Методические указания для выполнения курсовой работы по дисциплине «Системное программное обеспечение»

Тема: Разработка простейшего компилятора программ языка C++
Цель работы: Закрепление знаний, полученных в ходе изучения дисциплины «Системное программное обеспечение» по организации систем программирования, а также, приобретение умений по проектированию и разработке основных компонентов для выполнения первых этапов компиляции программ.

Ход работы:
- ознакомление с теоретическими сведениями об организации трансляторов;
- изучение основных подходов при построении компиляторов на основе подробного рассмотрения примера компилятора для программ языка MATH, описанного в методических указаниях;
- формирование классов для лексического анализатора приложения компилятора согласно варианту задания;
- составление дерева разбора для синтаксического анализатора;
- формирование таблицы переходов на основе дерева разбора;
- разработка блок-схемы алгоритма работы приложения компилятора;
- разработка приложения компилятора программ языка C++;
- проверка работы приложения компилятора при обработке демонстрационной входной программы, организованной согласно варианту;
- оформление пояснительной записки курсовой работы.

Задание:
Выполнить разработку приложения компилятора программ языка C++, реализующего первые два этапа трансляции: лексический анализ, синтаксический анализ. Для приложения сформировать классы лексем, дерево разбора, таблицу переходов и демонстрационную входную программу. В ходе компиляции, приложение должно осуществлять подсчет и вывод числа комментариев, операторов, операций и переменных во входной программе. Особенности входных программ для различных вариантов заданий приведены в приложении А.

Пояснительная записка курсовой работы должна содержать:

1. титульный лист;
2. задание на курсовую работу;
3. содержание пояснительной записки;
4. цель курсовой работы;
5. краткие теоретические сведения про построение трансляторов;
6. описание особенностей организации программы на языке C++, базирующейся на операторах, операциях и конструкциях согласно индивидуальному варианту задания студента;
7. сформированные классы лексем;
8. дерево разбора с описанием его формирования;

9. таблицу переходов, составленную по дереву разбора;
10. блок-схему работы приложения компилятора программ языка C++ с кратким описанием;
11. программный код приложения компилятора с комментариями к добавленным студентом программным листингам;
12. листинг демонстрационной входной программы;
13. отчета по выполнению синтаксического анализа;
14. выводы по проведенной работе;
15. список использованных источников;
16. CD–диск с файлами разработанного проекта*.

*Файлы классов лексем, демонстрационной входной программы, таблицы переходов, исполнимого файла приложения и проекта приложения компилятора программ должны быть приведены в электронном виде на носителе информации (CD - диск).

1. Теоретические сведения

Транслятор

Транслятор — программа, которая принимает на вход программу на одном языке (он в этом случае называется исходный язык, а программа — исходный код), и преобразует её в программу, написанную на другом языке (соответственно, целевой язык и объектный код).
В качестве целевого языка наиболее часто выступают машинный код, Ассемблер и байт-код, так как они более удобны (с точки зрения производительности) для последующего исполнения.
Часто встречаются две разновидности трансляторов: компиляторы и интерпретаторы.
Компиляторы — выдают результат в виде исполняемого файла (в данном случае считаем, что компоновка входит в компиляцию). Этот файл транслируется один раз (может быть запущен самостоятельно) и не требует для работы наличия на персональном компьютере создавшего его транслятора
Интерпретаторы — исполняют программу после разбора (в этом случае в роли объектного кода выступает внутреннее представление программы интерпретатором). Исполняется она построчно. В данном случае программа транслируется (интерпретируется) при каждом запуске (если объектный код кэшируется, возможны варианты) и требует для исполнения наличия на машине интерпретатора и исходного кода.
Помимо трансляторов и интерпретаторов, существует множество промежуточных вариантов. Так, большинство современных интерпретаторов перед исполнением переводят программу в байт-код (так как его покомандно выполнять гораздо проще, а значит, быстрее) или даже прямо в машинный код (в последнем варианте от интерпретатора остался только автоматический запуск, поэтому такой «интерпретатор» называется ЛТ-компилятором).

Компилятор

Большая часть компиляторов переводят программу с некоторого высокоуровневого языка программирования в машинный код, который может быть непосредственно выполнен центральным процессором. Как правило, этот код также должен выполняться в среде конкретной операционной системы, поскольку использует предоставляемые ей возможности (системные вызовы, библиотеки функций). Архитектура (набор программно-аппаратных средств), для которой производится компиляция, называется целевой машиной.

Некоторые компиляторы (например, Java) переводят программу не в машинный код, а в программу на некотором специально созданном низкоуровневом языке. Такой язык — байт-код — также можно считать языком машинных команд, поскольку он подлежит интерпретации виртуальной машиной. Например, для языка Java это JVM (язык виртуальной машины Java), или так называемый байт-код Java (вслед за ним все промежуточные низкоуровневые языки стали называть байт-кодами). Для языков программирования на платформе .NET Framework (C#, Managed C++, Visual Basic .NET и другие) это MSIL (Microsoft Intermediate Language, «Промежуточный язык фирмы Майкрософт»).

Программа на байт-коде подлежит интерпретации виртуальной машиной, либо ещё одной компиляции уже в машинный код непосредственно перед исполнением. Последнее называеется «Just-In-Time компиляция» (JIT), по названию подобного компилятора для Java. MSIL-код компилируется в код целевой машины также JIT-компилятором, а библиотеки .NET Framework компилируются заранее).

Для каждой целевой машины (IBM, Apple и т. д.) и каждой операционной системы или семейства операционных систем, работающих на целевой машине, требуется написание своего компилятора. Существуют также так называемые кросс-компиляторы, позволяющие на одной машине и в среде одной ОС получать код, предназначенный для выполнения на другой целевой машине и/или в среде другой ОС. Кроме того, компиляторы могут быть оптимизированы под разные типы процессоров из одного семейства (путём использования специфичных для этих процессоров инструкций). Например, код, скомпилированный под процессоры семейства i686, может использовать специфичные для этих процессоров наборы инструкций — MMX, SSE, SSE2.

Существуют программы, которые решают обратную задачу — перевод программы с низкоуровневого языка на высокоуровневый. Этот процесс называют декомпиляцией, а программы — декомпиляторами. Но, поскольку компиляция — это процесс с потерями, точно восстановить исходный код, скажем, на C++ в общем случае невозможно. Более эффективно декомпилируются программы в байт-кодах — например, существует довольно надёжный декомпилятор для Flash.

Процесс компиляции состоит из следующих этапов:

1) Лексический анализ - на этом этапе последовательность символов исходного файла преобразуется в последовательность лексем;

2) Синтаксический (грамматический) анализ - последовательность лексем преобразуется в дерево разбора;

3) Семантический анализ - дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д. Результат обычно называется «промежуточным представлением/кодом», и может быть дополненным деревом разбора, новым деревом, абстрактным набором команд или чем-то ещё, удобным для дальнейшей обработки;
4) Оптимизация - выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может быть на разных уровнях и этапах, например над промежуточным кодом или над конечным машинным кодом;
5) Генерация кода - из промежуточного представления выдается код на целевом языке.
В конкретных реализациях компиляторов, эти этапы могут быть разделены или совмещены в том или ином виде.
Важной исторической особенностью компилятора, отраженной в его названии (англ. compile — собирать вместе, составлять), являлось то, что он мог производить и компоновку (то есть содержал две части — транслятор и компоновщик). Это связано с тем, что раздельная компиляция и компоновка как отдельная стадия сборки выделились значительно позже появления компиляторов, и многие популярные компиляторы (например, GCC) до сих пор физически объединены со своими компоновщиками. В связи с этим, вместо термина «компилятор» иногда используют термин «транслятор» как его синоним: либо в старой литературе, либо когда хотят подчеркнуть его способность переводить программу в машинный код (и наоборот, используют термин «компилятор» для подчеркивания способности собирать из многих файлов один).

2. Описание языка программирования MATH

Рассмотрим пример организации первых двух этапов компиляции для несуществующего процедурного языка программирования MATH (он сформирован для обоснования теоретических сведений, приведенных в курсовой работе, на основе симбиоза подходов, предложенных в языке Pascal и его клонах по реализации простейших математических операций). Ниже приведены правила организации программных конструкций на MATH.

Служебные слова языка

Язык MATH характеризуется девятью служебными словами, которые приведены в таблице 1.

Таблица 1 – Описание служебных слов

Служебное слово	Описание
PROGRAM	заголовок новой программы
BEGIN	начало тела программы
END	окончание тела программы
IF	начало тела ветви программы
END_IF	окончание тела ветви программы
TYPE_I	целочисленный тип данных
TYPE_R	вещественный тип данных
SET_V	вывод значения на экран
GET_V	ввод значения

Структура программы MATH

В файле программы на языке MATH допускается написание сразу нескольких листингов. Все они должны начинаться со служебного слова PROGRAM. Отдельная программа может быть как безымянной, так и характеризоваться именем, которое указывается после заголовка. Код, связанный с листингом прописывается в программном блоке тела программы. Ниже приведен формат листинга программы:

PROGRAM <имя программы>
BEGIN
    <тело программы>
END

Тело программы может включать любые допустимые по синтаксису языка команды.

Сложный оператор языка MATH

Язык MATH включает один сложный оператор – IF-END_IF. Этот оператор используется для выделения участков кода в ветви, выполняемые по условию. Формат оператора приведен ниже:

IF [<диапазон значений>]
    <тело оператора>
END_IF

<диапазон значений> представляет собой одно или два неравенства, написанных через разделитель «,», таким образом, есть возможность условного выбора или исключения одного диапазона значений:
$A \in (X1,X2) \rightarrow [A<X2,A>X1]$ или $[A>X1,A<X2]$
$B \notin (X1,X2) \rightarrow [B>X2,B<X1]$ или $[B<X1,B>X2]$
$C \in (-\infty,X1) \rightarrow [C<X1]$
$D \in (X2,\infty) \rightarrow [D>X2]$

Оператор IF-END_IF может иметь любую степень вложенности, то есть позволяет строить ступенчатые конструкции.

Разделители команд и разделители форматирования

Программный код может быть написан как в строку, так и с соблюдением правил форматирования. Для разделения отдельных команд используются операторы IF-END_IF и «;».
Также, как и в большинстве существующих языков, здесь присутствуют разделители форматирования – символы пробела, табуляции и перехода на новую строку.

Объявление переменных

Переменные программы могут обладать как глобальной так и локальной областью видимости в рамках тела листинга. Допустимо объявление переменных (локальных) внутри сложного оператора IF-END_IF.
Для объявления переменных используются два типа данных TYPE_I и TYPE_R, целочисленный и вещественный соответственно.
Допускается выполнять объявление как одной, так и сразу нескольких переменных через разделитель «,» (списка переменных). Формат объявления переменных приведен ниже:

<тип данных> <список переменных>;

Имена переменных согласно лексике языка могут начинаться с буквы или знака нижнего подчеркивания и содержать следующий ряд символов: «a»-«z», «A»-«Z», «0»-«9», «_».

Операции языка

В приведенной ниже таблице 2 перечислены все допустимые в языке простые операторы для выполнения математических операций.

Таблица 2 – Описание математических операторов

Оператор*	**Описание**	**Оператор***	**Описание**
+	сложение	<=	меньше либо равно
-	вычитание	>=	больше либо равно
*	умножение	!=	не равно
/	деление	=	равно
<	меньше	:=	присваивание
>	больше		

*Для упрощения понимания примера унарные математические операции не включены в рассматриваемый язык, таким образом, подразумевается, что отрицательные значения переменных могут получаться только путем выполнения какой-либо операции.

Выражения, реализующие вычисление значений переменных, могут содержать только один оператор присваивания и один оператор математической операции. Формат такого выражения приведен ниже:

<вычисляемая переменная> :=
<переменная или значение><символ операции>< переменная или значение >;

Значения переменных

Как сказано выше, значения, участвующие в программном коде не могут быть отрицательными. Целочисленные записываются в виде последовательности символов «0»-«9». Вещественные, как и целочисленные, включают десятеричный набор цифр и записываются через точку, отделяющую их дробную часть.
Кроме численных значений в MATH можно совместно с оператором вывода на экран использовать константную строку, формат которой приведен ниже:

|<текст строки>|

Операторы ввода/вывода

Для вывода результатов на экран используется оператор SET_V. Он позволяет отображать значение переменной, константное значение, константную строку и результат вычисления математической операции. За одно использование допускается вывод лишь одного значения. Формат оператора приведен ниже:

SET_V <значение>;

Для получения значений, вводимых пользователем, используется оператор GET_V, позволяющий выполнить запись введенного значения в указанную переменную. Формат оператора приведен ниже:

GET_V <переменная>;

Комментарии

В языке MATH присутствуют два вида комментариев: однострочный и многострочный. Однострочный комментарий позволяет исключить из программного листинга текст, расположенный справа от служебного символа «#». Многострочный

комментарий исключает совокупность строк находящихся между служебными символами «#$» и «$#».

Приведем пример программы написанной, согласно перечисленным выше правилам:

#$ *Программа выполняет функцию простейшего калькулятора, реализующего операции умножения, деления, сложения и вычитания* $#

```
# Заголовок программы. Указано имя - CALC
PROGRAM CALC
# Начало тела программы
BEGIN
# Объявление целочисленных переменных A, B, C
        TYPE_I A , B , C ;
# Объявление вещественной переменной Z
        TYPE_R Z ;
# Вывод на экран запроса значения A
        SET_V | Vvedite znachenie A: | ;
# Получение введенного значения в переменную A
        GET_V A ;
# Вывод на экран запроса значения A
        SET_V | Vvedite znachenie B: | ;

# Получение введенного значения в переменную B
        GET_V B ;
# Вывод на экран запроса значения символа операции
        SET_V | Vibirite operaciju (0 – "*"; 1 – "/"; 2 – "+"; 3 – "-"): | ;
# Получение введенного значения в переменную C
        GET_V C ;
# Проверка на соответствие значения переменной C нулю
        IF [ C = 0 ]
# Вычисление произведения чисел A и B
                Z := A * B ;
# Окончание тела оператора  IF-END_IF
        END_IF
# Проверка на соответствие значения переменной C одному
        IF [ C = 1 ]
# Проверка переменной B на значение не равное нулю
                IF [ B != 0 ]
# Вычисление частного чисел A и B
                        Z := A / B ;
# Окончание тела оператора  IF-END_IF
                END_IF
# Проверка переменной B на значение равное нулю
                IF [ B = 0 ]
```

```
# Вывод на экран сообщения об ошибке при делении на 0
                    SET_V | Oshibka delenija na 0! | ;
# Окончание тела оператора  IF-END_IF
              END_IF
# Окончание тела оператора  IF-END_IF
        END_IF
# Проверка на соответствие значения переменной C двум
        IF [ C = 2 ]
# Вычисление суммы чисел A и B
              Z := A + B ;
# Окончание тела оператора  IF-END_IF
        END_IF
# Проверка на соответствие значения переменной C трем
        IF [ C = 3 ]
# Вычисление разности чисел A и B
              Z := A - B ;
# Окончание тела оператора  IF-END_IF
        END_IF
# Проверка не соответствия значения переменной C диапазону [0,3]
        IF [ C < 0 , C > 3 ]
# Вывод на экран сообщения об ошибке неверного ввода номера операции
              SET_V | Vi vveli nedopustimij nomer operacii! | ;

# Окончание тела оператора  IF-END_IF
        END_IF
# Вывод на экран строки
        SET_V | Rezultat: | ;
# Вывод на экран результата вычислений
        SET_V Z ;
# Окончание тела программы
END
```

Программа, приведенная выше, написана с использованием разделения всех служебных слов, символов и значений посредством символа «пробел». Это не совсем обычно, однако позволяет упростить распознавание слов на этапе лексического анализа. **При выполнении курсовой работы студент также может использовать данный подход!**

Теперь, когда ознакомление с языком MATH завершено, перейдем к рассмотрению написания программы, реализующей первые этапы компиляции для приведенного языка.

3. Лексический анализ

Как уже было сказано выше, при лексическом анализе из символов формируются лексемы исходного языка. Предопределенные лексемы сохраняются в виде списка (так называемые классы и их элементы). В ходе анализа формируются дополнительные списки имен переменных, функций и классов, а также всех константных значений кода. Кроме того, на данном этапе отбрасываются все комментарии.

Сформируем классы для языка MATH:

class 1:
1. PROGRAM
2. BEGIN
3. END
4. IF
5. END_IF
6. SET_V
7. GET_V
8. TYPE_I
9. TYPE_R
class 2:
1. +
2. -
3. *
4. /
5. <
6. >
7. <=
8. >=
9. !=
10. =
11. :=
class 3:
1. [
2.]
3. ,
4. ;
class 4:
1. |

Каждый класс включает группу служебных слов и символов, объединенных общей идеей: служебные слова, операции, разделители, специальные символы, ограничивающие константные значения.

На первом этапе трансляции компилятор формирует из программного кода последовательность числовых комбинаций (лексическую последовательность), характеризующих каждое служебное слово, оператор, имя или константу как номер класса / номер в классе (номер элемента). Например:

PROGRAM BEGIN END → 1-1, 1-2, 1-3

Также этот этап позволяет обнаружить ошибки написания служебных слов, имен данных и функций.
Такие же комбинации хранятся в таблице, описывающей правила синтаксического разбора, что позволяет осуществить легкий переход на следующий этап.

4. Синтаксический (грамматический) анализ

При синтаксическом анализе осуществляется проверка кода на соответствие правилам языка программирования. Для этого служат дерево разбора (LL1 грамматика) и составленная по нему таблица переходов.
Дерево разбора представляет собой алгоритм, согласно которому выполняется сопоставление формирования конструкций языка программирования. Оно составляется на основе понимания всех допускаемых правил написания кода и характеризует всю совокупность последовательностей команд, которые могут быть организованы в рамках языка.

Пропишем дерево разбора для языка MATH и пронумеруем состояния для дальнейшей организации таблицы переходов:

1 2 3 4 5 6 7
<Программа> → PROGRAM <имя программы> BEGIN <команды> END <Программа>

8 10
<имя программы> → name

9 11
<имя программы> → ε

12 15 16 17 18
<команды> → <тип переменной> <список переменных> ; <команды>
13 19 20
<команды> → <оператор> <команды>
14 21
<команды> → ε

22 24
<тип переменной> → TYPE_I
23 25
<тип переменной> → TYPE_R

26 27 28
<список переменных> → name <список>

29 31 32
<список> → , <список переменных>
30 33
<список> → ε

34 38 39 40 41 42
<оператор> → name := <A> ; <команды>
35 43 44 45 46
<оператор> → SET_V <значение> ; <команды>
36 47 48 49 50
<оператор> → GET_V name ; <команды>
37 51 52 53 54 55 56 57 58
<оператор> → IF [<неравенство> <граница диапазона>] <команды>END_IF <команды>

59 61 62
<A> → name <операция>
60 63 64
<A> → literal <операция>

65 67 68
<операция> → <символ> <B>
66 69
<операция> → ε

70 72
<символ> → <арифметический оператор>
71 73
<символ> → <оператор отношения>

74 78
<арифметический оператор> → +
75 79
<арифметический оператор> → -
76 80
<арифметический оператор> → *
77 81
<арифметический оператор> → /

82 88
<оператор отношения> → >
83 89
<оператор отношения> → <
84 90
<оператор отношения> → >=
85 91
<оператор отношения> → <=
86 92
<оператор отношения> → !=
87 93
<оператор отношения> → =

94 96
<B> → name
95 97
<B> → literal

98 100
<значение> → <A>
99 101 102 103
<значение> → | literal |

104 105 106 107
<неравенство> → name <оператор отношения> <B>

108 110 111
<граница диапазона> → , <неравенство>
109 112
<граница диапазона> → ε

При построении дерева разбора все группируемые элементы программы заменяются смысловыми обозначениями. Для указания наличия имени переменной в коде используют слово name, а константного значения – literal (иногда, в целях упрощения, для имен функций, классов, структур могут также быть задействованы специальные слова). На эти специальные обозначения в ходе лексического анализа должны формироваться дополнительные классы.
В дереве разбора формируются своеобразные циклы (например, для <команды>). Для выхода из них используются дополнительные переходы на символ «ε». Такой переход ставится в конце циклического перебора и характеризует все возможные комбинации символов, не учтенные ранее.
Нумерация элементов дерева разбора используется для организации связей переходов по состояниям в таблице. Принято нумеровать вначале точки входа в каждую группу, а, затем, подряд построчно элементы групп.

По сформированному дереву разбора заполняется таблица переходов. По ней приложение компилятор будет осуществлять проверку правил организации листинга входной программы. Таблица переходов имеет следующий формат:

Состояние	**Терминал**	**Переход**	**Принять**	**В Стек**	**Из Стека**	**Ошибка**	**Значение ошибки**

Состояние – порядковый номер состояния согласно дереву разбора;
Терминал – список не групповых значений (терминалов), которые могут встретиться в данном состоянии;
Переход – номер состояния, на которое нужно осуществить переход с текущего;
Принять – логический флаг, определяющий нужно или нет сделать шаг вдоль строки к следующему состоянию;

В Стек – номер состояния заносимого в стек, на которое необходимо будет вернуться после окончания спуска по дереву;
Из Стека – логический флаг, определяющий нужно или нет вернуться на состояние, которое было отмечено в стеке;
Ошибка – логический флаг, определяющий будет ли считаться ошибкой не совпадение текущего элемента лексической последовательности с терминалами состояния;
Значение ошибки – комментарий к ошибке.

При формировании таблицы можно пользоваться следующими закономерностями для флагов:

Закономерности	Переход	Прин.	В С.	Из С.	Ош.
перебор столбца	след. номер в строке	-	-	-	-
конец перебора столбца (не Е)	след. номер в строке	-	-	-	+
конец перебора столбца (Е)	0	-	-	+	+
терминал в середине строки	след. номер в строке	+	-	-	+
терминал в конце строки	0	+	-	+	+
спуск на уровень ниже	номер перв. эл. столбца	-	след. номер в строке	-	+

Пример заполнения таблицы переходов для языка MATH по составленной грамматике приведен в приложении Б.
После составления таблицы переходов необходимо осуществить замену всех указанных терминалов (кроме специальных literal, name и т.д.) на соответствующие комбинации номеров элементов класса. После этого таблица будет готова для использования компилятором.

5. Разработка приложения, реализующего функции компилятора

Для облегчения задачи программирования приложения компилятора в приложении В приведен ряд упрощенных функций для обработки программного листинга и лексической последовательности. Приведенная ниже таблица дает обзор предложенных функций:

Функция	Назначение
GetFileText()	Чтение текста из файла
FormClassesArr()	Формирование динамического массива основных классов
Lexical()	Выполнение лексического анализа
DelComments()	Удаление и подсчет комментариев
CheckWord()	Анализ текущего слова (оператора)
CheckAResult()	Проверка результата анализа слова
GetSyntTableString()	Получение значений полей строки таблицы переходов
Syntactical()	Выполнение синтаксического разбора
IsInString()	Проверка наличия номера в строке терминалов
FormReport()	Формирование файла отчета

Реализации приведенных функций тесно взаимосвязаны. Работа с классами осуществляется посредством использования следующей структуры:

```
struct Classes        //структура для хранения информации про классы
{
        CString num;        //строка для номера класса/в классе
        CString word;       //строка для служебного слова или оператора
}*pCls,*pNCls;              //переменные-указатели - экземпляры структуры
```

Пример реализации компилятора на базе предложенных функций:

Создадим в Visual C++ приложение с интерфейсом, приведенным на рисунке ниже:

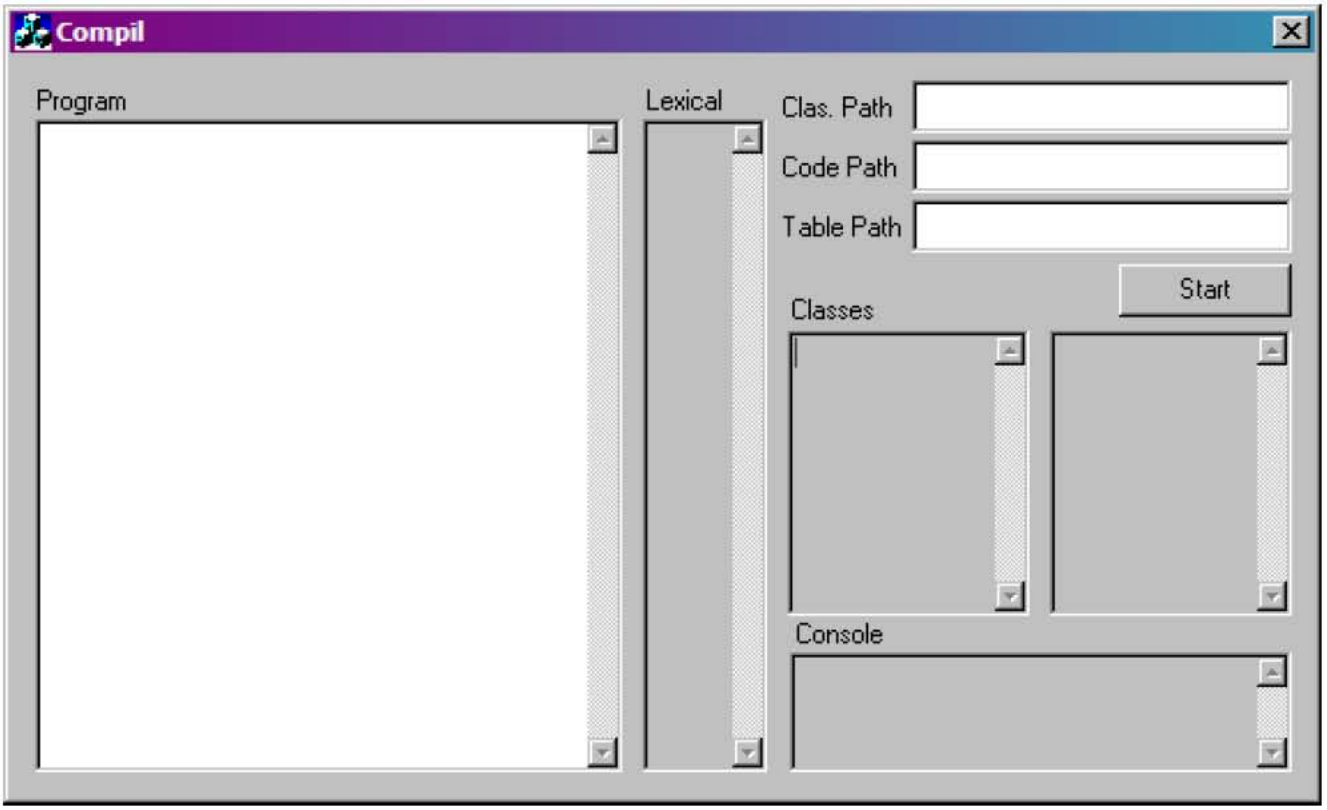


Таблица идентификаторов и имен переменных

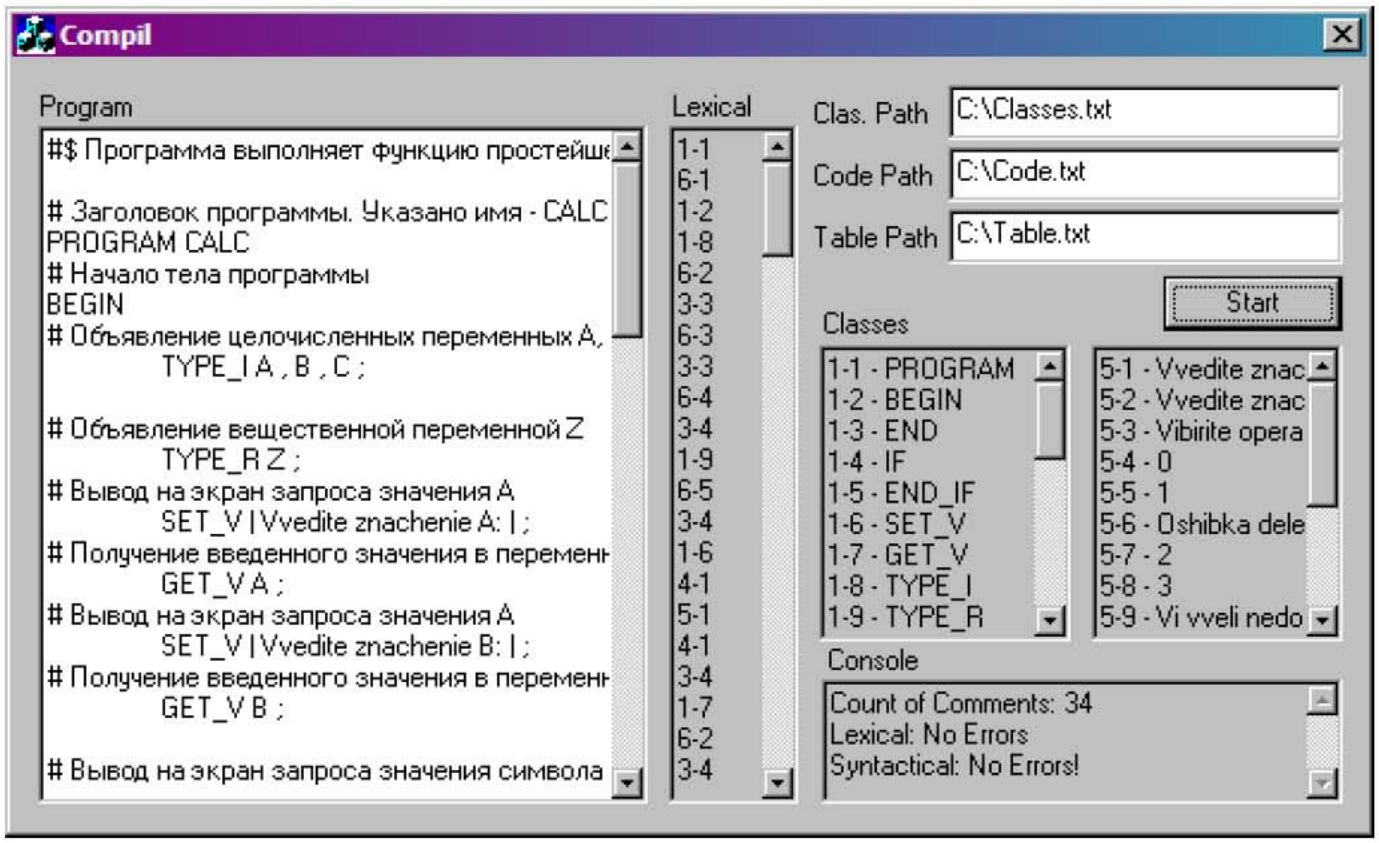
Поле /элемент	Идентификатор	Имя переем.	Тип перем.	Назначение
Program	IDC_E_PROGRAM	m_program	CEdit	Вывод программы
Lexical	IDC_E_LEX	m_ls	CEdit	Вывод лексич. последовательности
Clas. Path	IDC_E_CLSPARTH	m_pcls	CEdit	Ввод пути к файлу классов
Code Path	IDC_E_CDPATH	m_pcode	CEdit	Ввод пути к файлу программы
Table Path	IDC_E_TABPATH	m_ptabl	CEdit	Ввод пути к файлу таблицы перех.
Classes 1	IDC_E_CLS1	m_classes	CEdit	Вывод основных классов
Classes 2	IDC_E_CLS2	m_nclasses	CEdit	Вывод дополнительных классов
Console	IDC_E_CONSOLE	m_console	CEdit	Вывод результата работы
Start	IDC_B_START			Кнопка запуска компиляции

Добавим в основной класс приложения объявление структуры Classes и приведенные в приложении В функции. На базе функций сформируем листинг упрощенной компиляции входной программы и поместим его в тело функции обработки нажатия кнопки «Start»:

```
	CString Cls,Table,Code;		//переменные для хранения текста из входных файлов
	CString path;			//путь к файлу
	CString Result;				//Строка для вывода в окно "Console"

	bool fc=0,fp=0,ft=0;		//флаги контроля открытия входных файлов

	m_pcls.GetWindowText(path);		//получение пути к файлу классов
	if(GetFileText(path,Cls))	//если удалось получить текст из файла
		fc=1;				//установить флаг в 1
	m_pcode.GetWindowText(path);//получение пути к файлу с кодом программы
	if(GetFileText(path,Code))	//если удалось получить текст из файла
		fp=1;				//установить флаг в 1
	m_ptabl.GetWindowText(path);//получение пути к файлу с таблицей переходов
	if(GetFileText(path,Table))	//если удалось получить текст из файла
		ft=1;				//установить флаг в 1

	if(fc&fp&ft!=0)				//если все флаги в 1
	{
		m_program.SetWindowText(Code); //вывод кода программы в окно "Program"

//ФОРМИРОВАНИЕ И ОТОБРАЖЕНИЕ ОСНОВНЫХ КЛАССОВ
		int res;			//переменная для хранения размерности массива классов
		FormClassesArr(Cls, &pCls, res);//создание массива основных классов
		CString sCls;  //строка для формирования визуализации основных классов
		CString vs;		//строка для формирования записей основных классов
		for(int i=0;i<res;i++)  //циклл перебора элементов массива классов
		{
			vs=(pCls+i)->num+" - "+(pCls+i)->word;//формирование записи
			sCls+=vs+"\r\n";	//добавление записи в sCls
		}
		m_classes.SetWindowText(sCls);	//вывод классов в 1е окно "Classes"

//УДАЛЕНИЕ И ПОДСЧЕТ КОММЕНТАРИЕВ
		int CntCom;			//счетчик комментариев
		Code=DelComments(Code,CntCom,"#$","$#","#");//подсчет и удаление комментариев
//формирование отчета по комментариям
		Result.Format("Count of Comments: %i\r\n",CntCom);

//ЛЕКСИЧЕСКИЙ АНАЛИЗ КОДА
//переменная для хранения размерности массива классов
		int Nres;
		CString LexString;		//строка для хранения лексической последовательности
		CString LS;	//строка для вывода лексической последовательности в окно "Lexical"
```

```
//выполнение лексического анализа, формирования классов значений и имен переменных
//и получение лексической последовательности
		LexString=Lexical(Code, pCls, res, &pNCls, Nres);
		LS=LexString;		//пересохранение лексической последовательности
		if(LexString.GetAt(0)!='E')//проверка на отсутствие ошибок в ходе анализа
		{
//ФОРМИРОВАНИЕ И ОТОБРАЖЕНИЕ ДОПОЛОЛНИТЕЛЬНЫХ КЛАССОВ
//замена всех запятых на переходы на новую строку
			LS.Replace(",","\r\n");
//вывод лексической последовательности в окно "Lexical"
			m_ls.SetWindowText(LS);
			sCls.Empty();				//очистка строки
			for(int i=0;i<Nres;i++)		//циклл перебора элементов массива классов
			{
				vs=(pNCls+i)->num+" - "+(pNCls+i)->word;//формирование записи
				sCls+=vs+"\r\n";		//добавление записи в sCls
			}
			m_nclasses.SetWindowText(sCls);	//вывод классов во 2е окно "Classes"

			Result+="Lexical: No Errors\r\n";	//добавление в строку текста
			Result+="Syntactical: ";			//добавление в строку текста
//СИНТАКСИЧЕСКИЙ РАЗБОР ЛЕКСИЧЕСКОЙ ПОСЛЕДОВАТЕЛЬНОСТИ
			int lcls;				//номер последнего класса (специальные символы)
//получение номера последнего класса
			lcls=atoi((pCls+res-1)->num.Left((pCls+res-1)->num.Find("-")));
//добавление в строку результата синтаксического разбора
			Result+=Syntactical(Table, LexString,lcls);
			m_console.SetWindowText(Result);//вывод результата в окно "Console"
		}
		else		//если в ходе анализа произошла ошибка
		{
			Result+="Lexical: ";  //добавление в строку текста
			Result+=LexString;		//добавление в строку лексической ошибки
			m_console.SetWindowText(Result);//вывод результата в окно "Console"
		}
		delete [] pCls;		//удаление массива основных классов
		delete [] pNCls;	//удаление массива дополнительных классов
	}
	else			//если хотя бы один флаг в 0
		AfxMessageBox("Can't open some files!"); //вывод сообщения
```

Воспользуемся разработанным приложением для компиляции приведенной выше демонстрационной программы языка MATH. Для этого создадим на основе описанных в теоретической части списка основных классов, программного листинга и таблицы переходов файлы «Classes.txt», «Code.txt», «Table.txt». Результат работы программы представлен на рисунке ниже:



Ход синтаксического анализа можно проследить при помощи созданного в ходе работы файла «Report.txt», который приведен в приложении Г.

Рекомендации к выполнению: перед выполнением курсовой работы студенту настоятельно рекомендуется внимательно ознакомиться с методическими указаниями и приведенным примером реализации приложения компилятора языка MATH. Для адаптации предложенного ряда функций к индивидуальному заданию необходимо дополнить их кодами защиты от ошибок пользователя и кодами подсчета числа операторов, операций и переменных.

Перечень литературы

1. Гордеев А.В., Молчанов А.Ю. «Системное программное обеспечение», СПб.: Питер, 2002, - 736 с.
2. Ахо А.В., Лам М.С., Сети Р., Ульман Д.Д., «Компиляторы: Принципы, технологии, инструменты», 2е изд., ООО «И.Д. Вильямс», 2008, - 1184 с.

Приложение А

Задания согласно вариантам по списку

Таблица – Операции языка С++

Операторы	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
ун. -	+					+	+				+		+			+	+				+		+		
+	+					+	+				+		+			+	+				+		+		
-	+					+	+				+		+			+	+				+		+		
*	+					+	+				+		+			+	+				+		+		
/	+					+	+				+		+			+	+				+		+		
%	+					+	+				+		+			+	+				+		+		
++	+					+	+				+		+			+	+				+		+		
--	+					+	+				+		+			+	+				+		+		
<	+	+						+				+		+				+				+		+	
>	+	+						+				+		+				+				+		+	
<=	+	+						+				+		+				+				+		+	
>=	+	+						+				+		+				+				+		+	
= =	+	+						+				+		+				+				+		+	
!=	+	+						+				+		+				+				+		+	
<<		+	+				+		+						+		+		+						+
>>		+	+				+		+						+		+		+						+
&		+	+				+		+						+		+		+						+
|		+	+				+		+						+		+		+						+
^		+	+				+		+						+		+		+						+
~		+	+				+		+						+		+		+						+
&&			+	+				+		+			+			+		+		+			+		
||			+	+				+		+			+			+		+		+			+		
!			+	+				+		+			+			+		+		+			+		
=	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
+=				+	+				+		+			+					+		+			+	
-=				+	+				+		+			+					+		+			+	
*=				+	+				+		+			+					+		+			+	
/=				+	+				+		+			+					+		+			+	
%=				+	+				+		+			+					+		+			+	
&=					+	+				+		+			+					+		+			+
|=					+	+				+		+			+					+		+			+
^=					+	+				+		+			+					+		+			+
<<=					+	+				+		+			+					+		+			+
>>=					+	+				+		+			+					+		+			+
?:	+	+						+				+		+				+				+		+	

Таблица – Сложные операторы языка С++

Операторы	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
if-else	+							+						+										+	
switch			+	+					+				+		+		+		+		+		+		+
while		+				+				+								+							
do-while												+										+			
for					+		+				+					+				+					
break			+	+					+				+		+		+		+		+		+	+	+
continue		+				+		+		+		+						+				+			
goto	+				+		+				+			+		+				+					

Таблица – Конструкции и механизмы языка С++

Конструкция	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
функции	+							+							+										
массивы		+							+							+									
динамические массивы			+							+							+								
структуры				+							+							+				+			
классы					+							+							+					+		
наследование						+							+							+					+	
указатели							+							+							+					+