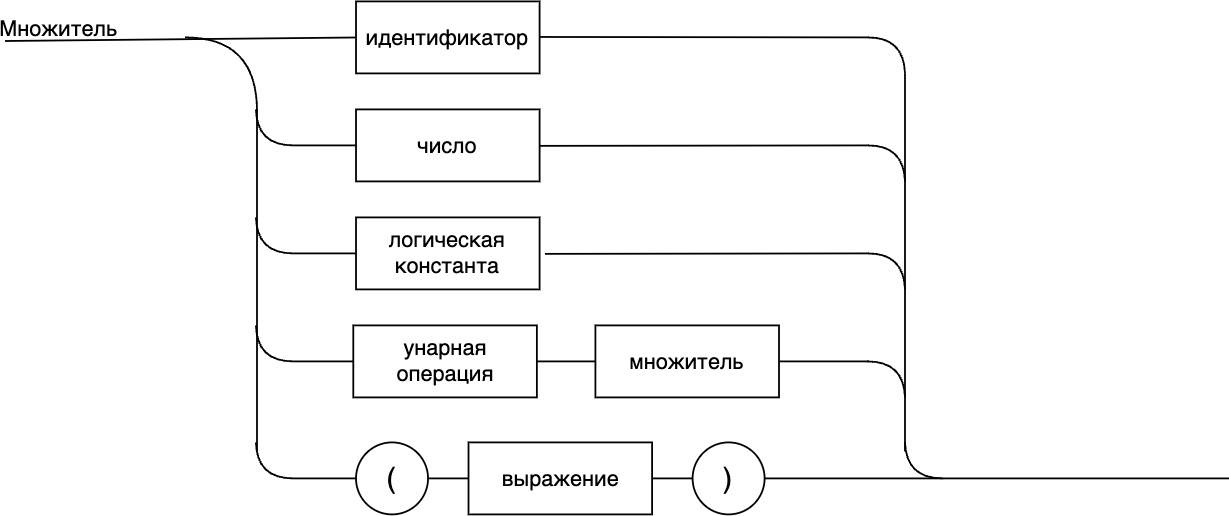


<слагаемое>::= <множитель> {<операции\_группы\_умножения> <множитель>}

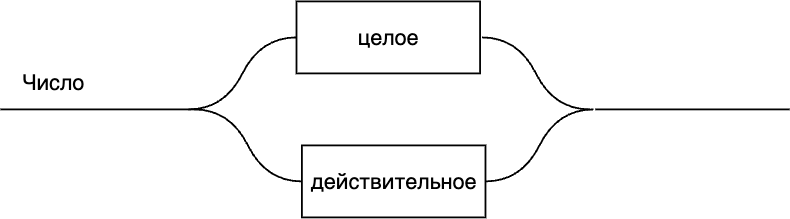


<множитель>::= <идентификатор> | <число> | <логическая\_константа> |

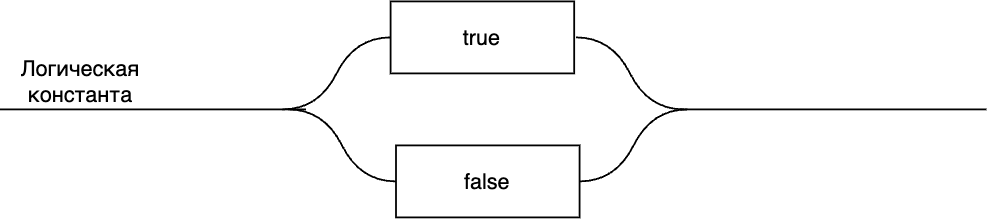
<унарная\_операция> <множитель> | «(»<выражение>«)»



<число>::=<целое>|<действительное>

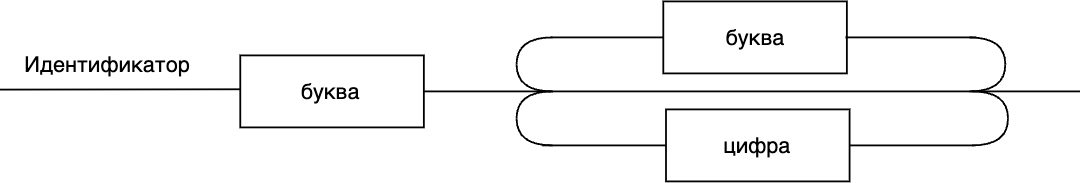


логическая\_константа>::= *true* | *false*



Правила, определяющие идентификатор, букву и цифру:

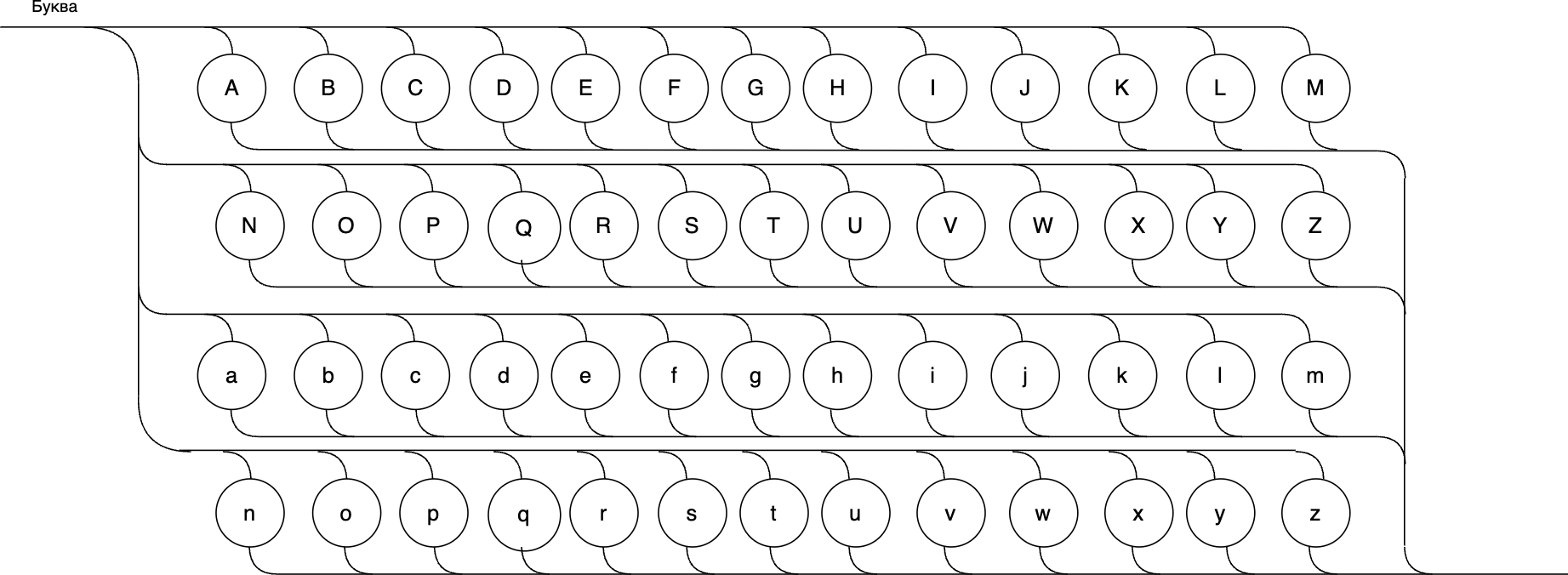
<идентификатор>::= <буква> {<буква> | <цифра>}



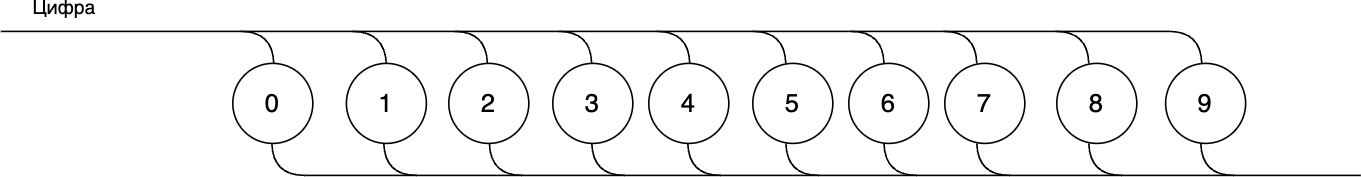
<буква>::= *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *O* | *P* | *Q* | *R* | *S* | *T* |*U* | *V*

| *W* | *X* | *Y* | *Z* | *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*



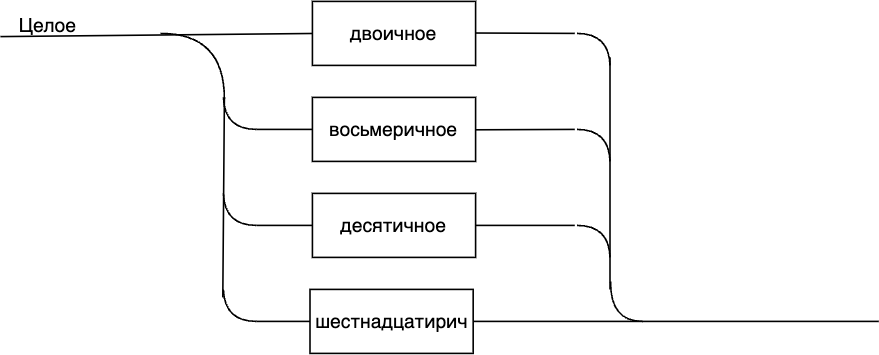
<цифра>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



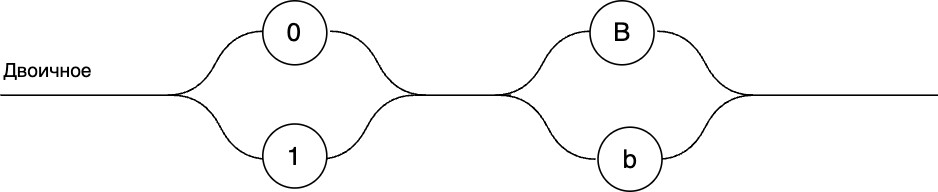
Правила, определяющие целые числа:

<целое>::= <двоичное> | <восьмеричное> |

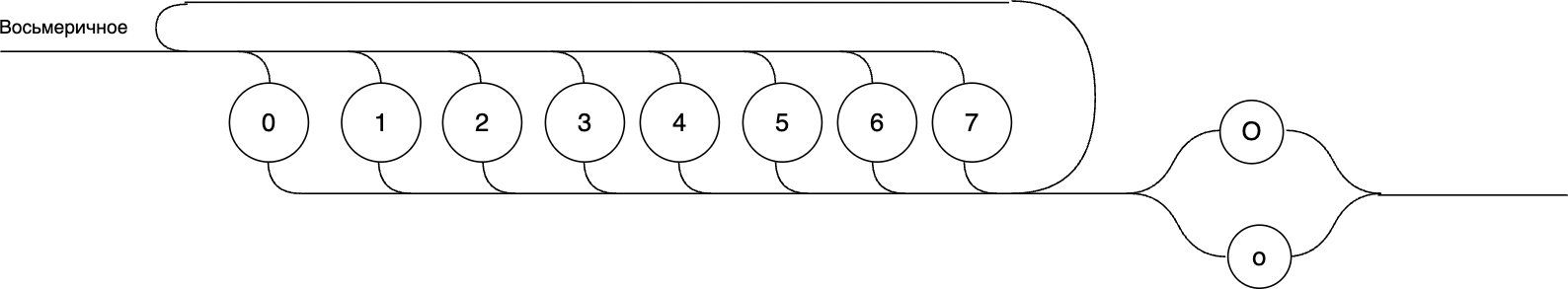
<десятичное>|<шестнадцатеричное>



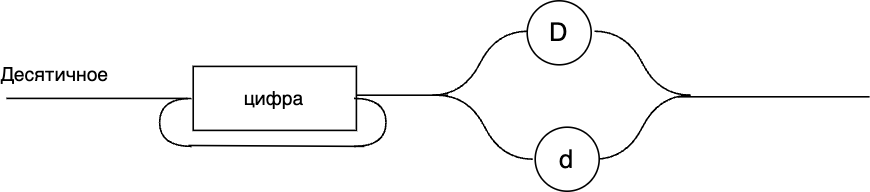
<двоичное>::= {/ 0 | 1 /} (*B* | *b*)



<восьмеричное>::= {/ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 /} (*O* | *o*)

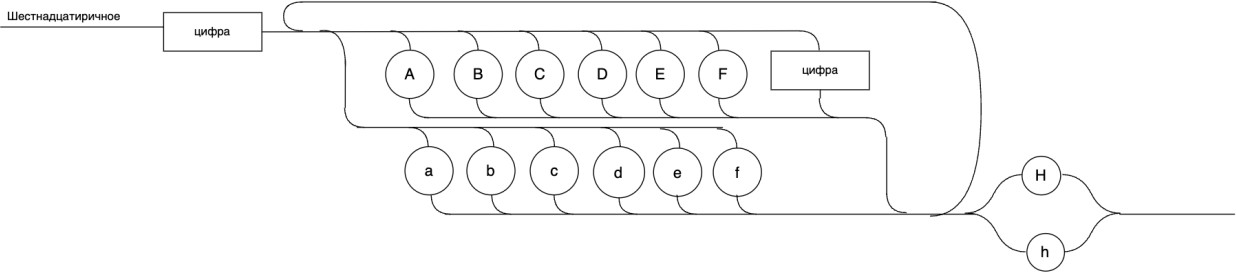


<десятичное>::= {/ <цифра> /} [*D* | *d*]



<шестнадцатеричное>::= <цифра> {<цифра> | *A* | *B* | *C* | *D* | *E* | *F* | *a* | *b* | *c* | *d* | *e*

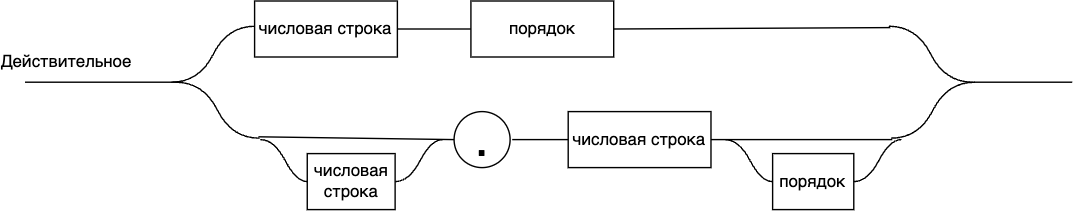
| *f*} (*H* | *h*)



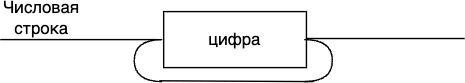
Правила, описывающие действительные числа:

<действительное>::= <числовая\_строка> <порядок> |[<числовая\_строка>] .

<числовая\_строка> [<порядок>]



<числовая\_строка>::= {/ <цифра> /}



<порядок>::= ( E | e )[+ | -] <числовая\_строка>

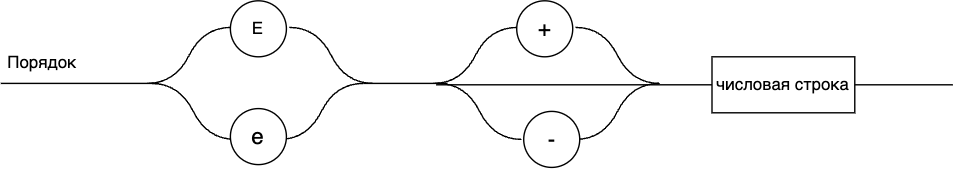


Таблица 5.5 – Структура программы

|  |  |
| --- | --- |
| Номер | Структура программы |
| 1 | <программа>::= program var <описание> begin <оператор> {; <опера-  тор>} end. |

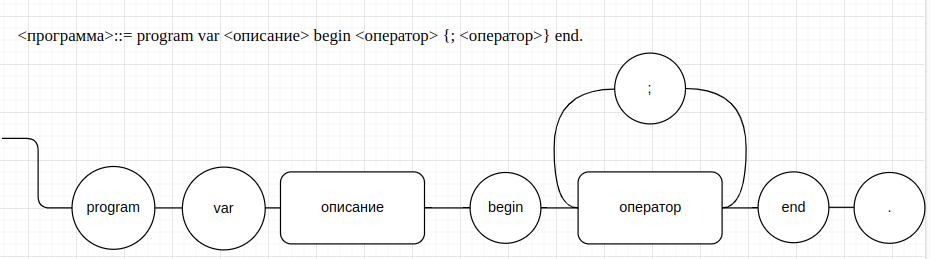


Таблица 5.6 - Синтаксис команд описания данных

|  |  |
| --- | --- |
| Номер | Синтаксис команд описания данных |
| 1 | <описание>::= {<идентификатор> {, <идентификатор> } : <тип> ;} |

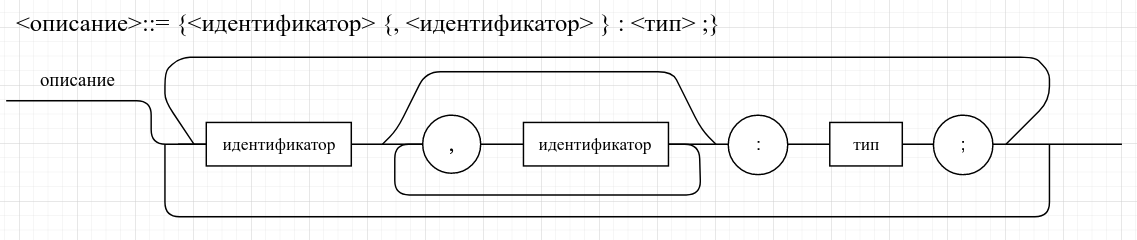
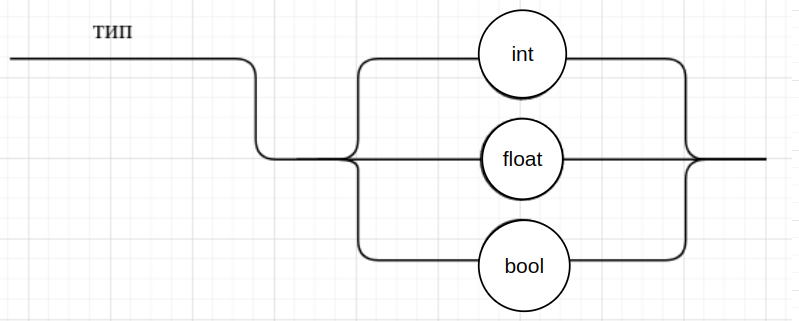


Таблица 5.7- Описание типов данных

|  |  |
| --- | --- |
| Номер | Описание типов  (в порядке следования: целый, действительный, логический) |
| 3 | <тип>::= int | float | bool |



Правило, определяющее оператор программы (пятая цифра варианта).

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

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

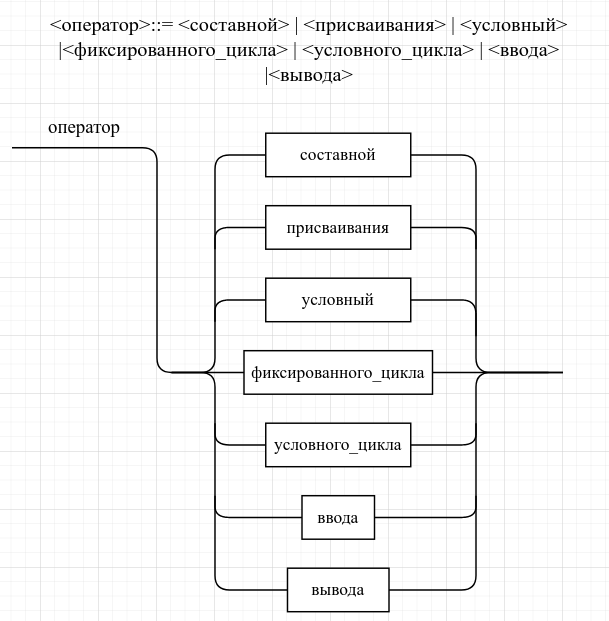


Таблица 5.8 - Синтаксис составного оператора

|  |  |
| --- | --- |
| Номер | Синтаксис оператора |
| 1 | <составной>::= <оператор> { ( : | перевод строки) <оператор> } |

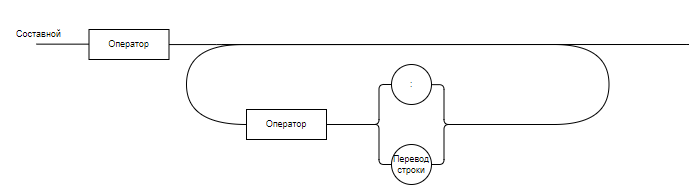


Таблица 5.9 - Синтаксис оператора присваивания

|  |  |
| --- | --- |
| Номер | Оператор присваивания |
| 1 | <присваивания>::= <идентификатор> *as* <выражение> |

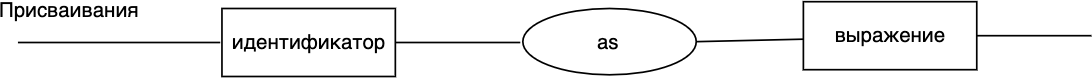


Таблица 5.10 - Синтаксис оператора условного перехода

|  |  |
| --- | --- |
| Номер | Оператор условного перехода |
| 1 | <условный>::= if <выражение> then <оператор> [ else <оператор>] |

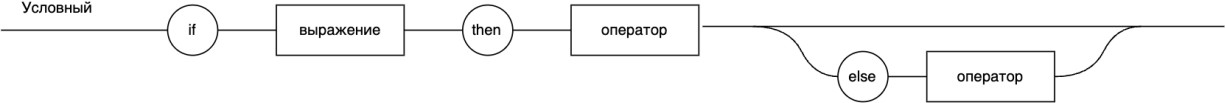


Таблица 5.11 - Синтаксис оператора цикла с фиксированным числом повторений

|  |  |
| --- | --- |
| Номер | Синтаксис оператора |
| 1 | <фиксированного\_цикла>::= *for* <присваивания> *to* <выражение> *do*  <оператор> |

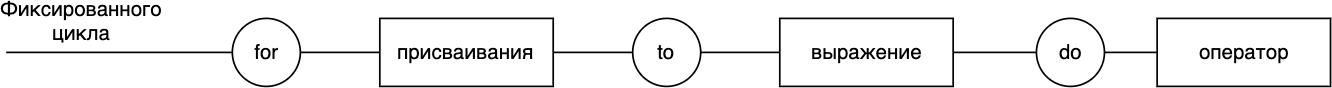


Таблица 5.12 - Синтаксис условного оператора цикла

|  |  |
| --- | --- |
| Номер | Синтаксис оператора |
| 1 | <условного\_цикла>::= *while* <выражение> *do* <оператор> |



Таблица 5.13 - Синтаксис оператора ввода

|  |  |
| --- | --- |
| Номер | Синтаксис оператора |
| 1 | <ввода>::= *read* «(» <идентификатор> {, <идентификатор> }«)» |

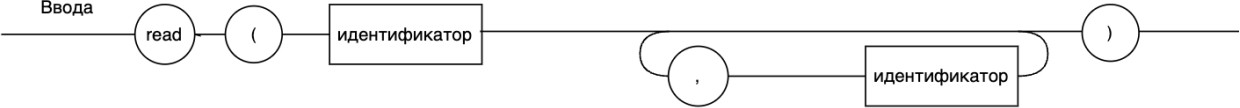


Таблица 5.14 - Синтаксис оператора вывода

|  |  |
| --- | --- |
| Номер | Синтаксис оператора |
| 1 | <вывода>::= *write* «(»<выражение> {, <выражение> }«)» |

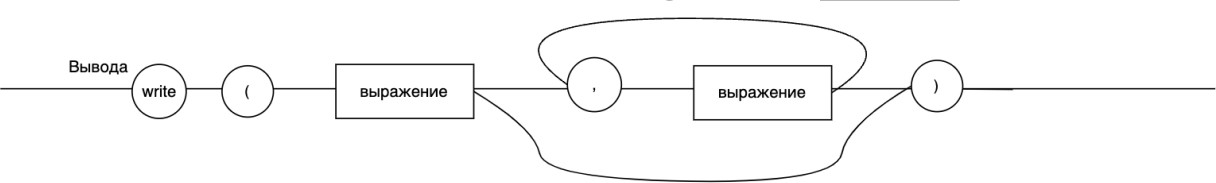
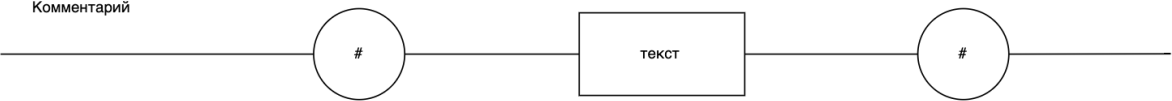


Таблица 5.15 – Синтаксис многострочных комментариев

|  |  |  |
| --- | --- | --- |
| Номер | Признак начала комментария | Признак конца комментария |
| 5 | # | # |



# Спецификация основных процедур и функций

# Лексический анализатор

Лексический анализатор (ЛА) – это первый этап процесса компиляции, на котором символы, составляющие исходную программу, группируются в отдельные минимальные единицы текста, несущие смысловую нагрузку – лексемы.

Задача лексического анализа - выделить лексемы и преобразовать их к виду, удобному для последующей обработки.

Каждая лексема представляет собой пару чисел вида (n, k), где n – номер таблицы лексем, k - номер лексемы в таблице.

Входные данные ЛА - текст транслируемой программы на входном языке.

Выходные данные ЛА - список лексем в таблице

Для хранения таблиц были использованы файлы с расширением *.txt*. Это обеспечивает простое и удобное измение таблиц без редактирования исходного кода компилятора.

Таблица № 1 – таблица ключевых слов:

|  |  |
| --- | --- |
| Номер | Ключевое слово |
| 1 | if |
| 2 | then |
| 3 | else |
| 4 | for |
| 5 | to |
| 6 | do |
| 7 | while |
| 8 | read |
| 9 | write |
| 10 | int |
| 11 | float |
| 12 | bool |
| 13 | program |
| 14 | var |
| 15 | begin |
| 16 | end |
| 17 | true |
| 18 | false |

Таблица №2 - таблица разделителей

|  |  |  |  |
| --- | --- | --- | --- |
| Номер | Разделитель | Номер | Разделитель |
| 1 | as | 13 | or |
| 2 | NE | 14 | ( |
| 3 | EQ | 15 | ) |
| 4 | LT | 16 | , |
| 5 | LE | 17 | ; |
| 6 | GT | 18 | : |
| 7 | GE | 19 | { |
| 8 | plus | 20 | } |
| 9 | min | 21 | . |
| 10 | mult | 22 | ~ |
| 11 | div |
| 12 | and |

Таблицы идентификаторов (3) и чисел (4) формируются в ходе лексического анализа.

Опишем результаты работы лексического анализатора для модельного языка *М*.

Составим ЛА для модельного языка. Предварительно введем следующие обозначения для переменных, процедур и функций.

## Переменные:

1. *СН* – очередной входной символ;
2. *S* - буфер для накапливания символов лексемы;
3. *B* – переменная для формирования числового значения константы;
4. *CS* - текущее состояние буфера накопления лексем с возможными значениями: *Н* - начало, *I* - идентификатор, *N* - число, *С* - комментарий, *DV* – двоеточие, *О* - ограничитель, *V* - выход, *ER* –ошибка;
5. *t* - таблица лексем анализируемой программы с возможными значениями: *TW* - таблица служебных слов *М*-языка, *TL* – таблица ограничителей *М*-языка, *TI* - таблица идентификаторов программы, *TN* – чисел, используемых в программе;

6) *z* - номер лексемы в таблице *t* (если лексемы в таблице нет, то *z*=0).

## Процедуры и функции:

*СН*;

1. *gc* – процедура считывания очередного символа текста в переменную
2. *let* – логическая функция, проверяющая, является ли переменная *СН*

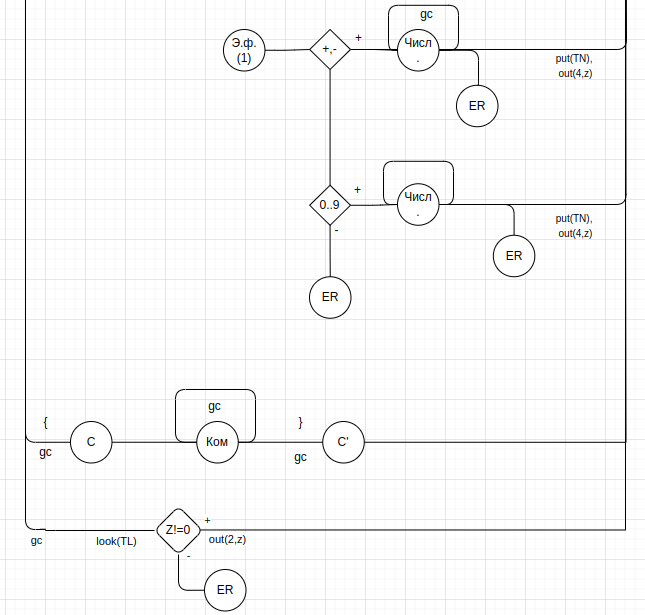
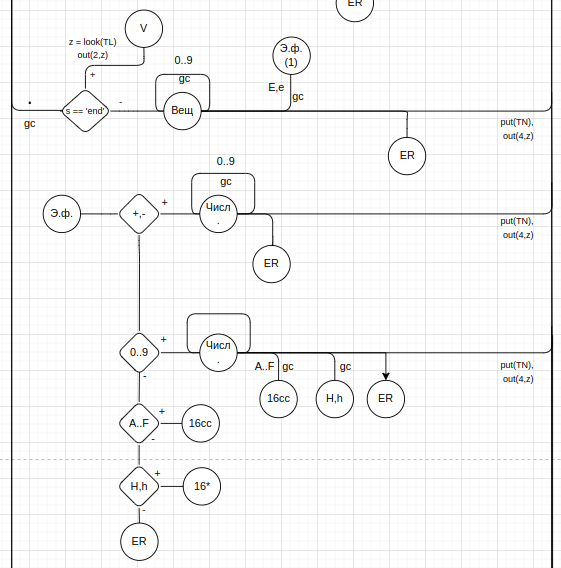
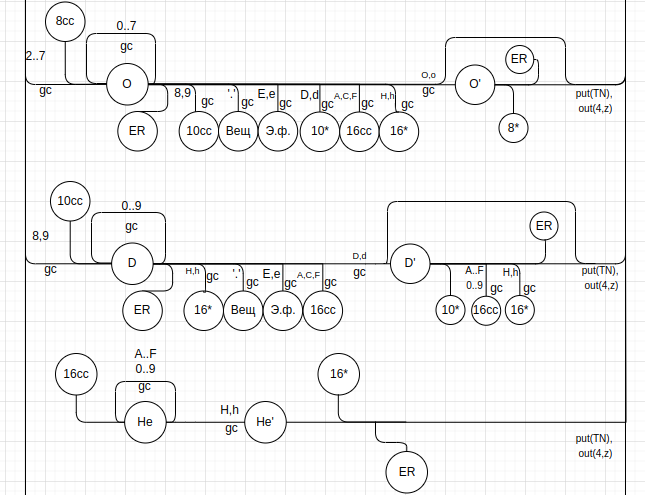
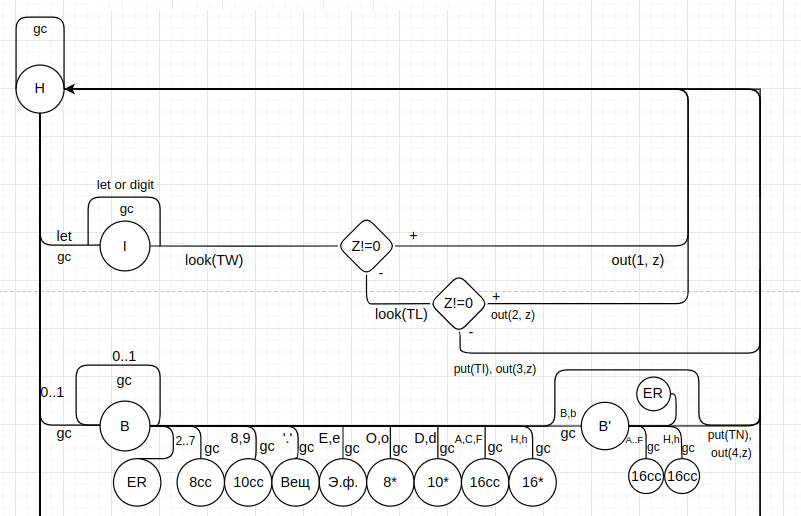
буквой;

1. *digit* - логическая функция, проверяющая, является ли переменная

*СН* цифрой;

1. *nill* – процедура очистки буфера *S*;
2. *add* – процедура добавления очередного символа в конец буфера *S*;
3. *look*(*t*) – процедура поиска лексемы из буфера *S* в таблице *t* с возвращением номера лексемы в таблице;
4. *put*(*t*) – процедура записи лексемы из буфера *S* в таблицу *t*, если там не было этой лексемы, возвращает номер данной лексемы в таблице;
5. *out*(*n*, *k*) – процедура записи пары чисел (*n*, *k*) в файл лексем. Построим ДС с действиями для распознавания и формирования внутреннего представления лексем модельного языка М

# Диаграмма состояний с действиями



# Синтаксический анализатор

Задача синтаксического анализатора: провести разбор текста программы, сопоставив его с эталоном, данным в описании языка. Для синтаксического разбора используются контекстно-свободные грамматики (КС-грамматика).

Один из эффективных методов синтаксического анализа – метод рекурсивного спуска.

В основе метода рекурсивного спуска лежит левосторонний разбор строки языка. Исходной сентенциальной формой является начальный символ грамматики, а целевой – заданная строка языка. На каждом шаге разбора правило грамматики применяется к самому левому нетерминалу сентенции. Данный процесс соответствует построению дерева разбора цепочки сверху вниз (от корня к листьям). В курсовой работе применяется такая проверка открытия и закрытия программы, как:

1. Код в программе начинается с «program», который является ключевым словом;
2. Считав разделитель «.», объявляется конец программы;
3. Если разделитель «.» не обнаружен, выводится ошибка.

# Семантический анализ

В ходе семантического анализа проверяются отдельные правила записи исходных программ, которые не описываются КС-грамматикой. Эти правила носят контекстно-зависимый характер, их называют семантическими соглашениями или контекстными условиями.

Соблюдение контекстных условий для языка М предполагает три типа проверок:

1. обработка описаний;
2. анализ выражений;
3. проверка правильности операторов.

Вход: файл лексем в числовом представлении.

Выход: заключение о семантической правильности программы или о типе обнаруженной семантической ошибки.

Задача обработки описаний – проверить, все ли переменные программы описаны правильно и только один раз.

Задача анализа выражений – проверить описаны ли переменные, встречающиеся в выражениях, и соответствуют ли типы операндов друг другу и типу операции.

Задачи проверки правильности операторов:

1. выяснить, все ли переменные, встречающиеся в операторах, описаны; установить соответствие типов в операторе присваивания слева и справа от символа «as»;

Задача решается проверкой типов в соответствующих местах программы.

Если при анализе цепочек лексем возникает ошибка или встретилась неизвестная лексема или лексема, не принадлежащая анализируемому оператору, то вызывается исключение с текстом ошибки.

# Структурная организация данных

# Спецификация входных данных

Входными данными для нашего анализатора является файл с программным кодом на модельном языке. Пример программного кода представлен ниже:

program var  
{identifiers}  
a, b, i: int;  
c, d: float;  
  
{operators}  
begin  
{assignment}  
a as 6;  
b as 8;  
{read}  
read(c);  
d as c plus 6;  
{write}  
write(d);  
  
{if-then-else}  
if d GT c  
then c as c mult 2  
else c as 0.0:  
write(c);  
  
{for loop}  
for i as 0 to 10 do  
c as c plus i:  
d as d plus i;  
  
write(c);  
write(d);  
  
while c LE d  
do c as c plus 1;  
  
end.

# Спецификация выходных данных

Выходными данными для нашего анализатора является список, состоящий из номера таблицы и номера записи в ней через запятую, а так же сообщение об ошибке или об успешном выполнении. (Рис. 2).

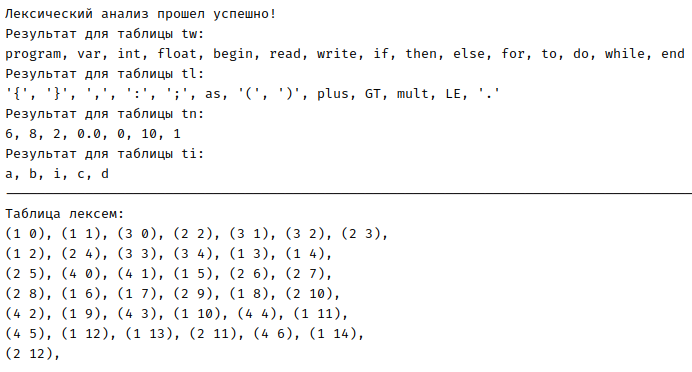


Рис. 2 – Выходные данные анализатор

# Разработка алгоритма решения задач

# Укрупненная схема алгоритма программного средства

Укрупненная схема алгоритма программного средства представлена на рисунке 3.

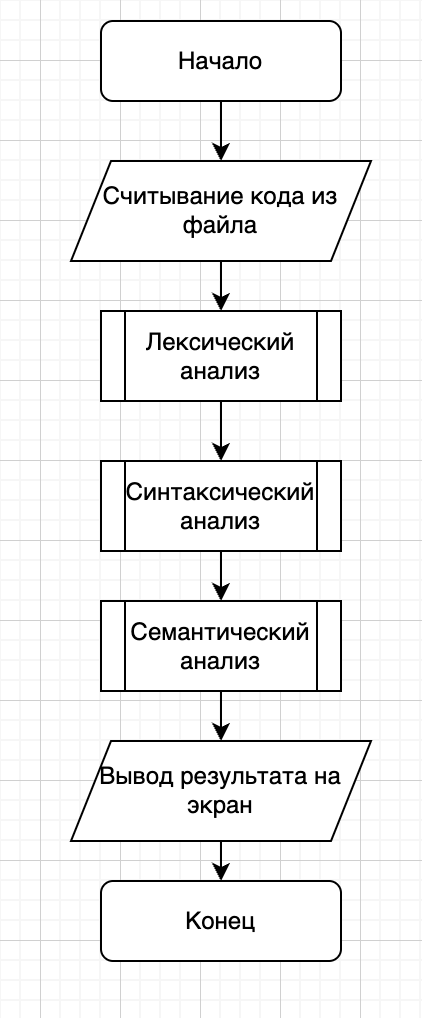


Рис. 3 - Блок-схема алгоритма программного средстваетальная разработка алгоритмов отдельных подзадач

# Блок-схема лексического анализатора

Реализован прямой лексический анализатор, построенный на основе детерминированного автомата, объединяющего множество автоматов, распознающих отдельные лексемы. На каждом шаге автомат анализирует входной символ, переходя в соответствующее состояние и формируя токен, если текущая последовательность символов соответствует одному из шаблонов.

Шаблоны лексем реализованы в виде отдельных функций, каждая из которых отвечает за обработку определенного типа лексем (например, идентификаторов или чисел).

Алгоритм лексического анализатора:

1. Исходный текст программы анализируется построчно. Каждая строка обрабатывается символ за символом.
2. Если символ соответствует началу возможной лексемы (например, буква для идентификаторов или цифра для чисел), запускается сбор символов до формирования полной лексемы, которая затем сравнивается с шаблоном из списка допустимых лексем.
3. Для каждой завершенной лексемы создаётся токен в виде экземпляра перечисления (enum Token), который добавляется в список токенов.
4. Результатом работы лексического анализатора является последовательность токенов, представляющая входной текст программы. Эта последовательность передаётся синтаксическому и семантическому анализаторам для дальнейшей обработки.

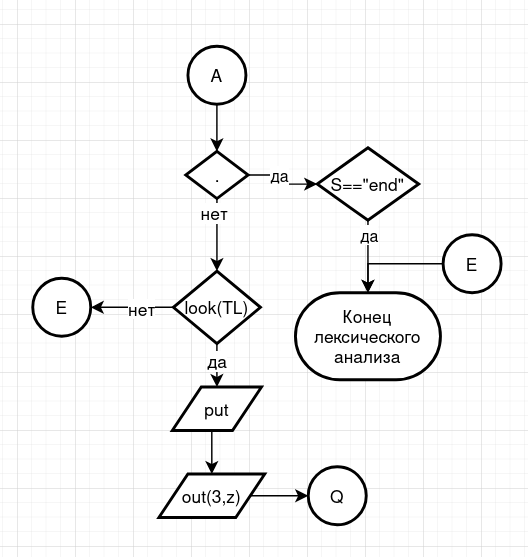
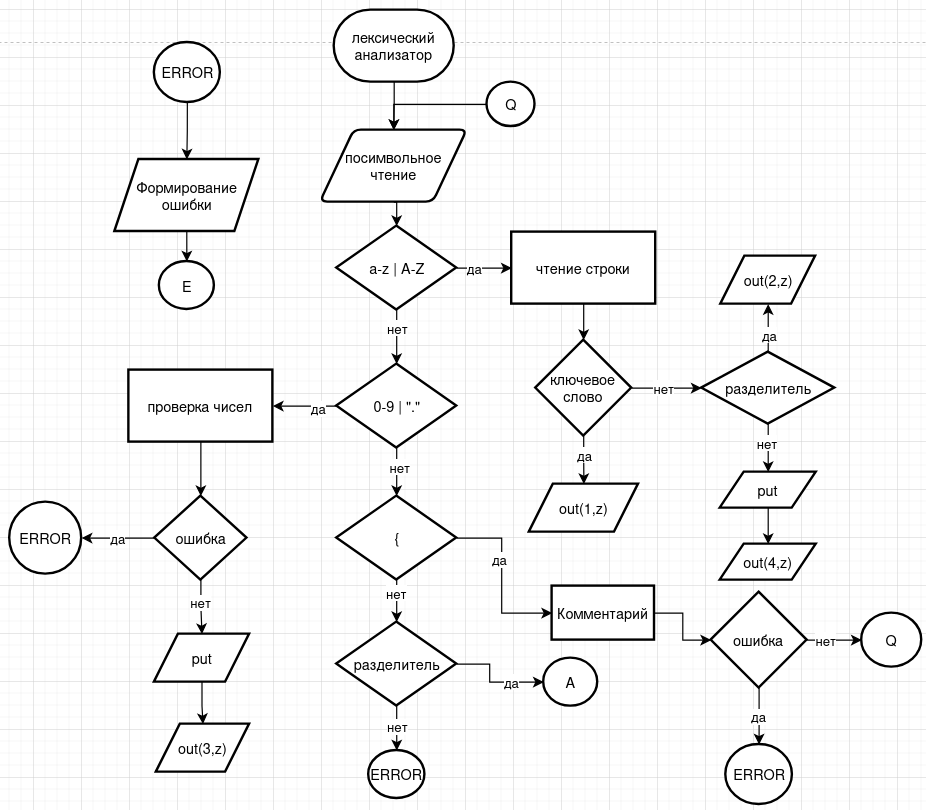


Рис. 4 – Блок-схема лексического анализатора

# Блок-схема синтаксического анализатора

Задача синтаксического анализатора: провести разбор текста программы, сопоставив его с эталонными шаблонами, данным в описании языка.

Для синтаксического разбора используются контекстно-свободные грамматики (КС-грамматика). Один из эффективных методов синтаксического анализа – метод рекурсивного спуска. В основе метода рекурсивного спуска лежит левосторонний разбор строки языка. Исходной сентенциальной формой является начальный символ грамматики, а целевой – заданная строка языка. На каждом шаге разбора правило грамматики применяется к самому левому не терминалу сентенции. Данный процесс соответствует построению дерева разбора цепочки сверху вниз (от корня к листьям).

Алгоритм синтаксического анализа:

1. Исходный текст программы разбивается на фразы по разделителю ;. Каждая фраза представляет собой отдельное выражение или оператор, который подлежит синтаксическому анализу. Каждая полученная фраза проверяется на соответствие ключевому слову операции в соответствии с шаблоном из списка операций;
2. Каждая фраза преобразуется в последовательность токенов с использованием результатов работы лексического анализатора. Эти токены затем проверяются на соответствие синтаксической грамматике.
3. Во время синтаксического анализа каждый токен фразы проверяется на соответствие шаблону, описанному в грамматике. Для каждого типа лексемы существуют соответствующие функции, которые проверяют структуру токенов и их соответствие правилам грамматики.
4. Каждый нетерминальный символ грамматики имеет свою процедуру. Эти процедуры отвечают за разбор подцепочек в тексте, выводящих из данного нетерминала. Если подцепочка не может быть найдена или не

соответствует грамматике, вызывается функция обработки ошибок, которая сообщает о синтаксической ошибке и завершает разбор.

1. Если подцепочка успешно найдена, процедура завершает свою работу и возвращает управление в точку вызова. Таким образом, процедура рекурсивного спуска позволяет постепенно разбирать программу с применением грамматических правил.
2. В процессе синтаксического анализа строится абстрактное синтаксическое дерево (AST), которое представляет собой структуру данных, отражающую синтаксическую структуру программы. Каждый узел дерева соответствует определённой грамматической конструкции, например, выражению, оператору или условию.
3. Результатом работы синтаксического анализатора являются: Абстрактное синтаксическое дерево (AST), которое содержит синтаксическое представление программы, а так же заполненные справочники присвоений значений переменным, а также справочники математических и условных выражений, которые далее передаются семантическому анализатору.

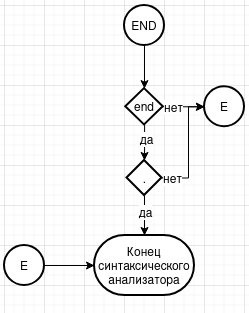
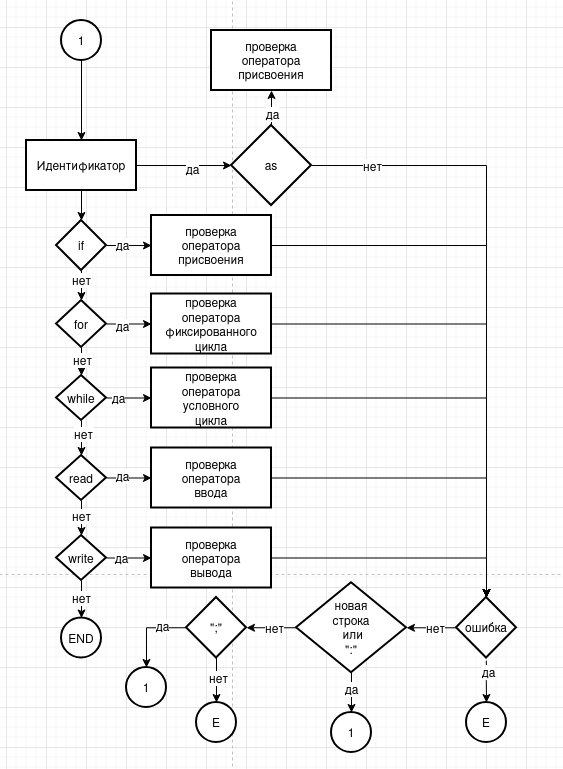
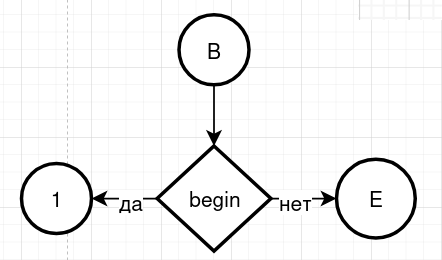
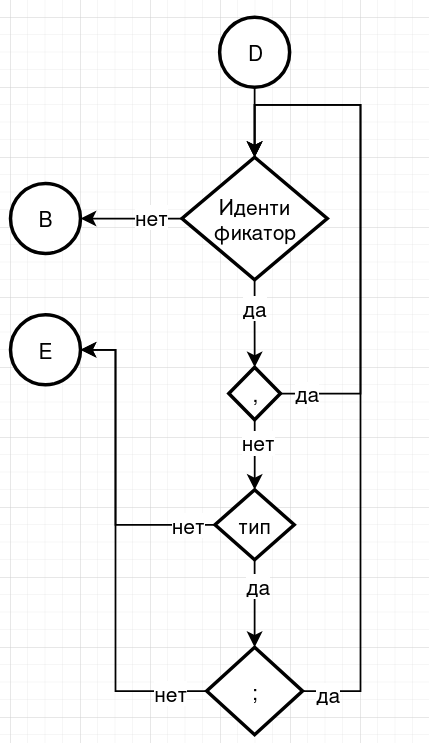
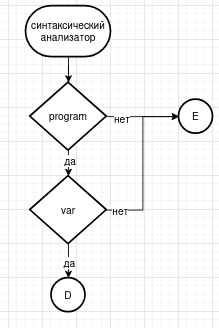


Рис. 5 – Блок-схема синтаксического анализатора

# Блок-схема семантического анализатора

Семантический анализ включает в себя проверку контекстно- зависимых условий, также называемых семантическими соглашениями, которые не могут быть полностью описаны контекстно-свободной грамматикой. Эти условия касаются трех основных аспектов: обработки описаний, анализа выражений и проверки правильности операторов.

Семантический анализатор осуществляет проверку типов при выполнении операций, включая арифметические, логические и присваивания с использованием списков выражений, полученных от синтаксического анализатора и справочников переменных и констант, полученных от лексического анализатора.

Использование справочников для хранения информации о переменных обеспечивает быстрый доступ к типам и значениям переменных.

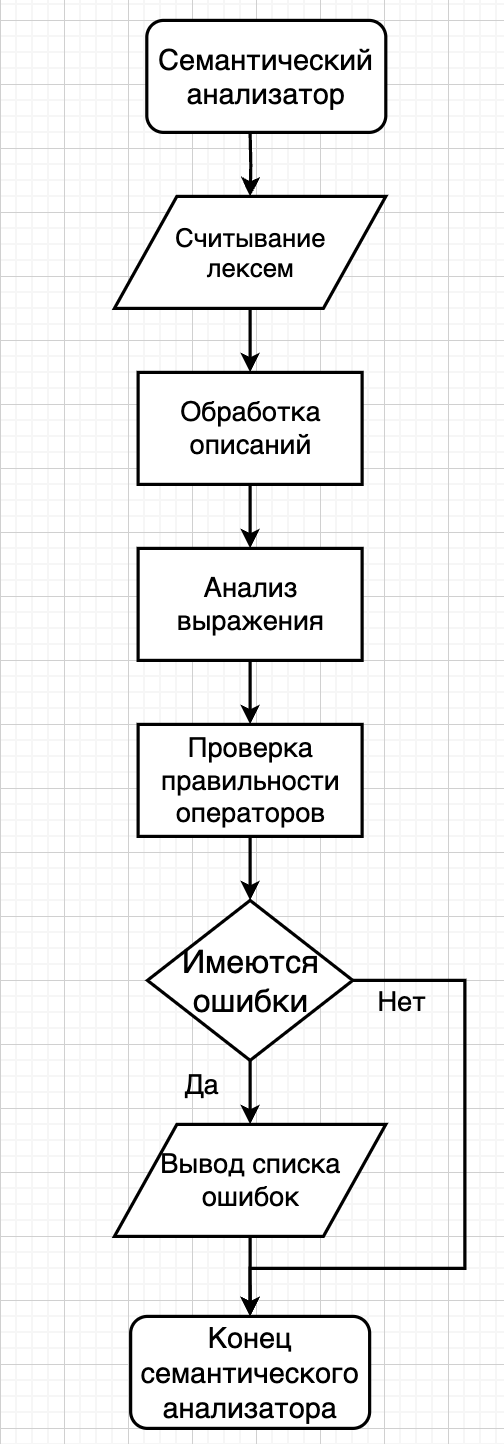


Рис. 6 – Блок-схема семантического анализатора

# Работа с программным средством

# Описание интерфейса

Примеры работы программы на разных наборах исходного кода проиллюстрированы на скриншотах в части 6.2. Интерфейс программы состоит из скрипта запуска программы(main.py), файла с исходным кодом и файлами таблиц: tw и tl.

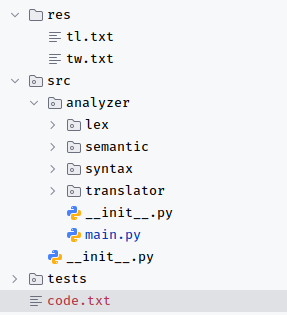


Рис. 7 – Главный интерфейс

# Перечень ошибок

## Лексические ошибки:

1. Недопустимый входной символ
2. Неожиданный конец файла

## Синтаксические ошибки:

1. Неожиданная лексема

## Семантические ошибки:

1. Повторное объявление идентификатора
2. Несоответствие типов данных
3. Идентификатор не объявлен

# Пример работы компилятора

Правильное выполнение работы компилятора представлено на рисунке 8.

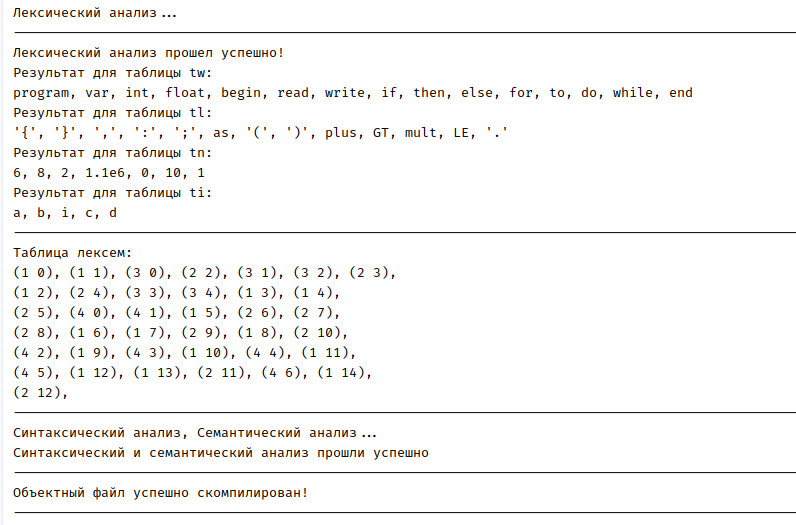


Рис. 8 - Пример правильной работы компилятора

## Лексические ошибки:

Ошибка “Недопустимый входной символ” приведена на рис. 9.

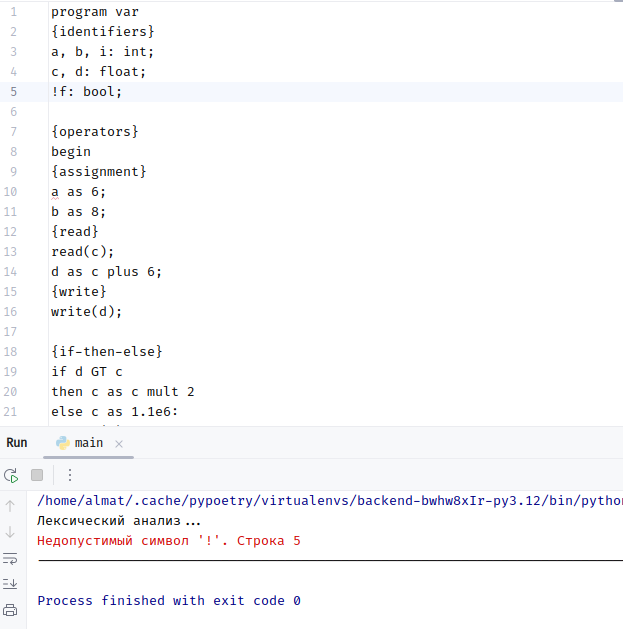


Рис. 9 - Ошибка “Недопустимый входной символ”

Ошибка “Неожиданный конец файла” приведена на рис. 10.

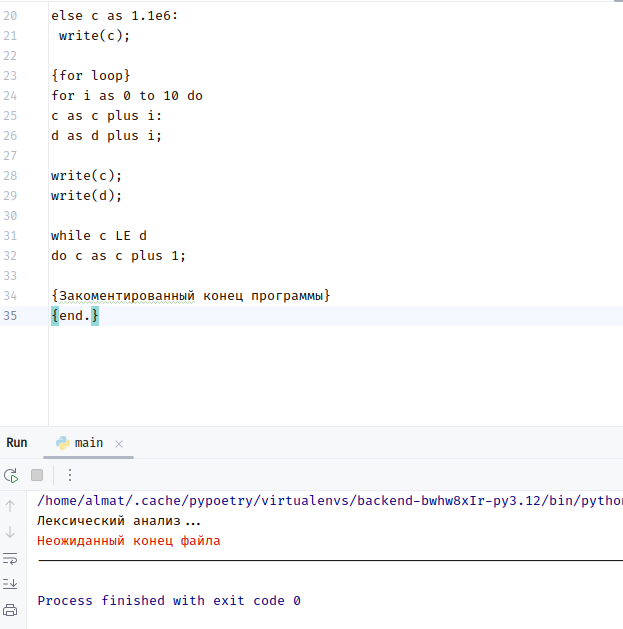


Рис. 10 - Ошибка “Неожиданный конец файла”

## Синтаксические ошибки:

Ошибка “Неожиданная лексема“ приведена на рис. 11.

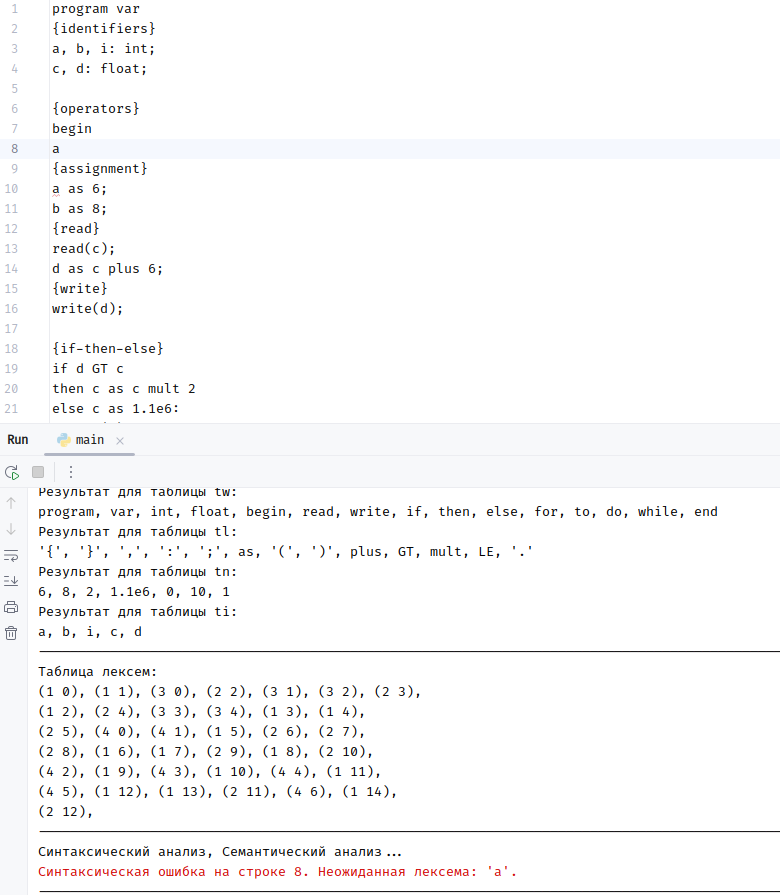


Рис. 11 - Ошибка “Неожиданная лексема“

## Семантические ошибки:

Ошибка “Повторное объявление идентификатора“ приведена на рис.

12.

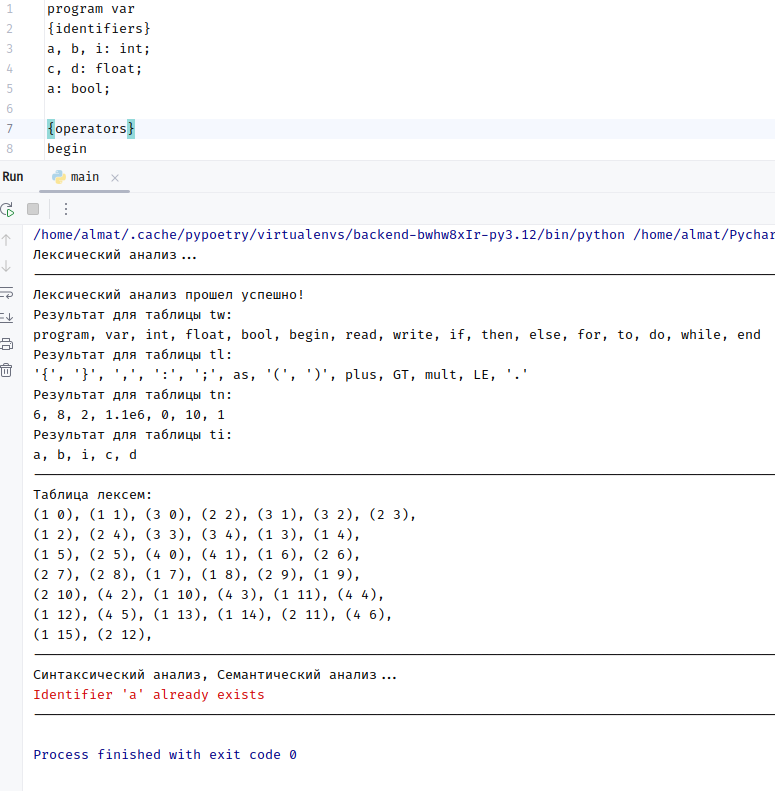


Рис. 12 - Ошибка “Повторное объявление идентификатора“

Ошибка “Несоответствие типов данных” приведена на рис. 13.

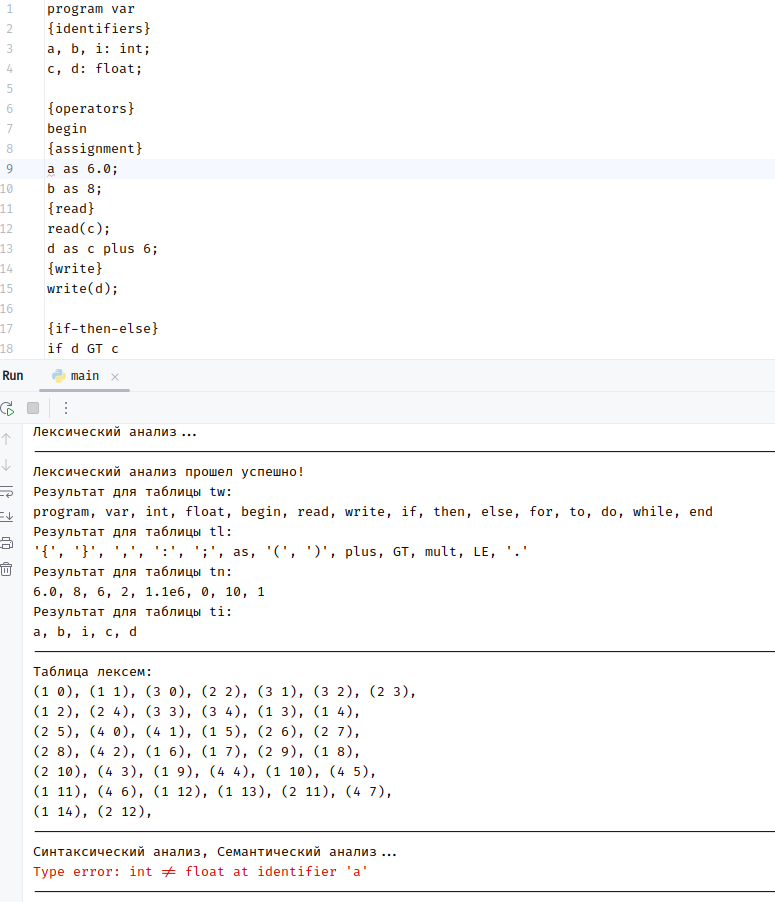


Рис. 13 - Ошибка “Несоответствие типов данных”

Ошибка “Идентификатор не объявлен” приведена на рис. 14.

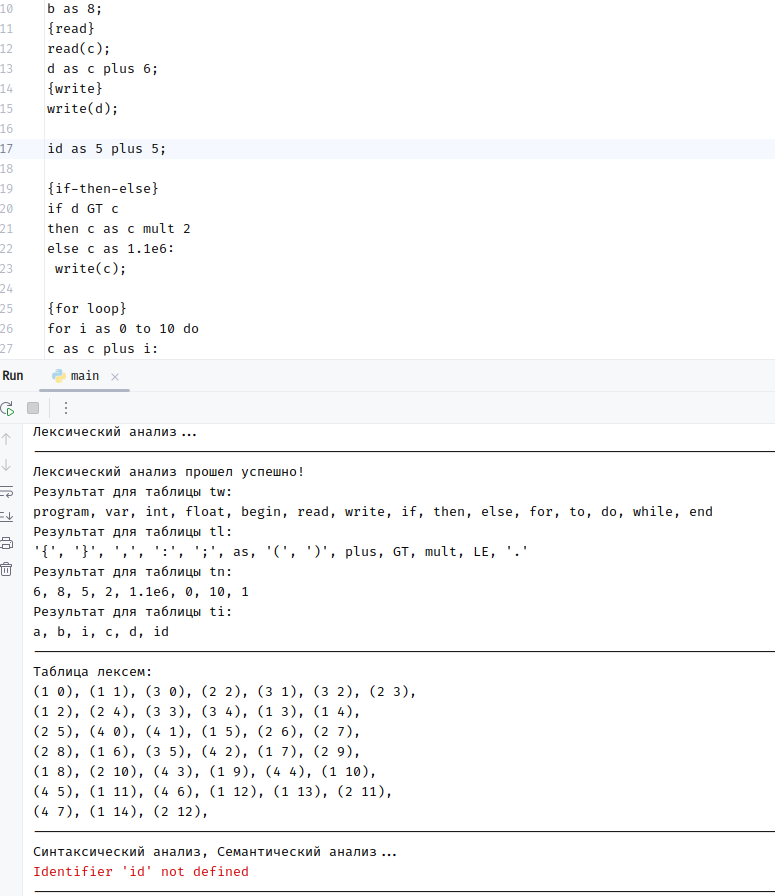


Рис. 14 - Ошибка “Идентификатор не объявлен”

# Генерация объектного файла

Алгоритм генерации объектного файла можно разбить на несколько ключевых этапов: разбор исходного кода, создание промежуточного представления (IR), генерация ассемблерного кода, и генерация и запись объектного файла. Каждый из этих этапов выполняет важную роль в процессе компиляции.

1. На первом этапе исходный код программы анализируется с использованием лексического анализатора, который разбивает его на токены. Токены — это минимальные единицы текста (лексемы), такие как ключевые слова, идентификаторы, числа и разделители. Эти токены используются на следующем этапе — синтаксическом анализе.
2. Синтаксический анализатор использует грамматику языка программирования для построения абстрактного синтаксического дерева (AST). AST — это структурированное представление программы, где каждый узел дерева соответствует синтаксическому элементу языка. С помощью AST можно легко визуализировать структуру программы и убедиться в её синтаксической корректности.
3. На основе AST строится промежуточное представление (IR), которое служит связующим звеном между высокоуровневым исходным кодом и низкоуровневым машинным кодом.
4. После создания IR мы можем вручную заполнить ассемблерный файл либо генерируется ассемблерный код для целевой архитектуры в результате выполнения работы компилятора. Ассемблерный код — это низкоуровневое представление программы, которое всё ещё зависит от целевой машины, но ближе к машинному коду. На этом этапе программа уже может быть скомпилирована в машинный код.
5. На последнем этапе генерируется объектный файл. Это бинарный файл, содержащий машинный код программы, данные и метаинформацию. Важно, что объектный файл включает не только скомпилированный код, но и все необходимые структуры, такие как таблицы символов и релокации, для дальнейшего связывания и загрузки программы.

Данный код может быть представлен в виде ассемблерного кода (рис.

15) или непосредственно в бинарном формате (рис. 16).

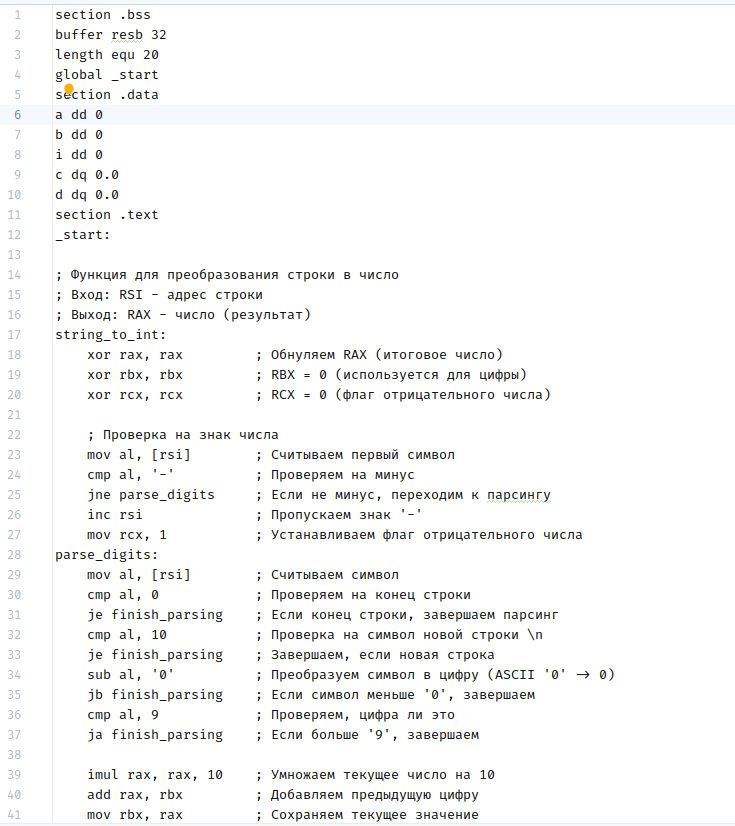


Рис. 15 – Результирующий ассемблер исходного кода

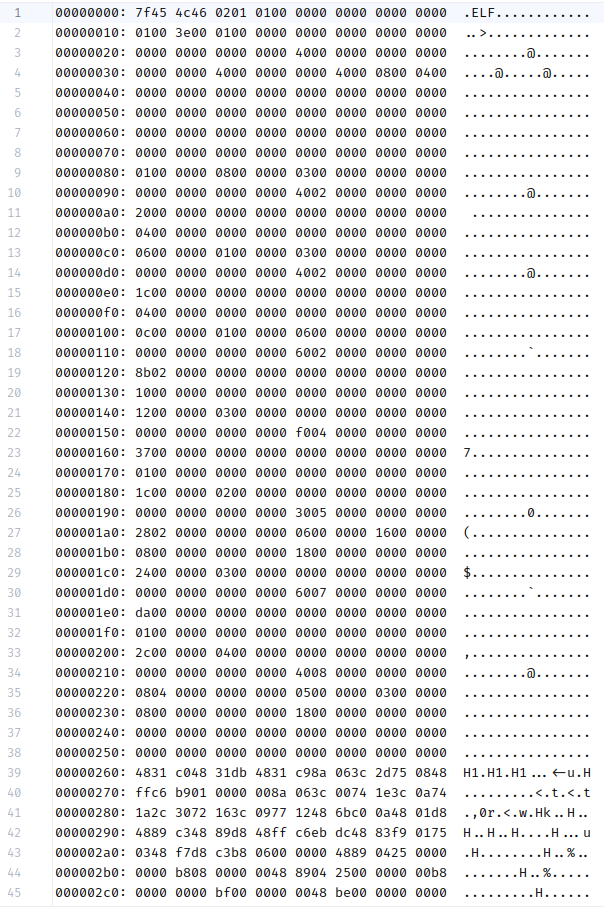


Рис. 16 – Результирующий машинный код исходного кода

# Заключение

В рамках выполнения задания по курсовой работе был разработан компилятор модельного языка, который осуществляет лексический, синтаксический и семантический анализы формального языка.

При подготовке к выполнению курсовой работы был проведен анализ заданного по варианту модельного языка, составлены таблицы ключевых слов и разделителей, описаны диаграммы Вирта и составлена диаграмма состояний с действиями для распознавания и формирования лексем языка. Для наглядного отображения процесса работы программы описаны блок- схемы лексического, синтаксического, семантического анализаторов.

В процессе выполнения курсовой работы проведено изучение методов формального лексического анализа для языка программирования, определенного в рамках задания по варианту.

По итогам выполнения курсовой работы и её завершения были изучены методы реализации лексических, синтаксических и семантических анализаторов.

# Список использованных источников

1. Ишакова Е.Н. Теория языков программирования и методов трансляции: учебное пособие / Е.Н. Ишакова. – Оренбург: ИПК ГОУ ОГУ, 2007. – 137 с. [Текст]
2. Калинин А.Г. Универсальные языки программирования: Семантический подход [Текст] / А.Г. Калинин, И.В. Мацкевич. – М.: Радио и связь, 1991. - 398, [1] с.: ил.; 21 см. - Библиогр.: с. 395-398.
3. Афанасьев А.Н. Формальные языки и грамматики: Учебное пособие. – Ульяновск: УлГТУ, 1997. – 84с. [Текст]
4. Братчиков И.Л. Синтаксис языков программирования / Под ред. С.С. Лаврова. – М.: Наука, 1975. - 262с. [Текст]
5. Волкова И.А., Руденко Т.В. Формальные языки и грамматики. Элементы теории трансляции. – М.: Диалог-МГУ, 1999. – 62с. [Текст]
6. Жаков В.И., Коровинский В.В., Фильчаков В.В. Синтаксический анализ и генерация кода. – СПб.: ГААП, 1993. – 26с. [Текст]
7. Рейуорд-Смит В. Теория формальных языков. Вводный курс: Пер. с англ. – М.: Радио и связь, 1988. – 128с. [Текст]
8. Серебряков В.И. Лекции по конструированию компиляторов. – М.: МГУ, 1997. – 171с. [Текст]

use

# Приложение А – Текст программы

import os  
import re  
  
  
from src.analyzer.lex.file import exists, is\_tw, is\_tl, write\_lex  
from src.analyzer.lex.writer import write, flush  
  
os.makedirs(os.path.join(os.getcwd(), 'out'), exist\_ok=True)  
  
flush(os.path.join(os.getcwd(),'out','lex','ti.txt'))  
flush(os.path.join(os.getcwd(),"out","lex","tl.txt"))  
flush(os.path.join(os.getcwd(),"out","lex","tw.txt"))  
flush(os.path.join(os.getcwd(),"out","lex","tn.txt"))  
flush(os.path.join(os.getcwd(),"out","lex","lex.txt"))  
  
result\_table = {  
 "tw": [],  
 "tl": [],  
 "tn": [],  
 "ti": []  
}  
  
lines\_count = 0  
tokens = (  
 # key words  
 'PROGRAM',  
 'VAR',  
 'BEGIN',  
 'INT',  
 'FLOAT',  
 'BOOL',  
 'IF',  
 'THEN',  
 'ELSE',  
 'TO',  
 'DO',  
 'FOR',  
 'WHILE',  
 'WRITE',  
 'READ',  
 'END',  
 'AS',  
 'TRUE',  
 'FALSE',  
  
 # delimiters  
 "PLUS",  
 "MIN",  
 "MULT",  
 "DIV",  
 "LPAREN",  
 "RPAREN",  
 # "LBRACE",  
 # "RBRACE",  
 "DOT",  
 "COMMA",  
 "SEMICOLON",  
 "COLON",  
 'OR',  
 'AND',  
 'LT',  
 'LE',  
 'EQ',  
 'NEQ',  
 'GT',  
 'GE',  
 'NOT',  
 # "NEWLINE",  
 # "DELIMITER",  
 # other  
 "NUMBER",  
 'ID',  
  
 # number  
 "NEWLINE",  
)  
  
def t\_newline(t):  
 r"""\n+"""  
 t.lexer.lineno += len(t.value)  
  
def t\_ignore\_COMMENT(t):  
 r"\{[^}]\*\}"  
 path\_to\_table = os.path.join(os.getcwd(), 'out','lex','tl.txt')  
  
 bracers = ["{", "}"]  
 for i in bracers:  
 if not exists(i, path\_to\_table):  
 write(i, path\_to\_table)  
 result\_table['tl'].append(f"'{i}'")  
  
  
  
t\_NEWLINE = r'\n'  
t\_ignore = ' \t'  
literals = [  
 '(',  
 ')',  
 ';',  
 ":",  
 ",",  
 '.',  
 '~',  
 '{',  
 '}'  
 , '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'  
 , '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'  
 , '0'  
 , '1'  
 , '2'  
 , '3'  
 , '4'  
 , '5'  
 , '6'  
 , '7'  
 , '8'  
 , '9'  
 ]  
  
precedence = (  
 ('nonassoc', 'LT', 'GT', 'LE', 'GE', 'EQ', 'NEQ'), # Nonassociative operators  
 ('left', 'PLUS', 'MIN'),  
 ('left', 'MULT', 'DIV'),  
 ('right', 'NOT'), # Unary minus operator  
)  
  
  
  
reserved = {  
 "as": 'AS',  
 "plus": 'PLUS',  
 "min": 'MIN',  
 "mult": 'MULT',  
 "div": 'DIV',  
 "LT": 'LT',  
 "GT": 'GT',  
 "LE": 'LE',  
 "EQ": 'EQ',  
 "NEQ": 'NEQ',  
 "begin": 'BEGIN',  
 "end": 'END',  
 "var": 'VAR',  
 "program": 'PROGRAM',  
 "write": "WRITE",  
 "read": "READ",  
 'do': 'DO',  
 'to': 'TO',  
 'if': 'IF',  
 'then': 'THEN',  
 'else': 'ELSE',  
 'for': 'FOR',  
 'while': 'WHILE',  
 'int': 'INT',  
 'float': 'FLOAT',  
 'bool': 'BOOL',  
 'true': 'TRUE',  
 'false': 'FALSE',  
 'or': 'OR',  
 'and': 'AND'  
}  
  
def t\_NUMBER(t):  
 r"""[0-1]+[bB]|  
 [0-7]+[oO]|  
 [0-9a-fA-F]+[hH]|  
 (?:\d\*\.\d+|\d+)(?:[eE][+-]?\d+)?[dD]?  
 """  
 path\_to\_table = os.getcwd()+"/out/lex/tn.txt"  
 if not exists(t.value, path\_to\_table):  
 write(t.value, path\_to\_table)  
 write\_lex(t.value, 'tn')  
 result\_table["tn"].append(t.value)  
 return t  
def t\_DELIMITER(t):  
 r"""\.|;|:|,|~|\(|\)"""  
 types = {  
 ".": "DOT",  
 ":": "COLON",  
 ";": "SEMICOLON",  
 ",": "COMMA",  
 "~": "NOT",  
 "(": "LPAREN",  
 ")": "RPAREN",  
 }  
 t.type = types[t.value]  
 if t.type == 'DOT':  
 t.lexer.lexstate = "END"  
 path\_to\_table = os.path.join(os.getcwd(), 'out','lex','tl.txt')  
 if not exists(t.value, path\_to\_table):  
 write(t.value, path\_to\_table)  
 write\_lex(t.value, 'tl')  
 result\_table['tl'].append(f"'{t.value}'")  
 return t  
def t\_ID(t):  
 r"[a-zA-Z][a-zA-Z0-9]\*"  
 t.type = reserved.get(t.value, 'ID')  
  
 tables\_path = os.path.join(os.getcwd(), "out", "lex")  
 ti = os.path.join(tables\_path,'ti.txt')  
 tw = os.path.join(tables\_path,'tw.txt')  
 tl = os.path.join(tables\_path,'tl.txt')  
  
 if not exists(t.value, ti):  
 if t.type == "ID":  
 write(t.value, ti)  
 write\_lex(t.value, 'ti')  
 result\_table['ti'].append(t.value)  
 else:  
 if is\_tw(t.value) and not exists(t.value, tw):  
 write(t.value, tw)  
 write\_lex(t.value, 'tw')  
 result\_table['tw'].append(t.value)  
 elif is\_tl(t.value) and not exists(t.value, tl):  
 write(t.value, tl)  
 write\_lex(t.value, 'tl')  
 result\_table['tl'].append(t.value)  
  
 return t  
def t\_error(t):  
 t.lexer.skip(1)  
 raise SyntaxError(f"Недопустимый символ '{t.value[0]}'. Строка {t.lineno}")  
  
  
def t\_eof(t):  
 if t.lexer.lexstate != 'END':  
 raise SyntaxError(f"Неожиданный конец файла")

# Приложение Б – Генерация объектного файла

dadwa  
start = '\_P'  
instructions: list[Node] = []  
  
  
def p\_error(p: LexToken):  
 if p is None:  
 raise SyntaxError('Syntax error EOF')  
 else:  
 line\_number = p.lineno - len(open(f'{os.getcwd()}/code.txt', 'r').readlines()) -1  
 raise SyntaxError(f"Синтаксическая ошибка на строке {line\_number}. Неожиданная лексема: '{p.value}'.")  
  
  
def p\_p(p):  
 """\_P : PROGRAM \_D \_B DOT"""  
 p[0] = f'{p[1]} {p[2]} {p[3]} {p[4]}'  
 lol = instructions  
 with open('output.asm', 'w') as f:  
 for i in instructions:  
 f.write(i.generate\_code()+'\n')  
 # print(i.generate\_code())  
def p\_b(p):  
 """\_B : BEGIN \_O END"""  
 p[0] = f'{p[1]} {p[2]} {p[3]}'  
 instructions.append(  
 EndNode()  
 )  
  
  
def p\_d(p):  
 """  
 \_D : VAR D1  
 | VAR  
 """  
 p[0] = p[1]  
 instructions.append(BufferSection())  
 instructions.append(VarSectionNode())  
 for i in founded:  
 instructions.append(DefinitionNode(i.name, i.type))  
 instructions.append(TextSectionNode())  
 instructions.append(UtilsFuncsNode())  
  
  
  
def p\_d1\_rec(p):  
 """D1 : \_I D1"""  
 p[0] = f'{p[1]} {p[2]}'  
def p\_d1(p):  
 """D1 : \_I"""  
 p[0] = p[1]  
  
  
def p\_i(p):  
 """\_I : I1 COLON \_T SEMICOLON"""  
 p[0] = f'{p[1]} {p[2]} {p[3]} {p[4]}'  
 ids = re.split(r'[ \t,]+', p[1])  
 for i in ids:  
 add\_id(  
 Identifier(name=i, type=p[3])  
 )  
 for i in founded:  
 if i.count > 1:  
 raise AlreadyExistsError(f"Identifier '{i.name}' already exists")  
  
 if debug:  
 print(f'ids: {[f.name for f in founded]}')  
  
  
def p\_i1(p):  
 """I1 : I2"""  
 p[0] = p[1]  
  
  
def p\_i1\_comma(p):  
 """I1 : I1 COMMA I2"""  
 p[0] = f'{p[1]}{p[2]}{p[3]}'  
def p\_i2(p: YaccProduction):  
 """I2 : ID"""  
 p[0] = p[1]  
  
  
def p\_o(p):  
 """\_O : O1"""  
 p[0] = p[1]  
  
  
def p\_os(p):  
 """\_O : O1 SEMICOLON \_O"""  
 p[0] = f'{p[1]}{p[2]}{p[3]}'  
def p\_o\_empty(p):  
 """\_O : """  
 pass  
def p\_o2(p):  
 """  
 O2 : \_S  
 """  
 l = p[1]  
 p[0] = p[1]  
  
  
def p\_s(p):  
 """  
 \_S : O1  
 """  
 l = p[1]  
 p[0] = p[1]  
  
  
def p\_s\_rec(p):  
 """\_S : \_S S1 O1"""  
 p[0] = f'{p[1]} {p[3]}'  
 p[0] = {  
 'nodes': p[1]['nodes']  
 }  
 temp = p[0]  
 p[0]['nodes'].extend(p[3]['nodes'])  
  
  
def p\_o1(p):  
 """  
 O1 : \_A  
 | \_F  
 | \_W  
 | \_Y  
 | \_U  
 | \_V  
 """  
 p[0] = p[1]  
 if debug:  
 print(f"O1: {p[0]}")  
 for i in p[1]['nodes']:  
 instructions.append(i)  
  
  
def p\_s1(p):  
 """  
 S1 : COLON  
 """  
 p[0] = p[1]  
  
  
def p\_a(p):  
 """  
 \_A : I2 AS \_E  
 """  
 p[0] = f'{p[1]} {p[2]} {p[3].value}'  
 IdChecker(p[1]).check()  
 AssignmentChecker().check(p[1], p[3].type)  
  
 p[0] = {  
 "nodes": []  
 }  
 p[0]['nodes'].append(AssignNode(p[1]))  
  
  
if\_counter = 0  
def p\_f(p):  
 """  
 \_F : F1 ELSE O2  
 """  
 p[0] = f'{p[1]} {p[2]} {p[3]}'  
 p[0] = {  
 "nodes": p[1]['nodes']  
 }  
 global if\_counter  
 p[0]['nodes'].append(ElseNode(if\_counter))  
 p[0]['nodes'].extend(p[3]['nodes'])  
 p[0]['nodes'].append(ElseEndNode(if\_counter))  
 p[0]['nodes'].append(EndIfNode(if\_counter))  
  
 if\_counter += 1  
def p\_f\_f1(p):  
 """\_F : F1"""  
 p[0] = p[1]  
  
  
def p\_f1(p):  
 """  
 F1 : IF \_E THEN O2  
 """  
 global if\_counter  
 p[0] = f'{p[1]} {p[2].value} {p[3]} {p[4]}'  
 if debug:  
 print(f'F1: {p[0]}')  
  
 if p[2].type != 'bool':  
 raise TypeError(f"Type error: {p[2]} != bool at if statement")  
  
 p[0] = {  
 "nodes": [IfNode(if\_counter)]  
 }  
 p[0]['nodes'].extend(p[4]['nodes'])  
 p[0]['nodes'].append(ThenEndNode(if\_counter))  
 # w = p[0]  
 if debug:  
 print(f"F1: {p[0]}")  
  
for\_counter = 0  
def p\_w(p):  
 """\_W : FOR \_A TO \_E DO O2 """  
 p[0] = f'{p[1]} {p[2]} {p[3]} {p[4]} {p[5]} {p[6]}'  
 global for\_counter  
 p[0] = {  
 "nodes": [  
 \*p[2]['nodes'],  
 ForAssignNode(),  
 # \*p[4]['nodes'],  
 ForToNode(),  
 ForLoopNode(for\_counter),  
 ForConditionNode(for\_counter),  
 \*p[6]['nodes'],  
 ForIncrementNode(for\_counter),  
 ForEndNode(for\_counter),  
 ]  
 }  
 for\_counter += 1  
while\_counter = 0  
def p\_y(p):  
 """  
 \_Y : WHILE \_E DO O2  
 """  
 p[0] = f'{p[1]} {p[2]} {p[3]} {p[4]}'  
 if debug:  
 print(f"Y: {p[0]}")  
  
 if p[2].type != 'bool':  
 raise TypeError(f"Type error: {p[2]} != bool at while statement")  
  
 global while\_counter  
 p[0] = {  
 "nodes": [  
 ForAssignNode(),  
 WhileLoopNode(while\_counter),  
 WhileConditionNode(while\_counter),  
 \*p[4]['nodes'],  
 WhileEndNode(while\_counter)  
 ]  
 }  
 while\_counter += 1  
def p\_u(p):  
 """  
 \_U : READ LPAREN I1 RPAREN  
 """  
 p[0] = f'{p[1]} {p[2]} {p[3]} {p[4]}'  
 IdChecker(p[3]).check()  
 if debug:  
 print(f"READ: {p[0]}")  
  
 p[0] = {  
 "nodes": []  
 }  
 for i in re.split(",", p[3]):  
 p[0]['nodes'].append(ReadNode(i))  
  
  
def p\_v(p):  
 """  
 \_V : WRITE LPAREN E1 RPAREN  
 """  
 p[0] = f'{p[1]} {p[2]} {p[3]} {p[4]}'  
 p[0] = {  
 "nodes": []  
 }  
  
 # for i in re.split(",", p[3].value):  
 p[0]['nodes'].append(WriteNode(p[3].value))  
  
  
def p\_e1(p):  
 """E1 : E1 COMMA \_E  
 | \_E  
 """  
 if len(p) > 2:  
 p[0] = f'{p[1]} {p[2]} {p[3]}'  
 # p[0] = {  
 # 'nodes': p[1]['nodes'].extend(p[3]['nodes'])  
 # }  
 else:  
 l = p[1]  
 p[0] = p[1]  
  
 # p[0] = {  
 # "nodes": p[1]['nodes']  
 # }  
##  
ип  
# #  
def p\_e(p):  
 """  
 \_E : \_Z  
 """  
 # p[0] = to\_typed\_value(p[1])  
 # print(f"E: {p[0]}")  
 p[0] = p[1]  
  
  
def p\_e\_rec(p):  
 """\_E : \_E Z1 \_Z"""  
 p[0] = f"{p[1].value} {p[2]} {p[3].value}"  
 p[0] = Expression(TypeChecker().compare(p[1].type, p[3].type, p[2]), p[0])  
  
 sign = {  
 "LT": LTNode,  
 "GT": GTNode,  
 "LE": LENode,  
 "GE": GENode,  
 "NEQ": NEQNode,  
 "EQ": EQNode,  
 }  
  
 left = p[1].value  
 right = p[3].value  
  
 instructions.append(  
 ValueNode(left) if not IdentifierChecker(left).exists() else IdentifierNode(left)  
 )  
 instructions.append(  
 sign[p[2]](f'[{right}]' if IdentifierChecker(right).exists() else right)  
 )  
  
  
def p\_z(p):  
 """\_Z : \_J"""  
 p[0] = p[1]  
 if debug:  
 print(f"Z: {[f for f in p]}")  
  
  
def p\_z\_rec(p):  
 """\_Z : \_Z J1 \_J"""  
 p[0] = f"{p[1].value} {p[2]} {p[3].value}"  
 p[0] = Expression(TypeChecker().compare(p[1].type, p[3].type, p[2]), p[0])  
  
 sign = {  
 "plus": AddNode,  
 "min": MinNode,  
 "or": OrNode,  
 }  
  
 instructions.append(  
 sign[p[2]](p[3].value)  
 )  
 if debug:  
 print(f"Z rec: {p[0]}")  
  
  
def p\_j(p):  
 """\_J : \_M"""  
 p[0] = p[1]  
 if debug:  
 print(f"J: {p[0]}")  
  
  
def p\_j\_rec(p):  
 """\_J : \_J M1 \_M"""  
 p[0] = f"{p[1].value} {p[2]} {p[3].value}"  
 # p[0] = eval(p[0])  
 if debug:  
 print(f"rec J: {p[0]}")  
 p[0] = Expression(TypeChecker().compare(p[1].type, p[3].type, p[2]), p[0])  
  
 sign = {  
 'mult': MultNode,  
 'div': DivNode,  
 'and': AndNode  
 }  
 instructions.extend([  
 sign[p[2]](p[3].value)  
 ])  
  
  
def p\_m1(p):  
 """  
 M1 : MULT  
 | DIV  
 | AND  
 """  
 sign = {  
 'mult': '\*',  
 'div': '/',  
 'and': '&',  
 }  
 # p[0] = sign[p[1]]  
 p[0] = p[1]  
 if debug:  
 print(f"M1: p[1]={p[1]}")  
  
  
def p\_j1(p: YaccProduction):  
 """  
 J1 : PLUS  
 | MIN  
 | OR  
 """  
 p[0] = p[1]  
  
  
def p\_z1(p):  
 """  
 Z1 : NEQ  
 | EQ  
 | LT  
 | LE  
 | GT  
 | GE  
 """  
 if debug:  
 print(f"Z1: {[f for f in p]}")  
 op = {  
 'NEQ': '!=',  
 "EQ": "==",  
 "LT": "<",  
 "LE": "<=",  
 "GT": ">",  
 "GE": ">=",  
 }  
 p[0] = p[1]  
def p\_m(p):  
 """  
 \_M : \_N  
 | \_L  
 | NOT\_M  
 | PAREN\_M  
 """  
 p[0] = p[1]  
 if debug:  
 print(f"M: p[1]={p[1]}")  
  
  
def p\_id(p):  
 """\_M : I2"""  
 IdChecker(p[1]).check()  
 \_id = get\_id\_by\_name(p[1])  
 if debug:  
 print(f"ID in expression: {\_id}")  
 p[0] = \_id.type  
 p[0] = Expression(p[0], p[1])  
  
 instructions.append(IdentifierNode(p[1]))  
  
  
def p\_not\_m(p):  
 """  
 NOT\_M : NOT \_M  
 """  
 if p[2].type != 'bool':  
 raise TypeError(f"Expected: bool. Got type: {p[2]}'")  
  
 bools = {  
 "true": 1,  
 "false": 0  
 }  
  
 p[0] = Expression('bool', f'{p[1]}{bools[p[2].value]}')  
  
  
def p\_paren\_m(p):  
 """  
 PAREN\_M : LPAREN \_E RPAREN  
 """  
 p[0] = Expression(p[2].type, p[2].value)  
  
  
def p\_l(p):  
 """  
 \_L : TRUE  
 | FALSE  
 """  
 # p[0] = f'{p[1][:1].upper()}{p[1][1:]}'  
 p[0] = Expression('bool', p[1])  
  
 instructions.append(  
 ValueNode(p[1])  
 )  
  
  
def p\_n(p):  
 """\_N : NUMBER"""  
 p[0] = 'int'  
 n: str = p[1]  
 if debug:  
 print(f'value parse : {p[1]}')  
 p[0] = Expression(NumberTypeParser(n).get\_value\_type(), p[1])  
 if debug:  
 print(f'number: {p[0]}')  
  
 instructions.append(ValueNode(p[1]))  
def p\_t(p):  
 """  
 \_T : INT  
 | FLOAT  
 | BOOL  
 """  
 p[0] = p[1]  
  
  
import ast  
  
  
def to\_typed\_value(value):  
 try:  
 return ast.literal\_eval(value)  
 except (ValueError, SyntaxError):  
 return value