Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Интерпретатор C-подобного языка программирования»**

**Выполнили**:

студенты группы 3823Б1ПМ1-1

Загрядсков М.А., Болтенков С.С.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2025

**Содержание**

[Постановка задачи 3](#_Toc152527359)

[Метод решения 4](#_Toc152527360)

[Руководство пользователя 7](#_Toc152527361)

[Описание программной реализации 8](#_Toc152527362)

[Подтверждение корректности 9](#_Toc152527363)

[Результаты экспериментов 10](#_Toc152527364)

[Заключение 16](#_Toc152527365)

[Приложение 17](#_Toc152527366)

[Источники 18](#_Toc152527367)

# Постановка задачи

Требовалось:

1. Спроектировать и реализовать программу, способную проинтерпретировать (выполнить по отдельным командам) программу на C-подобном языке. Должно поддерживаться максимально возможное количество ключевых слов и операций из языка C98.
2. Протестировать программу на нескольких примерах с использованием всех поддерживаемых ключевых слов и операций. Убедиться в корректности обработки ошибок, которые могут быть в пользовательской программе.

# Руководство пользователя

Взаимодействие с исходной и пользовательской программами осуществляется через консоль. Пользовательская программа должна быть описана в файле code.cpp в папке проекта code. Вывод ошибок, а также ввод/вывод пользовательской программы осуществляется также через консоль.

Поддерживаемые команды:

1. Операции с переменными:
   1. Логические операции: ==, <=, >=, <, >, !=, &&, ||. Любая переменная приводима к логическому типу
   2. Арифметические операции: =, +, +=, -, -=, \*, \*=, /, /=, % (для целочисленных типов данных), %= (для целочисленных типов данных).
2. Типы данных: void (для возвращаемых значений функций), int, string, double.
3. Условные операторы: if, elif, else.
4. Операторы циклов: for, while. В for поддерживается объявление итерационной переменной в условии for.
5. Стандартные предопределенные функции:
   1. print(var1, var2, …, varn) – выводит в консоль n переменных через пробел, после выводит специальный символ новой строки.
   2. scan(var1, var2, …, varn) – запрашивает из консоли n переменных через клавишу новой строки (enter).
   3. Математические функции: abs(var), sin(var), cos(var), sqrt(var).
6. Прочие ключевые слова и специальные символы:
   1. return – используется для возврата значений из функций.
   2. Символ «;» используется для конца команды, «,» для объявления аргументов функции, «{}» используются для выделения блоков тел функций, условных операторов и операторов цикла, «()» используются для изменения приоритета операций, для условий операторов, а также для вызова и объявления функций.
7. Отступы, количество пробелов и разбиение на строки не влияет на исполнение программы, кроме случаев, когда различить имя ключевого слова и пользовательской переменной невозможно.

# Описание программной реализации

В папке проекта доступны следующие файлы:

1. Этот файл отчета «Отчет\_лабораторная\_Интерпретатор\_Загрядсков\_Максим\_Болтентов\_Станислав\_3823Б1ПМ1\_1.docx».
2. Файл code/code.cpp пользовательского кода.
3. include/calculator.h – заголовочный файл описания класса calculator для вычислений арифметических выражений.
4. include/interpretator.h – заголовочный файл описания класса interpretator для предобработки пользовательского кода, обработки условных операторов.
5. include/lexems/variable.h – заголовочный файл описания класса variable для переменных программы.
6. include/lexems/operation.h – заголовочный файл описания класса operation арифметических и логических операций над переменными.
7. include/lexems/specialLexem.h – заголовочный файл описания класса specialLexem специальных символов, таких как скобки приоритета, символа «;» и так далее.
8. include/lexems/myoperators.h – заголовочный файл описания класса myoperators ключевых слов, в том числе условных операторов и операторов цикла.
9. include/lexems/operand.h – заголовочный файл описания абстрактного класса операндов operand, общий класс для констант и переменных.
10. include/lexems/function.h – заголовочный файл описания класса function для функций программы.
11. include/lexems/keyWords.h – заголовочный файл описания абстрактного класса keyWords ключевых слов, поддерживаемых интерпретаторов, таких как типы данных и условные операторы.
12. include/lexems/dataType.h – заголовочный файл описания класса dataType типов данных, используемых в программе.
13. include/lexems/constant.h – заголовочный файл описания класса констант constant, использующегося для литералов и неименованных констант возврата из арифметических выражений и функций.
14. include/lexems/commonLexem.h – заголовочный файл описания абстрактного класса commonLexem, общего для всех лексем программы.
15. src/main.cpp – файл функции main, где происходит инициализация класса interpretator.
16. src/\*.cpp, src/lexems/\*.cpp – соответствующие вышеописанным файлам реализации соответствующих классов.
17. sln – папка проекта Visual Studio 2022.
18. README.md – файл-аннотация для репозитория.

Каждому командному слову (лексеме) в коде пользовательской программы соответствует свой класс. Система наследования этих классов реализована следующим образом:

Рис. 1. Диаграмма наследования классов лексем программы.

Краткое описание методов, которые содержит каждый из классов:

* commonLexem:
  + getName/setName – возвращает/устанавливает имя лексемы.
  + getInd/setInd – возвращает/устанавливает номер строки, в которой находится лексема.
  + getPos/setPos - возвращает/устанавливает позицию в строке, в которой находится лексема.
  + Виртуальный метод showInfo – выводит в консоль всю информацию о данной лексеме. Используется для отладки.
  + Виртуальный метод getClass – возвращает в виде строки название настоящего класса лексемы, то есть того класса, которого была создана переменная изначально.
* specialLexem: определены виртуальные методы базового класса.
* function:
  + Определены виртуальные методы базового класса.
  + Объявлен и определен компаратор для хранения данных этого типа в контейнере std::set.
* keyWords: определены виртуальные методы базового класса.
* operand:
  + isValidCharForOperand – возвращает true/false в зависимости от того, является ли символ корректным для имени операнда.
  + getTypeId/setTypeId – возвращает/устанавливает значение идентификатора типа данных.
  + getValue/setValue – возвращает/устанавливает значение операнда.
  + isTrue – возвращает, является ли операнд после приведения к логическому типу истинным или ложным.
* operation:
  + Определены виртуальные методы базового класса.
  + getPriority/setPriority – возвращает/устанавливает значение приоритета операции.
* myoperators:
  + Определены виртуальные методы базового класса.
  + getBegin/getEnd – возвращает начало/конец блока данного оператора (тела цикла, тела условного оператора) в формате индекса массива лексем, в котором хранится пользовательская программа после обработки.
* dataType:
  + Определены виртуальные методы базового класса.
  + getTypeId – возвращает значение идентификатора типа данных.
* constant:
  + Определены виртуальные методы базового класса.
  + Определен копирующий оператор присваивания.
  + isInteger(std::string) – возвращает true/false в зависимости от того, является ли входящая на вход строка целочисленной константой или нет.
  + isValidConstant(std::string) – возвращает true/false в зависимости от того, является ли входящая на вход строка корректным литералом.
* variable:
  + Определены виртуальные методы базового класса.
  + Объявлен и определен компаратор для хранения данных этого типа в контейнере std::set.
  + isValidVariable(std::string) – возвращает true/false в зависимости от того, является ли входящая на вход строка корректной переменной.
  + isValidCharForVariable(char) – возвращает true/false в зависимости от того, является ли символ корректным для имени переменной.

Описание хода выполнения программы:

1. Инициализируется класс interpretator. На его вход подается вектор строк, содержащий код программы в строковом формате.
2. Класс interpretator выполняет парсинг исходного кода пользовательской программы, сохраняя функции в отдельную таблицу функций для удобного вызова функций. Все лексемы дифференцируются по их классам.

|  |  |
| --- | --- |
| Исходное выражение: | Преобразованное выражение: |
| for (expr1; expr2; expr3) {  doSomething();  } | expr1;  while (expr2;) {  doSomething();  expr3;  }. |

1. Для удобства исполнения, оператор for преобразуется в оператор while переносом команд по следующему принципу:
2. Для удобства исполнения условные операторы if, elif, else и оператор while добавляют   
   операторы JMP index, которые указывают, что необходимо сменить исполняемую лексему на лексему с индексом index в векторе лексем. Добавление происходит по следующему принципу:

|  |  |
| --- | --- |
| Исходное выражение: | Преобразованное выражение: |
| if (expr1) {  do1();  }  elif (expr2) {  do2();  }  else {  do3()  }; | if (expr1) JMPIF elifIndex  do1();  JMP endOfBlockIndex  elif (expr2) JMPIF elseIndex  do2(); JMP endOfBlock  else {  do3();  }  EndOfBlockIndex |
| Исходное выражение:  while (expr) {  doSomething();  } | Преобразованное выражение:  while (expr) JMPIF EndOfBlockIndex  doSomething();  JMP whileIndex  EndOfBlock |

Здесь JMPIF означает, что прыжок произойдет только в том случае, если выражение expr будет ложно. JMP означает, что переход произойдет в любом случае.

1. Затем вызывается метод execute, который пытается найти функцию int main() в таблице функций. Если такой функции нет – выбрасывается исключение. Иначе начинается выполнение.
2. Каждая функция execute принимает некоторую функцию, обрабатывает аргументы, добавляя их в таблицу переменных, присваивает в них входящие значения аргументов. Выполнение происходит следующим образом:
   1. Если попадается лексема – тип данных, то переменна, следующая после неё, добавляется в таблицу переменных.
   2. Если попадается условный оператор, выполнение продолжается в соответствие с перемещением выполняемого командного слова в соответствие с индексом в JMP.
   3. Если попадается return, функция execute завершается и возвращает значение, которое было вычислено, в форме неименованной константы.
   4. Во всех остальных случаях создается экземпляр класса calculator, в который передается выражение до «;» (или до «)» в случае условных операторов) и таблица переменных.
3. Класс calculator вызывает функцию calculate, производит разбор выражения в обратную польскую запись для вычисления выражения, подставляя вместо имен переменных значения этих переменных из входящей таблицы, изменяя значение этих переменных в случае вызова оператора присваивания. Если в процессе вычисления выражения необходимо вычислить значение пользовательской функции (т.е. любой, кроме стандартной), вызывается метод execute класса interpretator для вычисления этой функции.

Рис. 2. Схематичная диаграмма вызова двух основных классов программы interpretator и calculator.

Проиллюстрируем парсинг программы на простом примере:

1. Исходный код программы.

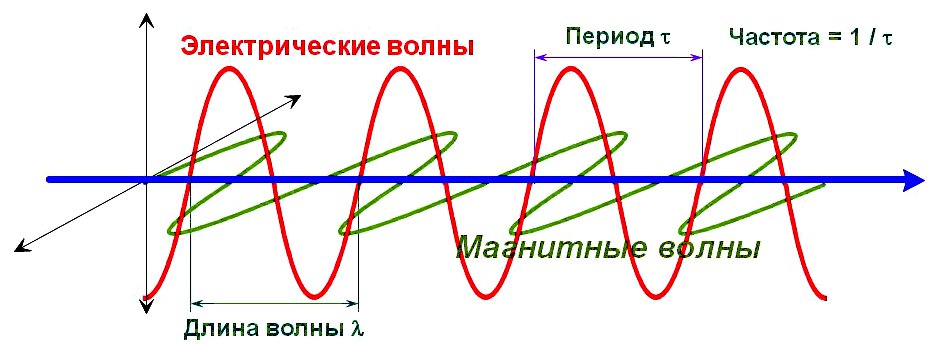


Рис. 3. Исходный код примера пользовательской программы.

1. Разделение кода на слова по пробелам, специальным символам и операторам.

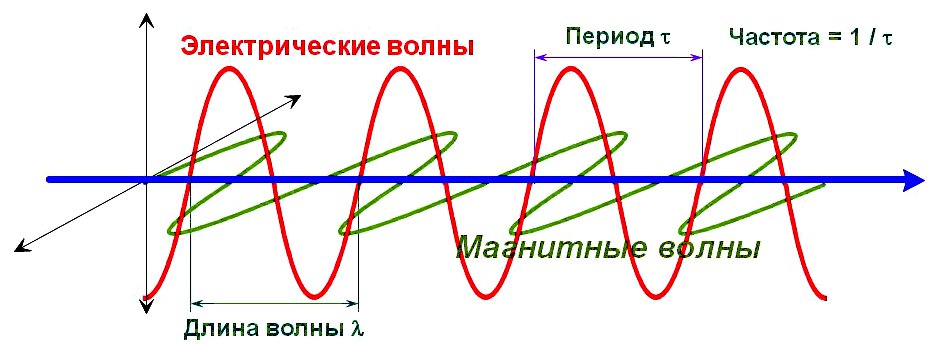


Рис 4. Исходый код примера пользовательской программы, разделенный пробелами по ключевым словам.

1. Разбор слов по соответствующим лексемам, вставка операторов JMP. Первое число обозначает индекс лексемы, затем выводится информация о каждой лексеме методом showInfo.

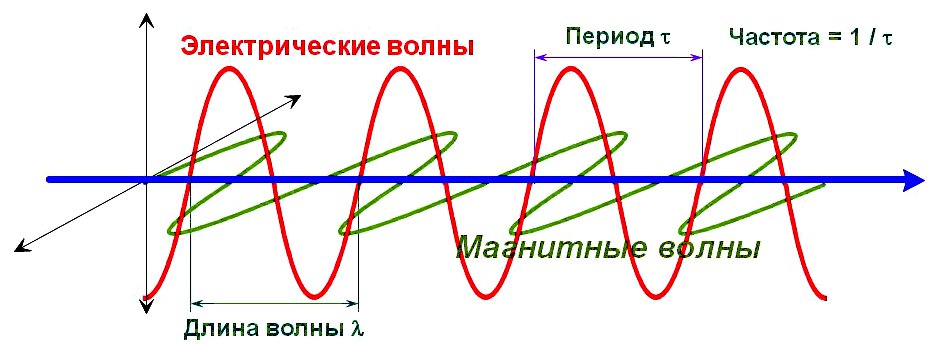


Рис. 5. Массив лексем исходной программы с информацией о каждой лексеме.

1. Схематичное представление создания экземпляров классов interpretator и calculator и вызова функций execute и calculate соответственно.

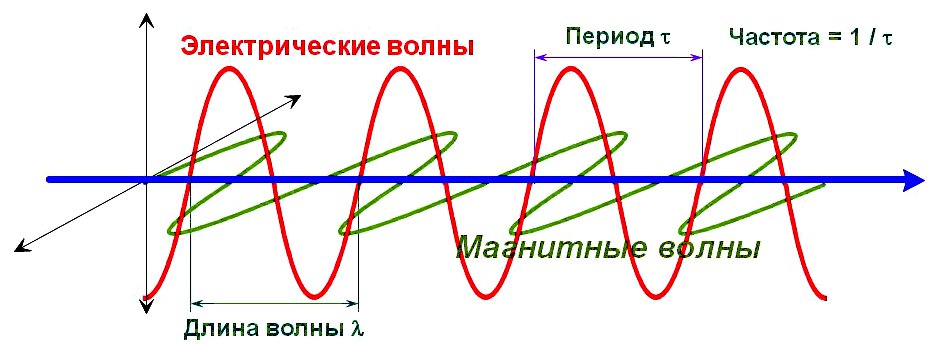


Рис. 6. Диаграмма вызовов функций классов interpretator и calculator.

# Подтверждение корректности

Для подтверждения корректности в программе значения функций сравнивались с значениями тех же функций при тех же аргументах, реализованных в библиотеке cmath. В результате при программа относительно точно вычисляла значения данных функций.

# Результаты экспериментов

Эксперименты проводились следующим образом: выбирался отдельный метод суммирования, затем происходило вычисление всех 4-х функций, после чего в файл записывалась разность значения функции из cmath и функции из \_math для одного аргумента. Вычисления производились для типа данных float, аргумент принимал значения от 0 до 64 с шагом в .

После первых нескольких экспериментов было обнаружено, что для слишком маленьких значений – количества итераций – точность сильно снижается, а при больших значениях функция обратного суммирования вычисляла ряд с большой ошибкой. Поэтому эмпирически была получена функция зависимости количества итераций от значения переменной , где – некоторые константы.

Сперва были получены точечные диаграммы разности значения функции из библиотеки cmath и значения функции суммирования для каждой функции и каждого типа суммирования на разных масштабах. Приведу некоторые из них:

*Рис. 1. Диаграммы разности значений прямой и попарной суммы для функции синуса и значений функции синуса из cmath.*

Как можно заметить здесь, попарная сумма синуса начинает расходиться немного раньше прямой суммы. Как можно будет увидеть позже, прямая сумма для синуса точнее, чем попарная.

*Рис. 2. Диаграммы разности значений обратной суммы функции синуса и обратной суммы функции синуса с массивом и значений функции синуса из cmath.*

На данных диаграммах видно, что обратная сумма синуса без массива начинает раньше расходиться и обладает большей погрешностью вблизи 0. Это может быть связано с тем, что вблизи нуля значения синуса малы, поэтому погрешность получения последнего члена и погрешность деления для получения старших членов в сумме наиболее заметны.

*Рис. 4. Диаграммы разности значений прямой суммы функции натурального логарифма и обратной суммы функции натурального логарифма с массивом и значений функции натурального логарифма из cmath.*

На данных диаграммах видно, что для обоих типов суммирования характерна большая погрешность при маленьких значениях аргумента. Но для обратной суммы погрешность гораздо больше и доходит до значений . Я связываю это с неверным получением последнего члена для суммы.

*Рис. 4. Диаграммы разности значений прямой суммы функции синуса, косинуса и натурального логарифма и значений функций синуса, косинуса и натурального логарифма из cmath.*

Как видно из диаграмм, все суммы принимают наиболее точные значения вблизи нулей функций.

Остальные диаграммы смотрите в приложении.

Следующим этапом в исследовании было сравнение точности суммирований. Для этого среди 4х значений разности выбиралось ближайшее к нулю, а тот способ, который достигал этого значений, получал один балл. Способ, набравший наибольшее количество баллов лучше вычислял значение функции. Приведу сюда таблицу сумм баллов:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Функция | Прямая сумма | Попарная сумма | Обратная сумма | Обратная сумма с массивом |
| Синус | 26653 | 16386 | 25583 | 25583 |
| Косинус | 17789 | 14718 | 21753 | 17812 |
| Экспонента | 20678 | 25743 | 10335 | 38513 |
| Логарифм | 16061 | 24078 | 14952 | 45766 |

*Рис. 5. Таблица сумм баллов за наиболее точное вычисление различных функций для различных способов суммирования. Зеленым цветом обозначен наиболее точный для данный функции способ суммирования, голубым – второй по точности.*

По итогам исследования можно заметить, что для синуса наиболее точной оказалась прямая сумма, немного менее точными стали обе обратные суммы. Для косинуса же обратная сумма была заметно точнее прямой суммы и обратной суммы с массивом. Данные между синусом и косинусом так различаются, поскольку вблизи точки 0 косинус находится вблизи 1, что понижает общую точность функций. Попарная сумма оказалась наименее точной, так как для этих функций характерна разность знаков соседних членов, а попарная сумма эффективна только в случае, если соседние члены достаточно разных порядков. В случае с тригонометрическими функциями, сами попарные суммы оказывались разных порядков больше, чем обычные члены в прямой сумме. Прямая сумма оказалась формально точнее обратных для синуса за счёт значений при больших . Для косинуса прямая сумма и обратная сумма с массивом оказались практически равной точности, а так как обратная сумма имеет свойство занижать результат, то при больших она оказалась формально точнее.

Для экспоненты и логарифма ситуация другая: поскольку для обеих функций не характерна разница знаков в соседних членов, но характерна быстрая сходимость, попарная сумма оказалась довольно точной. Обратная сумма с массивом стала наиболее точной как при маленьких , за счет точного вычисления последних битов, так и при больших , за счет точных первых битов и порядка благодаря первому члену суммы, который не суммировался с соседним как в случае с попарной суммой.

# Заключение

По результатам проведенных экспериментов были сделаны следующие выводы:

* Все способы недостаточно точны из-за самого принципа многочлена Маклорена: чем дальше от , тем меньше точность. Из-за этого при достаточно больших некоторые способы оказывались формально точнее, хотя все они были далеки от действительного значения.
* Для расчёта тригонометрических функций довольно точным оказался способ прямого суммирования из-за разницы знаков соседних членов. Для расчёта экспоненты и логарифма точным оказался способ попарного суммирования из-за большой разности между соседними членами. Для всех 4-х функций точным был способ обратного суммирования с массивом. Обычный способ обратного суммирования был менее точным из-за погрешности получения последнего члена.

# Приложение