Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Владимирский государственный университет

имени Александра Григорьевича и Николая Григорьевича Столетовых»

(ВлГУ)

разработка компилятора подмножества

процедурно-ориентированного языка

Пояснительная записка

RU. 643.02068048.0001-01 81 01

# АННОТАЦИЯ

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.

Курсовая работа заключается в создании компилятора подмножества процедурно-ориентированного языка. С исходным кодом проекта можно ознакомиться, перейдя по ссылке: <https://github.com/Nat33333/comp>.

В пояснительной записке приведены требования к разрабатываемому проекту и описаны все стадии проектирования компилятора, а именно: построение лексического анализатора, создание синтаксического анализатора, проектирование генератора объектного кода и построение оптимизатора, а также приведены результаты тестирования разработанного компилятора.СОДЕРЖАНИЕ

[АННОТАЦИЯ 2](#_Toc10580246)

[СОДЕРЖАНИЕ 3](#_Toc10580247)

[ПРИЛОЖЕНИЕ А 4](#_Toc10580248)

**1 Задание**

Курсовая работа заключается в создании компилятора с заданного подмножества языка Паскаль с незначительными модификациями и упрощениями (полное описание входного и выходного языков дано далее в задании для каждого варианта). Результатами курсовой работы являются программная реализация заданного компилятора и пояснительная записка, оформленная в соответствии с требованиями ГОСТ.

Компилятор рекомендуется построить из следующих составных частей:

1. лексический анализатор;
2. синтаксический анализатор;
3. оптимизатор;
4. генератор ассемблерного или байт кода.

Входной язык компилятора должен удовлетворять следующим требованиям:

* входная программа начинается ключевым словом *program* и заканчивается ключевым словом *end.*;
* входная программа может быть разбита на строки произвольным образом, все пробелы и переводы строки должны игнорироваться компилятором;
* текст входной программы может содержать комментарии любой длины, которые должны игнорироваться компилятором;
* входная программа должна представлять собой единый модуль, но требуется предусмотреть вызов функций;
* должны быть предусмотрены следующие варианты операторов входной программы:

1) оператор присваивания вида *<переменная>:=<выражение>*;

2) условный оператор вида *if  <выражение> then begin <оператор> end*, либо *if <выражение> then begin <оператор> end else begin <оператор> end*;

2) составной оператор вида *begin … end*;

3) оператор цикла *for <переменная>:=<выражение> do begin <оператор> end;*

4) выражения в операторах могут содержать следующие операции (минимум):

5) арифметические операции сложения (*+*), вычитания (*-*), деления (/), умножения (\*);

6) операции сравнения меньше (*<*), больше (*>*), равно (*=*), неравно (<>), больше или равно (>=), меньше или равно (<=);

7) логические операции «и» (*and*), «или» (*or*), «нет» (*not*);

8) break-continue;

9) Для изменения приоритета операций должны использоваться круглые скобки.

В качестве выходного (результирующего) языка должен использоваться язык ассемблера или байт-код, был выбран последний.

**2 Организация таблица идентификаторов**

**2.1 Исходные данные**

Требуется написать программу, которая получает на входе набор идентификаторов, организует таблицу по заданному методу и позволяет осуществить многократный поиск идентификатора в этой таблице. Список идентификаторов считать заданным в виде текстового файла.

Выданному варианту задания соответствует построение таблицы идентификаторов по методу цепочек. Для сравнения кроме предложенного метода реализуем построение таблицы идентификаторов по методу бинарного дерева. Для большей эффективности второй метод будет являться комбинированным.

**2.2 Назначение таблиц идентификаторов**

Проверка правильности семантики и генерация кода требуют знания характеристик идентификаторов, используемых в программе на исходном языке. Эти характеристики выясняются из описаний и из того, как идентификаторы используются в программе и накапливаются в таблице символов или таблице идентификаторов. Любая таблица символов состоит из набора полей, количество которых равно числу идентификаторов программы. Каждое поле содержит в себе полную информацию о данном элементе таблицы. Под идентификаторами подразумеваются константы, переменные, имена процедур и функций, формальные и фактические параметры.

В нашей курсовой работе таблица идентификаторов будет применяться для хранения информации об используемых переменных, а также в разделе 5 "Генерация и оптимизация ассемблерного кода" при распределении регистров микропроцессора между переменными.

**2.3 Метод цепочек**

Метод цепочек работает по следующему алгоритму:

Шаг 1. Во все ячейки хеш-таблицы поместить пустое значение, таблица идентификаторов пуста, переменная FreePtr (указатель первой свободной ячейки) указывает на начало таблицы идентификаторов; i:=1.

Шаг 2. Вычислить значение хеш-функции ni для нового элемента Ai. Если ячейка хеш-таблицы по адресу ni пустая, то поместить в нее значение переменной FreePtr и перейти к шагу 5; иначе перейти к шагу 3.

Шаг 3. Положить j:=1, выбрать из хеш-таблицы адрес ячейки таблицы идентификаторов mj и перейти к шагу 4.

Шаг 4. Для ячейки таблицы идентификаторов по адресу mj проверить значение поля ссылки. Если оно пустое, то записать в него адрес из переменной FreePtr и перейти к шагу 5; иначе j:=j+1, выбрать из поля ссылки адрес mj и повторить шаг 4.

Шаг 5. Добавить в таблицу идентификаторов новую ячейку, записать в нее информацию для элемента Ai (поле ссылки должно быть пустым), в переменную FreePtr поместить адрес за концом добавленной ячейки. Если больше нет идентификаторов, которые надо разместить в таблице, то выполнение алгоритма закончено, иначе i:=i+1 и перейти к шагу 2.

В разрабатываемой программе метод цепочек реализует модуль ChainFunc. Для реализации был использован алгоритм, представленный в методических указаниях. Результаты заполнения и поиска в таблице идентификаторов выводятся в главном окне программы.

**3 Проектирование лексического анализатора**

**3.1 Исходные данные**

Для выполнения данной части курсовой работы требуется написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем с указанием их типов и значений. Текст на входном языке задается в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа.

Программа должна допускать наличие комментариев неограниченной длины во входном файле.

**3.2 Принципы работы лексического анализатора**

Лексический анализатор имеет дело с различного рода константами и идентификаторами (к последним относятся и ключевые слова). Язык констант и идентификаторов в большинстве случаев является регулярным - то есть, может быть описан с помощью регулярных (праволинейных или леволинейных) грамматик. Распознавателями для регулярных языков являются конечные автоматы. Существуют правила, с помощью которых для любой регулярной грамматики может быть построен недетерминированный конечный автомат, распознающий цепочки языка, заданного этой грамматикой.

Работа автомата выполняется по тактам. На каждом очередном такте i автомат, находясь в некотором состоянии qi∈**Q**, считывает очередной символ w∈**Σ** из входной цепочки символов и изменяет свое состояние на qi+1=δ(qi,w), после чего указатель в цепочке входных символов передвигается на следующий символ и начинается такт i+1. Так продолжается до тех пор, пока цепочка входных символов не закончится. Конец цепочки символов часто помечают особым символом ⊥. Считается также, что перед тактом 1 автомат находится в начальном состоянии q0.

Графически автомат отображается нагруженным однонаправленным графом, в котором вершины представляют состояния, дуги отображают переходы из одного состояния в другое, а символы нагрузки (пометки) дуг соответствуют функции перехода. Если функция перехода предусматривает переход из состояния q в q’ по нескольким символам, то между ними строится одна дуга, которая помечается всеми символами, по которым происходит переход из q в q’.

**3.3 Схема распознавателя**

Граф конечного детерминированного автомата, используемого для распознавания входных цепочек языка, представлен на рисунке 3.

Начальное и конечное состояния при нормальной работе лексического анализатора совпадают (на рисунке состояние "q"). В случае ошибочной входной цепочки автомат попадает в состояние ошибки ERROR. При этом работа автомата останавливается.

Кроме того, типичными для автомата являются состояния ID (переменная) и CONSTANT (константа). Остальные состояния автомата определяются допустимыми для компилятора исходного языка лексемами.

Каждый переход в конечное состояние "q" сообщает о конце текущей входной цепочки. В этом случае производится анализ распознанной цепочки и перезапуск автомата для очередной входной цепочки символов. Заметим также, что возможна повторная обработка некоторых символов входной цепочки символов. Это необходимо в тех случаях, когда символ, приведший к переходу автомата в конечное состояние, является началом следующей цепочки символов.

**3.3 Результаты**

Программная реализация лексического анализатора представлена в виде модулей LexFSM и LexAnalyser. LexFSM – отражает работу автомата, распознающего входные цепочки символов. LexAnalyser подготавливает текст входного файла для отправки его символов модулю LexFSM по запросам последнего. Он игнорирует незначащие символы входного текста (символы табуляции, возврата каретки и т.д.). Когда LexFSM переходит в одно из конечных состояний, он передает управление модулю LexAnalyser. Тот, в свою очередь, выводит результаты работы на экран и сигнализирует в случае ошибки о строке, в которой произошла ошибка. Кроме того, если LexFSM переходит в конечное состояние "q" и таким образом распознает входную цепочку символов, он передает полученную лексему с информацией о ней модулю LexAnalyser, а тот добавляет ее в свой список лексем. По завершении обработки файла модуль LexAnalyser посылает список лексем графическому модулю LexTab, который отображает результирующий список лексем в виде таблицы.

Также каждый из модулей выдает подробную информацию о своей последовательности действий, благодаря чему удобно следить за ходом работы анализатора. Эти сведения также отображаются на экране с помощью графического модуля LexTab.

Рассмотрим ниже пример обработки текстового файла:

program

begin

begine:=1;

a:=5;

repeat

if a>3 then a:=3

else i:=0

endif

until i=0

end;

end.

Результаты работы лексического анализатора кратко

представлены в таблице 1.

Таблица 1

|  |  |  |
| --- | --- | --- |
| Лексема | Тип лексемы | Номер идентификатора в таблице идентификаторов |
| program | Ключевое слово 'program' |  |
| begin | Ключевое слово 'begin' |  |
| begine | Переменная | 1 |
| := | Оператор присваивания ':=' |  |
| 5 | Константа |  |
| ; | Разделитель |  |
| a | Переменная | 2 |
| := | Оператор присваивания ':=' |  |
| 5 | Константа |  |
| ; | Разделитель |  |
| repeat | Ключевое слово 'repeat' |  |
| if | Ключевое слово 'if' |  |
| a | Переменная | 2 |
| > | Знак операции '>' |  |
| 3 | Константа |  |
| then | Ключевое слово 'then' |  |
| a | Переменная | 2 |
| := | Оператор присваивания ':=' |  |
| 3 | Константа |  |

**4 Построение дерева вывода**

**4.1 Исходные данные**

Для программной реализации построения дерева вывода требуется написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием, порождает таблицу лексем и выполняет синтаксический разбор текста по заданной грамматике с построением дерева разбора. Текст на входном языке задается в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок.

**4.2 Синтаксический анализатор**

Исходная грамматика:

*G* ({ *program*, *end*., *if*, *then*, *else*, *endif*, *begin*, *end*, *repeat*, *until*, *and*, *or*, *not*, <<, >>, =, <, >, <>, (, ), -, +, *a*, *c*, ;, :=}, {*S*, *L*, *O*, *B*, *C*, *K*, *D*, *H*, *E*, T}, P, S)

*S* → *program* *L* *end*.

*L* → *O* | *O* ; *O* | *L* ;

*O* → *if B then O else O endif* | *if B then O endif* | *begin L end* | *repeat O until B* | *a* := *E*

*B* → *B or C* | *C*

*C* → *C and D* | *D*

*D* → *not D* | *H*

*H* → *E* < *E* | *E* > *E* | *E* = *E* | *E* <> *E* | (*B*) | *E* << *E* | *E* >> *E*

*E* → *E* – *T* | *E* + *T* | *T*

*T* → (*E*) | *a* | *c*

Множества крайних левых и крайних правых символов **L**(U), **R**(U) относительно всех нетерминальных символов грамматики представлены в таблице 2. В таблице 3 описаны итоговые множества крайних левых и крайних правых символов **L**(U), **R**(U) относительно всех нетерминальных символов грамматики.

Таблица 2

|  |  |  |
| --- | --- | --- |
| **U** | **L(U)** | **R(U)** |
| *T* | (, *a*, *c* | ), *a*, *c* |
| *E* | *E*, *T* | *T* |
| *H* | (, *E* | *E*, ) |
| *D* | *not*, *H* | *D*, *H* |
| *C* | *C*, *D* | *D* |
| *B* | *B*, *C* | *C* |
| *O* | *if*, *begin*, *repeat*, *a* | *endif*, *E*, *end*, *B* |
| *L* | *L*, *O* | *O*, ; |
| *S* | *program* | *end*. |

Таблица 3

|  |  |  |
| --- | --- | --- |
| **U** | **L(U)** | **R(U)** |
| *T* | (, *a*, *c* | ), *a*, *c* |
| *E* | *E*, *T*, (, *a*, *c* | *T*, ), *a*, *c* |
| *H* | (, *E*, *T*, *a*, *c* | *E*, ), *T*, *a*, *c* |
| *D* | *not*, *H*, (, *E*, *T*, *a*, *c* | *D*, *H*, *E*, ), *T*, *a*, *c* |
| *C* | *not*, *H*, (, *E*, *T*, *a*, *c* | *D*, *H*, *E*, ), *T*, *a*, *c* |
| *B* | *B*, *C* | *C*, *D*, *H*, *E*, ), *T*, *a*, *c* |
| *O* | *if*, *begin*, *repeat*, *a* | *endif*, *E*, *end*, *B* |
| *L* | *L*, *O*, *if*, *begin*, *repeat*, *a* | O, ;, *endif*, *E*, *end*, *B* |
| *S* | *program* | *end.* |

Множества крайних левых и крайних правых терминальных символов **L**t(U), **R**t(U) относительно всех нетерминальных символов грамматики и итоговые множества крайних левых и крайних правых терминальных символов **L**t(U), **R**t(U) относительно всех нетерминальных символов грамматики представлены соответственно в таблицах 4 и 5.

Таблица 4

|  |  |  |
| --- | --- | --- |
| **U** | **Lt(U)** | **Rt(U)** |
| *T* | (, *a*, *c* | ), *a*, *c* |
| *E* | -, + | -, + |
| *H* | <, >, =, <>, (, << | <, >, =, <>, ), >> |
| *D* | *not* | *not* |
| *C* | *and* | *and* |
| *B* | *or* | *or* |
| *O* | *if*, *begin*, *repeat*, *a* | *endif*, *end*, :=, *until* |
| *L* | ; | ; |
| *S* | *program* | *end*. |

Таблица 5

|  |  |  |
| --- | --- | --- |
| **U** | **Lt(U)** | **Rt(U)** |
| T | (, *a*, *c* | ), *a*, *c* |
| E | -, +, (,*a*, *c* | -, +, ), *a*, *c* |
| H | <, >, =, <>, (, <<, -, +, *a*, *c* | <, >, =, <>, ), >>, -, +, *a*, *c* |
| D | *not*, <, >, =, <>, (, <<, -, +, *a*, *c* | *not*, <, >, =, <>, ), >>, -, +, *a*, *c* |
| C | *and*, *not*, <, >, =, <>, (, <<, -, +, *a*, *c* | *and*, *not*, <, >, =, <>, ), >>, -, +, *a*, *c* |
| B | *or*, *and*, *not*, <, >, =, <>, (, <<, -, +, *a*, *c* | or, *and*, *not*, <, >, =, <>, ), >>, -, +, *a*, *c* |
| O | *if*, *begin*, *repeat*, *a* | *endif*, *end*, :=, *until*, *or*, *and*, *not*, <, >, =, <>, ), >>, -, +, *a*, *c* |
| L | ;, *if*, *begin*, *repeat*, *a* | ;, *endif*, *end*, :=, *until*, *or*, *and*, *not*, <, >, =, <>, ), >>, -, +, *a*, *c* |
| S | *program* | *end*. |

Остовная грамматика, полученная на основе исходной грамматики:

*G'* ({ *program*, *end*., *if*, *then*, *else*, *endif*, *begin*, *end*, *repeat*, *until*, *and*, *or*, *not*, <<, >>, =, <, >, <>, (, ), -, +, *a*, *c*, ;, :=}, {*E*}, P, S)

*E* → *program* *E* *end*.

*E* → *E* | *E* ; *E* | *E* ;

*E* → *if E then E else E endif* | *if E then E endif* | *begin E end* | *repeat E until E* | *a* := *E*

*E* → *E or E* | *E*

*E* → *E and E* | *E*

*E* → *not E* | *E*

*E* → *E* < *E* | *E* > *E* | *E* = *E* | *E* <> *E* | (*B*) | *E* << *E* | *E* >> *E*

*E* → *E* – *E* | *E* + *E* | *E*

*E* → (*E*) | *a* | *c*

Для различия скобок, определяющих соответственно приоритет арифметических и логических операций, дополним остовную грамматику дополгительным нетерминальным символом B, который будет обозначать логические выражения.

Преобразованная остовная грамматика примет следующий вид:

*G'* ({ *program*, *end*., *if*, *then*, *else*, *endif*, *begin*, *end*, *repeat*, *until*, *and*, *or*, *not*, <<, >>, =, <, >, <>, (, ), -, +, *a*, *c*, ;, :=}, {*E, B*}, P, S)

*E* → *program* *E* *end*.

*E* → *E* | *E* ; *E* | *E* ;

*E* → *if B then E else E endif* | *if B then E endif* | *begin E end* | *repeat E until B* | *a* := *E*

*B* → *B or B* | *B*

*B* → *B and B* | *B*

*B* → *not B* | *B*

*B* → *E* < *E* | *E* > *E* | *E* = *E* | *E* <> *E* | (*B*) | *E* << *E* | *E* >> *E*

*E* → *E* – *E* | *E* + *E* | *E*

*E* → (*E*) | *a* | *c*

**4.3 Результаты**

Программная реализация синтаксического анализатора представлена в виде модулей SynAnalyser и графического модуля SynTab. Модуль SynAnalyser проводит синтаксический анализ последовательного набора лексем, поступающего от лексического анализатора на основе правил остовной грамматики. Результатом его работы является структура, отражающая дерево синтаксического вывода. В случае ошибки на экране появляется сообщение о синтаксической ошибке и строка с ошибкой выделяется красным цветом. Графический модуль SynTab отвечает за графическое представление дерева вывода, а также выводит подробный отчет о последовательности действий, проводимых синтаксическим анализатором, и их результатах.

Рассмотрим работу синтаксического анализатора при обработке следующего файла:

program

(\* Это

комментарий \* (\* \*

\*)

begin

c:= c - b +d;

repeat

if a>3 then a :=3 else i:=0 endif

until i= 0

end ;

end.

Графически результат построения дерева вывода представлен на рисунке 4.

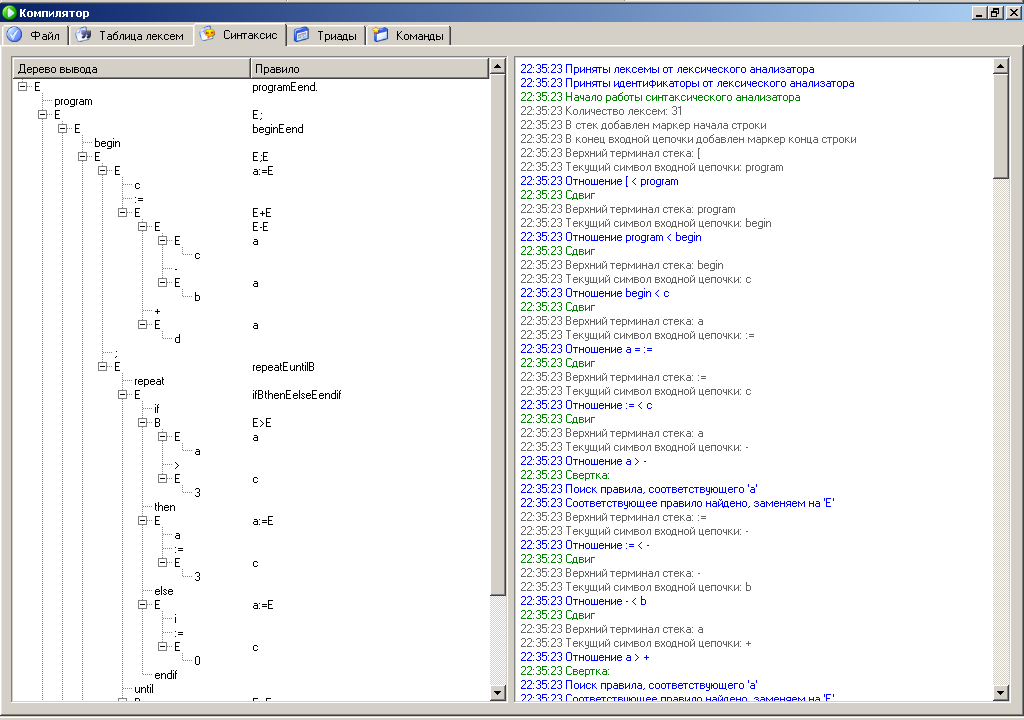


Рисунок 4 – Результат построения дерева вывода

**5 Генерация и оптимизация объектного кода**

**5.1 Исходные данные**

Для выполнения заключительной части курсовой работы требуется написать программу, которая на основании дерева синтаксического разбора порождает объектный код и затем выполняет его оптимизацию. В качестве исходного дерева синтаксического разбора рекомендуется взять дерево, которое порождает программа, построенная по заданию предыдущего раздела работы.

Результатом работы должна быть построенная на основании заданного предложения грамматики программа на объектном языке. В качестве объектного языка предлагается взять язык ассемблера для процессоров типа Intel 80x86 в реальном режиме. Все встречающиеся в исходной программе идентификаторы считать простыми скалярными переменными, не требующими выполнения преобразования типов.

Разработка данной части работы разбивается на два этапа – построение списка триад и генерация ассемблерного кода.

**5.2 Построение списка триад**

При построении списка триад производится рекурсивный проход дерева вывода, построенного синтаксическим анализатором, описанным в разделе 4. При этом вводятся следующие дополнительные типы триад:

– Триада *if* (*a*,*b*), где операнды *a* и *b* обязательно являются ссылками на триады. Смысл данной триады состоит в следующем: если результат вычисления триады *a*, являющейся логическим выражением, равен нулю, то производится переход по ссылке *b*. Иначе производится последовательный переход к следующей по списку триаде.

– Триада jmp (1, a), где первый операнд не несет смысловой нагрузки, а второй указывает ссылку на триаду, к которой на следующем этапе должен быть произведен безусловный переход.

Выражения исходного языка, не несущие семантической нагрузки, не порождают новых триад, но для таких узлов дерева вывода производится рекурсивный вызов функции построения триад.

Остальные выражения однозначно преобразуются в одну или несколько последовательных триад.

По завершении построения списка триад производится его оптимизация в виде исключения лишних операций. При этом используется еще один дополнительный тип триады – SAME (a, 0), где второй операнд не несет смысла, а первый указывает на триаду, которой идентична триада, которая была заменена на данное выражение.

**5.3 Генерация ассемблерного кода**

Генерация ассемблерного кода на основе списка триад не требует дополнительных преобразований, каждая триада может быть однозначно заменена на некоторую последовательность ассемблерных команд. При этом основной проблемой является правильное распределение регистров микропроцессора во время выполнения ассемблерного кода. Решение данного вопроса соответствует методу, предложенному в методических указаниях к курсовой работе [2].

Перед вставкой списка ассемблерных комнд, соответствующих текущей триаде, проверяется необходимость изменения содержимого аккумулятора. Если в нем уже содержится необходимое значение, его изменение является лишним, иначе первой из порожденных команд является команда *mov.* Аналогично, при необходимости сохранения результата выполнения операции (в случае, если он вызывается в следующих далее операциях), он сохраняется в одном из регистров.

**5.4 Результаты**

За генерацию списка триад отвечают модули Triad, TriListMaker и TriListOptimizer. Класс Triad определяет структуру, соответствующую одной триаде. Модуль TriListMaker генерирует первоначальный список триад на основе синтаксического дерева вывода, а модуль TriListOptimizer производит его оптимизацию в виде исключения лишних операций. Как только заканчивается оптимизация триад, результирующий список триад отправляется модулю AsmGenerator - модулю, отвечающему за генерацию ассемблерного кода. Результатом работы модуля AsmGenerator является результирующий объектный код на основе входного текстового файла. Результаты данных операций выводятся на экран при помощи модулей TriTab и AsmTab соответственно.

**6 Описание работы программы**

**6.1 Реализация таблицы идентификаторов**

На рисунке 5 представлено главное окно программной реализации таблицы идентификаторов.

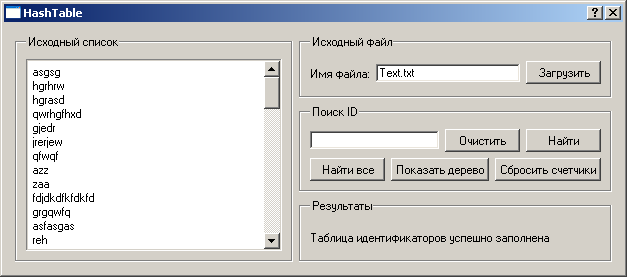


Рисунок 5 – Главное окно

После загрузки файла происходит автоматическое заполнение таблицы идентификаторов. После этого появляется возможность поиска необходимых идентификаторов в этой таблице. При нажатии на кнопку "Найти все" происходит поиск всех идентификаторов из входного файла и выдается статистика, указывающая количество сравнений при поиске текущего идентификатора, а также общее и среднее число сравнений (рисунок 6).

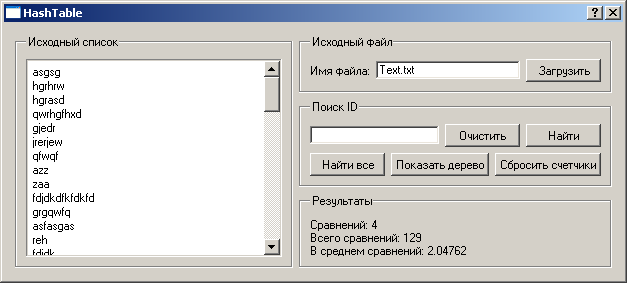


Рисунок 6 – Статистика сравнений при поиске идентификаторов

Для наглядного представления структуры таблицы идентификаторов есть возможность ее просмотра в виде иерархического дерева (рисунок 7).

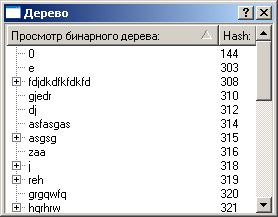


Рисунок 7 – Иллюстрация структуры таблицы идентификаторов

Исходный текст программы приведен в приложении А.

**6.2 Программная реализация лексического анализатора**

На основе графа конечного детерминированного автомата, приведенного в разделе 3, была разработана программная реализация лексического анализатора.

Вкладка загрузки файла (рисунок 8) предоставляет пользователю возможность выбора загружаемого файла, его просмотра, а также просмотра строки с ошибкой в случае лексической ошибки.

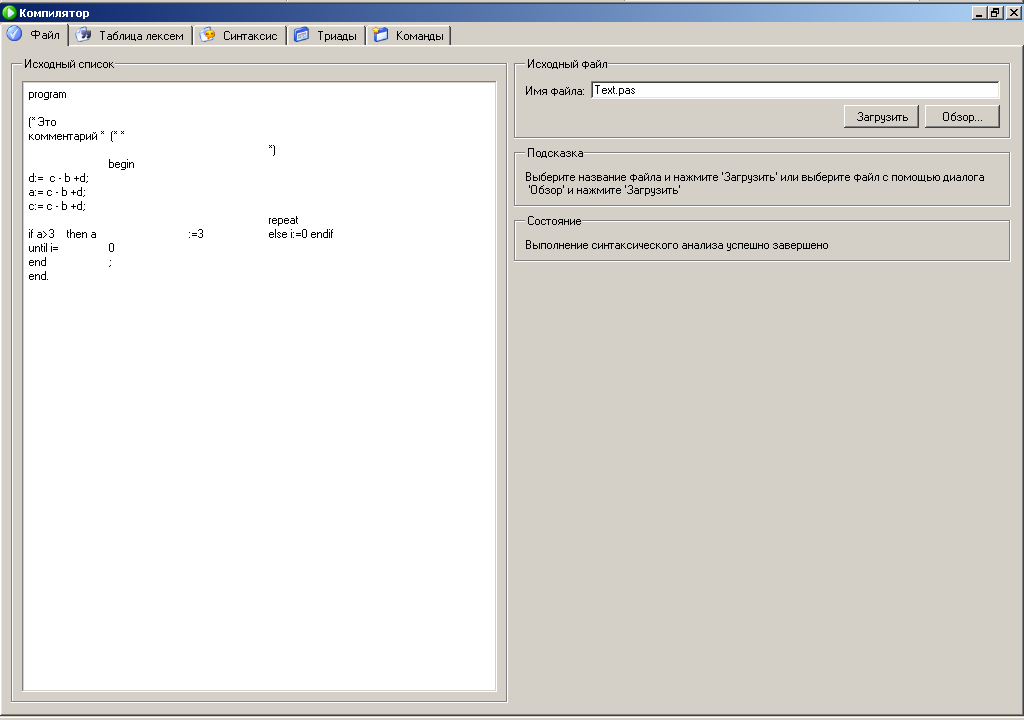


Рисунок 8 – Вкладка загрузки файла

Результаты работы лексического анализатора выводятся во вкладке "Таблица лексем" (Рисунок 9). Там же приводится подробный лог его работы.

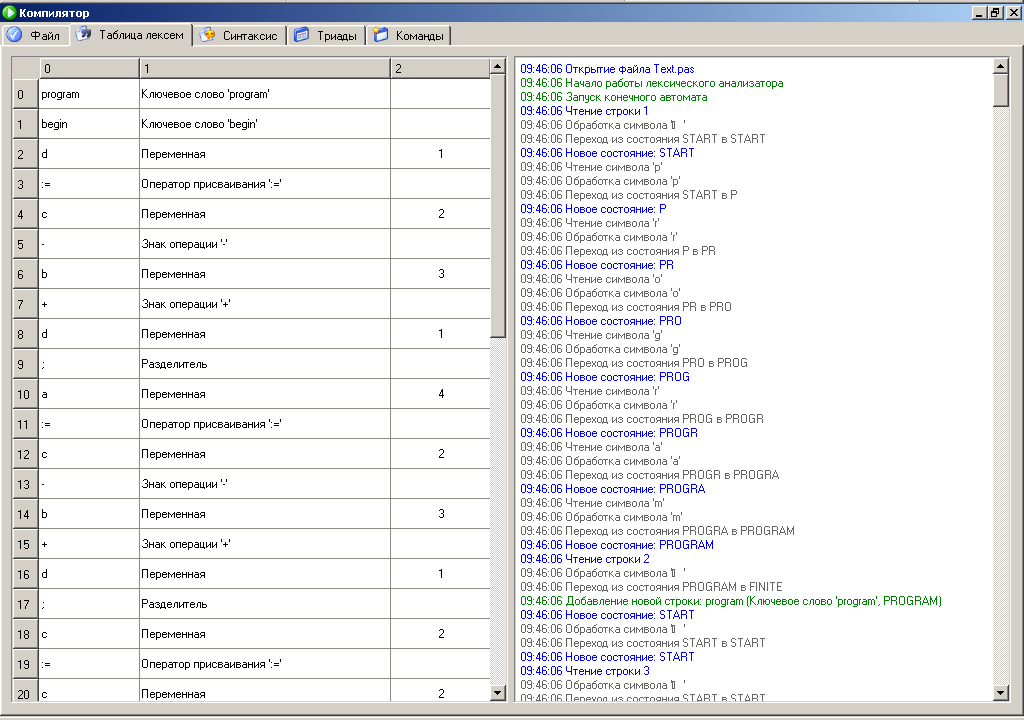


Рисунок 9 – Вкладка "Таблица лексем"

В случае лексической ошибки во входном файле на экран выводится сообщение об ошибке и во вкладке загрузки файла строка с ошибкой подсвечивается красным цветом (Рисунок 10).

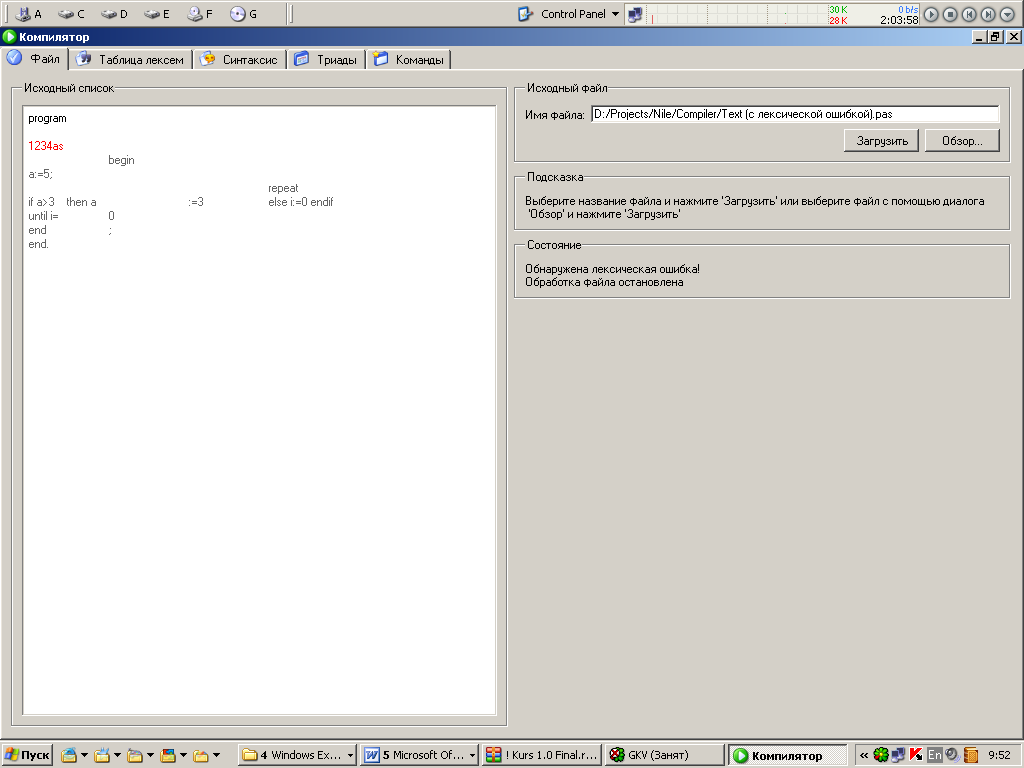


Рисунок 10 – Вывод ошибки

**6.3 Программная реализация синтаксического анализатора**

Реализация синтаксического анализатора аналогична программной реализации лексического анализатора.

Во вкладке "Синтаксис" выводится результат работы анализатора – дерево вывода (Рисунок 11). В этой же вкладке выводится и подробный лог его работы.

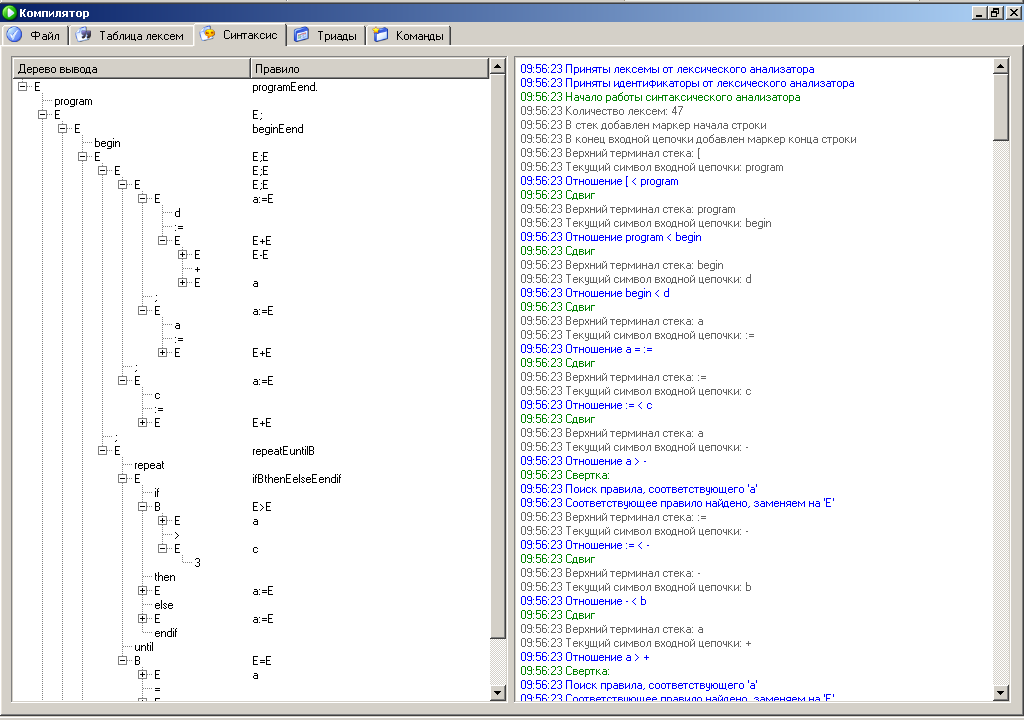


Рисунок 11 – Вкладка "Синтаксис"

Аналогично тому, как производится информирование об обнаруженной лексической ошибке, реализовано сообщение о синтаксической ошибке.

**6.4 Реализация генерации и оптимизации объектного кода**

Во вкладке "Триады" выводится информация о генерации триад и их оптимизации (рисунок 12). Выданному варианту задания соответствует оптимизация в виде исключения лишних операций. Ее результаты и подробный лог работы модуля представлены в этой же вкладке.

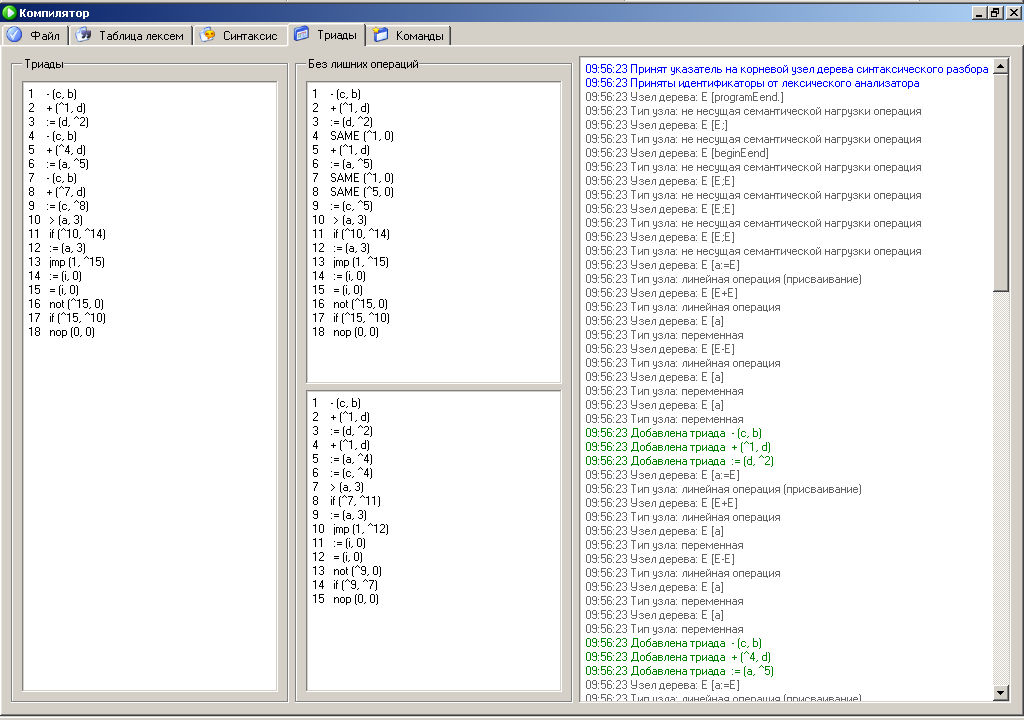


Рисунок 12 – Вкладка "Триады"

Результаты преобразования триад в код ассемблера выделены в отдельную вкладку "Команды" (рисунок 13). Эти результаты и являются результатом работы всей программы.

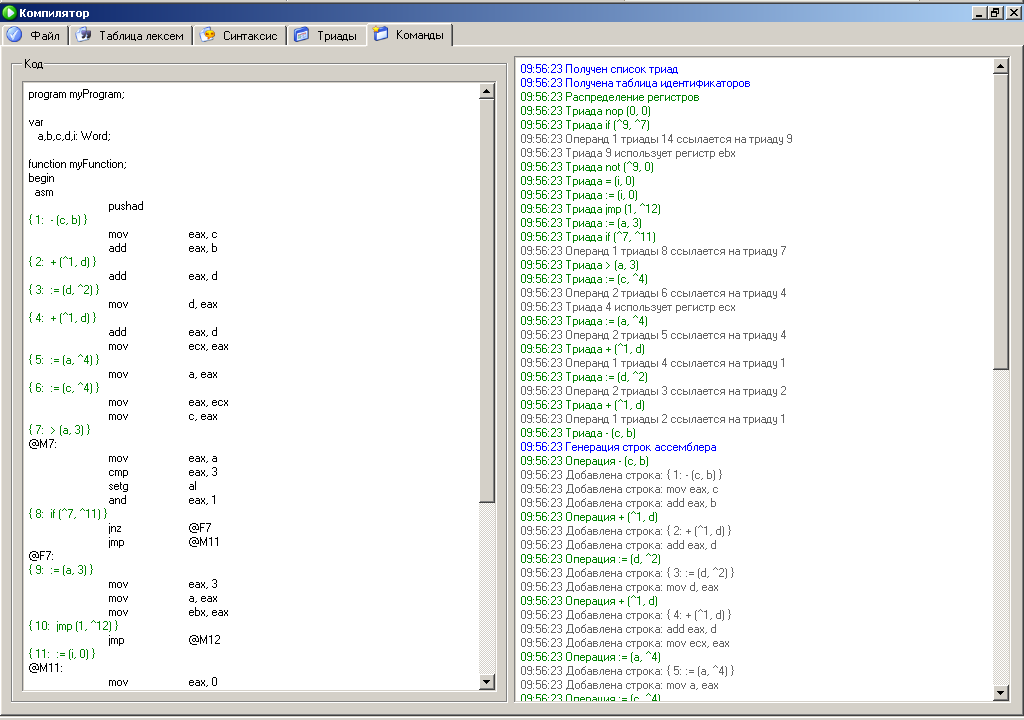


Рисунок 13 – Вкладка "Команды"

Исходные коды для модулей, описанных в разделах 6.2, 6.3 и 6.4, представлены в приложении Б.

**Заключение**

В процессе выполнения курсовой работы была разработана программа, реализующая компилятор заданного подмножества языка Паскаль с незначительными модификациями. Для ее разработки использовалась среда программной разработки Microsoft Visual Studio .NET 2003 с дополнительно интегрированной библиотекой классов Trolltech Qt v4.0.1.

# ПРИЛОЖЕНИЕ А

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Лист регистрации изменений | | | | | | | | | |
| Номера листов (страниц) | | | | | Всего  листов  (страниц)  в докум | №  документа | Входящий  № сопрово  дительного  документа  и дата | Подп. | Дата |
| Изм | изменен  ных | заме  ненных | новых | Анулиро  ванных |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |