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](#_okigpg9oilmn)

[Метод решения 4](#_u3jriflcaycf)

[Руководство пользователя 5](#_4u7yqcntq33k)

[Описание программной реализации 7](#_p7lq2s512gyv)

[Подтверждение корректности 16](#_fk8gik7mnrgu)

[Заключение 17](#_sitdi56wbyve)

[Источники 18](#_ymannoj7hpz3)

[Приложение 19](#_7kivbe4asgh8)

# 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 C89 [[2]](#_ymannoj7hpz3).
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.

# Метод решения

В качестве метода решения было выбрано написание программы на C++14 использованием некоторых встроенных библиотек. Возможно было использование только тех библиотек, которые не решали конкретно задачу интерпретации языка программирования. Для использования такого метода потребовалось:

1. Спроектировать структуру программы и разбиения на файлы для объявления и реализации различных классов.
2. Проанализировать, какие элементы проекта были реализованы ранее, какие элементы реализованы во встроенных библиотеках, а какие элементы необходимо написать, определить подзадачи проекта, а также список поддерживаемых функций и команд.
3. Распределить подзадачи между участниками данной лабораторной работы и выполнить все подзадачи.
4. Проверить программу на корректность, проанализировать результаты.

В качестве одной из подзадач был выбран подсчет арифметических выражений, который был реализован через обратную польскую запись [[3]](#_ymannoj7hpz3) в одной из предыдущих лабораторных работ [[1]](#_ymannoj7hpz3).

Обратная польская запись представляет собой представление арифметических выражений, удобное для алгоритмического вычисления. В такой записи, в отличие от привычной инфиксной нотации, операнды записываются перед операцией. Это позволяет избежать скобок в выражении, а также скачков по выражению при его вычислении. Помимо достоинств такой записи, существуют и недостатки, например, становится невозможным использование одного символа для унарного и бинарного минусов.

Структура программы представляет собой 2 базовых класса: calculator для вычисления арифметических выражения и interpretator для контроля исполнения программы, и несколько классов, соответствующих различным типам лексем. Лексемы реализованы через наследование от общего виртуального класса, описывающего основные методы и поля каждой лексемы.

# 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. Реализовано лексикографическое сравнение строк.
   2. Арифметические операции: =, +, +=, -, -=, \*, \*=, /, /=, % (для целочисленных типов данных), %= (для целочисленных типов данных). Реализована конкатенация строк оператором +, +=.
2. Типы данных: void (для возвращаемых значений функций), int, string, double, а также n-мерные массивы с каждым из типов данных, кроме void. Объявление массива и индексация по нему производятся одинаково за исключением того, что перед объявлением должен стоять тип данных. Например int A[5][4]; A[4][3] = 0.
3. Глобальные переменные, объявляются и инициализируются в самом начале текста программы до первой функции.
4. Условные операторы: if, elif, else.
5. Операторы циклов: for, while. В for НЕ поддерживается объявление итерационной переменной в первом условии.
6. Стандартные предопределенные функции:
   1. print(var\_1, var\_2, …, var\_n) – выводит в консоль n переменных через пробел, после выводит специальный символ новой строки.
   2. scan(var\_1, var\_2, …, var\_n) – запрашивает из консоли n переменных через клавишу новой строки (enter).
   3. Математические функции: abs(var), sin(var), cos(var), sqrt(var).
7. Прочие ключевые слова и специальные символы:
   1. return – используется для возврата значений из функций.
   2. Символ «;» используется для конца команды, «,» для объявления аргументов функции, «{}» для выделения блоков тел функций, условных операторов и операторов цикла, «()» для изменения приоритета операций, для условий операторов, а также для вызова и объявления функций.
8. 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. Файл code/tests.cpp тестового кода для программы.
4. include/calculator.h – заголовочный файл описания класса calculator для вычислений арифметических выражений.
5. include/interpretator.h – заголовочный файл описания класса interpretator для предобработки пользовательского кода, обработки условных операторов.
6. include/lexems/variable.h – заголовочный файл описания класса variable для переменных программы.
7. include/lexems/operation.h – заголовочный файл описания класса operation арифметических и логических операций над переменными.
8. include/lexems/specialLexem.h – заголовочный файл описания класса specialLexem специальных символов, таких как скобки приоритета, символа «;» и так далее.
9. include/lexems/myoperators.h – заголовочный файл описания класса myoperators ключевых слов, в том числе условных операторов и операторов цикла.
10. include/lexems/operand.h – заголовочный файл описания абстрактного класса операндов operand, общий класс для констант и переменных.
11. include/lexems/function.h – заголовочный файл описания класса function для функций программы.
12. include/lexems/keyWords.h – заголовочный файл описания абстрактного класса keyWords ключевых слов, поддерживаемых интерпретаторов, таких как типы данных и условные операторы.
13. include/lexems/dataType.h – заголовочный файл описания класса dataType типов данных, используемых в программе.
14. include/lexems/constant.h – заголовочный файл описания класса констант constant, использующегося для литералов и неименованных констант возврата из арифметических выражений и функций.
15. include/lexems/commonLexem.h – заголовочный файл описания абстрактного класса commonLexem, общего для всех лексем программы.
16. src/main.cpp – файл функции main, где происходит инициализация класса interpretator.
17. src/\*.cpp, src/lexems/\*.cpp – соответствующие вышеописанным файлам реализации соответствующих классов.
18. sln – папка проекта Visual Studio 2022.
19. README.md – файл-аннотация для репозитория.
20. Презентация к данному проекту «Презентация\_лабораторная\_Интерпретатор\_Загрядсков\_Максим\_Болтенков\_Станислав\_3823Б1ПМ1\_1.pptx».

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.
  + setArr(int) – устанавливает значение размерности массива переменной.
  + setSizes/getSizes – устанавливает/возвращает вектор размеров размерностей массива.

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 переменная, 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 функции, вызывается метод execute класса interpretator для вычисления этой функции. Процесс обработки n-мерных массивов классом calculator описан ниже.

Класс interpretator

Вычисление арифметических выражений

Класс calculator

Вызов функций

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

Проиллюстрируем парсинг программы на простом примере:

1. Исходный код программы.

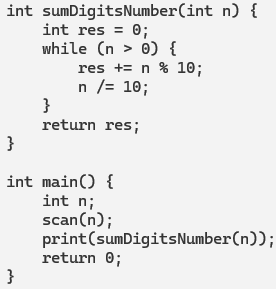


Рис. 3. Исходный код примера пользовательской программы.

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

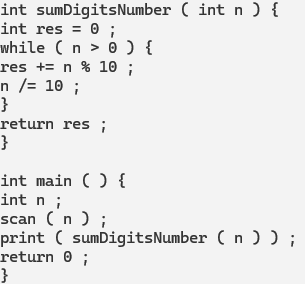


Рис 4. Исходый код примера пользовательской программы, разделенный пробелами по ключевым словам.

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

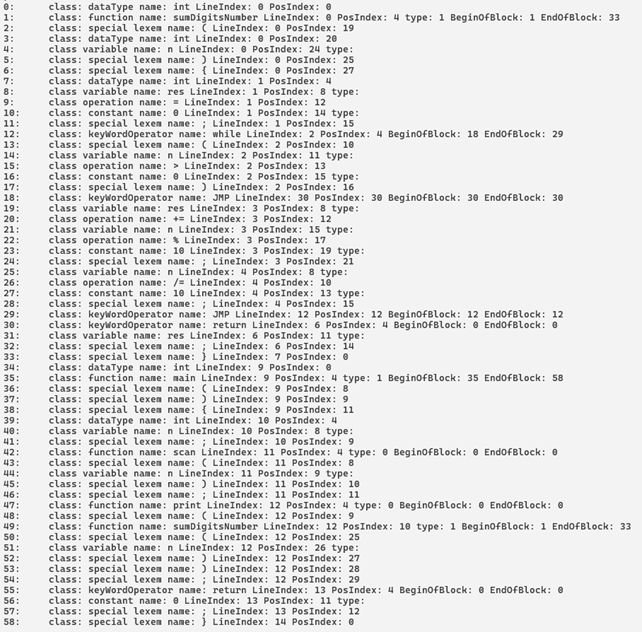


Рис. 5. Массив лексем исходной программы с информацией о каждой лексеме.

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

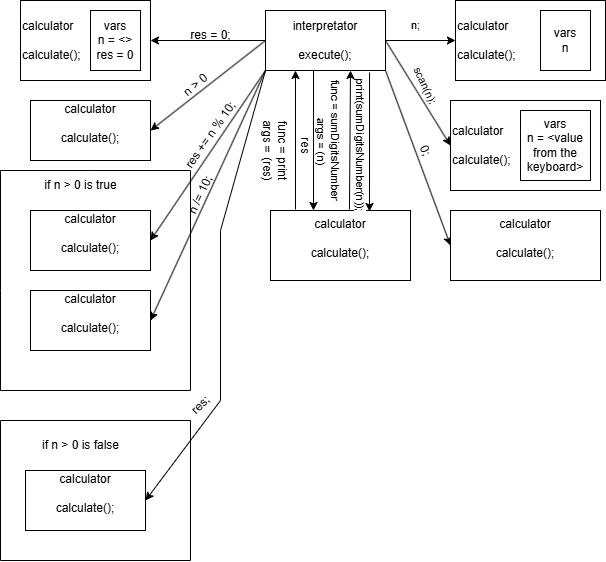


Рис. 6. Диаграмма вызовов функций классов interpretator и calculator.

Реализация обработки n-мерных массивов осуществляется следующим образом: при объявления массива вычисляется вектор размерностей (), где – размер размерности массива, . Метод setSizes класса variable принимает вектор размерностей, и выделяет память для одномерного массива соответствующей длины .

В ходе исполнения текста программы при обращении к элементу массива вычисляется сдвиг (), для вычисления арифметического выражения вместо подставляется переменная, находящиеся по адресу , где mas - указатель на начало массива. Дальше все вычисления проходят, как с переменной.

Формула вычисления сдвига, при обращении к элементу массива:

1. Для одномерного массива .
2. Для двумерного массива .
3. Для трёхмерного

Где ,- соответствующие размерности массивов.

Формула легко обобщается для обращения к элементу n-мерного массива: .

# Подтверждение корректности

Для подтверждения корректности был запущен тестовый пример пользовательской функции (код данной функции можно посмотреть в приложении), который использует все стандартные функции, операторы, типы данных, ключевые слова, а также рекурсию. После того, как данный код исполнился и завершился корректно, были проведены тесты на обработку ошибок в пользовательской программе, например, на отсутствие «;» или на отсутствие закрывающей скобки «)».

# Заключение

* Была спроектирована и создана программа, осуществляющая интерпретацию C-подобного кода с поддержкой наибольшего возможного количества ключевых слов.
* Была проверена корректность выполнения этой программы на тестовом примере.

# Источники

1. <https://github.com/ValentinV95/mp2-2024-lab3-postfix/pull/3>
2. <https://en.wikipedia.org/wiki/ANSI_C>
3. [https://ru.wikipedia.org/wiki/Обратная\_польская\_запись](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D0%BB%D1%8C%D1%81%D0%BA%D0%B0%D1%8F_%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%8C)

# Приложение

Код тестовой пользовательской программы:

int GLOB = -1;

void testIntOperation() {

print("Testing int operators...");

int a = 5;

if (a != 5) {

print("faild testIntOperation a != 5");

}

int b = a + 3;

if (b != 8) {

print("faild testIntOperation b != 8");

}

int c = b - 2;

if (c != 6) {

print("faild testIntOperation c != 6");

}

int d = c \* 2;

if (d != 12) {

print("faild testIntOperation d != 12");

}

int e = d / 3;

if (e != 4) {

print("faild testIntOperation e != 4");

}

int f = d % 5;

if (f != 2) {

print("faild testIntOperation f != 2");

}

f += 3;

if (f != 5) {

print("faild testIntOperation f != 5");

}

f -= 2;

if (f != 3) {

print("faild testIntOperation f != 3");

}

f \*= 4;

if (f != 12) {

print("faild testIntOperation f != 12");

}

f /= 3;

if (f != 4) {

print("faild testIntOperation f != 4");

}

f %= 3;

if (f != 1) {

print("faild testIntOperation f != 1");

}

int g = (a == 5) && (b == 8);

if (g == 0) {

print("faild testIntOperation g == 0");

}

int h = (a == 0) || (b == 8);

if (h == 0) {

print("faild testIntOperation h == 0");

}

if (a != 5) {

print("faild testIntOperation a != 5");

}

if (a == 6) {

print("faild testIntOperation a == 6");

}

if (a >= 6) {

print("faild testIntOperation a >= 6");

}

if (a > 6) {

print("faild testIntOperation a > 6");

}

if (b <= 7) {

print("faild testIntOperation b <= 7");

}

if (b < 8) {

print("faild testIntOperation b < 8");

}

return;

}

void testDoubleOperation() {

print("Testing double operators...");

double a = 5.5;

if (a != 5.5) {

print("faild testIntOperation a != 5.5");

}

double b = a + 3.2;

if (b != 8.7) {

print("faild testIntOperation b != 8.7");

}

double c = b - 2.1;

if (c != 6.6) {

print("faild testIntOperation c != 6.6");

}

double d = c \* 2.0;

if (d != 13.2) {

print("faild testIntOperation d != 13.2");

}

double e = d / 3.0;

if (e != 4.4) {

print("faild testIntOperation e != 4.4");

}

e += 1.1;

if (e != 5.5) {

print("faild testIntOperation e != 5.5");

}

e -= 0.5;

if (e != 5.0) {

print("faild testIntOperation e != 5.0");

}

e \*= 2.0;

if (e != 10.0) {

print("faild testIntOperation e != 10.0");

}

e /= 4.0;

if (e != 2.5) {

print("faild testIntOperation e != 10.0");

}

int g = (a == 5.5) && (b == 8.7);

if (g == 0) {

print("faild testIntOperation g == 0");

}

int h = (a == 0.0) || (b == 8.7);

if (h == 0) {

print("faild testIntOperation h == 0");

}

if (a != 5.5) {

print("faild testIntOperation a != 5.5");

}

if (a == 5.6) {

print("faild testIntOperation a == 5.6");

}

if (a >= 6.0) {

print("faild testIntOperation a >= 6.0");

}

if (a > 5.5) {

print("faild testIntOperation a > 5.5");

}

if (b <= 8.6) {

print("faild testIntOperation b <= 8.6");

}

if (b < 8.7) {

print("faild testIntOperation b < 8.7");

}

return;

}

void testStringOperation() {

print("Testing string operators...");

string a = "Hello";

if (a != "Hello") {

print("faild testStringOperation a != Hello")

}

string b = a + " World";

if (b != "Hello World") {

print(b)

print("faild testStringOperation b != Hello World")

}

b += "!";

if (b != "Hello World!") {

print("faild testStringOperation b != Hello World!")

}

if (a != "Hello") {

print("faild testStringOperation a != Hello");

}

if (a == "hello") {

print("faild testStringOperation a != hello");

}

if ("aaa" > "bbb") {

print("faild testStringOperation aaa > bbb");

}

if ("abb" >= "bbb") {

print("faild testStringOperation abb >= bbb");

}

if ("bbb" < "aaa") {

print("faild testStringOperation bbb < aaa");

}

if ("bbb" <= "abb") {

print("faild testStringOperation bbb <= abb");

}

int c = (a == "Hello") && (b == "Hello World!");

if (c == 0) {

print("faild testStringOperation c == 0");

}

int d = (a == "hello") || (b == "Hello World!");

if (d == 0) {

print("faild testStringOperation d == 0");

}

return;

}

int testElse(int n) {

if (n == 0) {

return 10;

}

else {

return 30;

}

return;

}

int testElif(int n) {

if (n == 0) {

return 10;

}

elif (n == 1) {

return 15;

}

elif (n == 2) {

return 20;

}

else {

return 30;

}

return;

}

int testWhile(int a) {

int i = 0;

while (i < a) {

i += 1;

}

return i;

}

int testFor(int a) {

int cnt = 0;

for (int i = 0; i < a; i+=1) {

cnt += 1;

}

return cnt;

}

void testOperators() {

print("Testing elif operator...");

if (testElif(0) != 10) {

print("faild testElif(0) != 10");

}

if (testElif(2) != 20) {

print("faild testElif(2) != 20");

}

if (testElif(1) != 15) {

print("faild testElif(1) != 15");

}

if (testElif(3) != 30) {

print("faild testElif(3) != 30");

}

print("Testing else operator...");

if (testElse(0) != 10) {

print("faild testElse(0) != 10");

}

if (testElse(2) != 30) {

print("faild testElse(2) != 30");

}

if (testElse(1) != 30) {

print("faild testElse(1) != 30");

}

if (testElse(3) != 30) {

print("faild testElse(3) != 30");

}

print("Testing for operator...");

if (testFor(10) != 10)

{

print("faild testFor(10) != 9");

}

if (testFor(-5) != 0)

{

print("faild testFor(-5) != 0");

}

print("Testing while operator...");

if (testWhile(10) != 10)

{

print("faild testWhile(10) != 9");

}

if (testWhile(-5) != 0)

{

print("faild testWhile(-5) != 0");

}

return;

}

int fibonacci(int n) {

if (n <= 2) {

return 1;

}

else {

return fibonacci(n - 1) + fibonacci(n - 2);

}

}

void testRecursion() {

print("Testing recursion...");

if (fibonacci(-1) != 1) {

print("faild fibonacci(-1) != 1");

}

if (fibonacci(1) != 1) {

print("faild fibonacci(1) != 1");

}

if (fibonacci(8) != 21) {

print("faild fibonacci(8) != 21");

}

if (fibonacci(3) != 2) {

print("faild fibonacci(3) != 2");

}

if (fibonacci(9) != 34) {

print("faild fibonacci(9) != 34");

}

if (fibonacci(2) != 1) {

print("faild fibonacci(2) != 1");

}

if (fibonacci(5) != 5) {

print("faild fibonacci(5) != 5");

}

return;

}

void testArrays() {

print("Testing arrays1...");

int n1 = 4;

int n2 = 5;

int n3 = 2;

int n4 = 6;

int n5 = 3;

int i;

int j;

int k;

int l;

int p;

int A[n1][n2][n3][n4][n5];

for (i = 0; i < n1; i += 1) {

for (j = 0; j < n2; j += 1) {

for (k = 0; k < n3; k += 1) {

for (l = 0; l < n4; l += 1) {

for (p = 0; p < n5; p += 1) {

A[i][j][k][l][p] = i \* j \* k \* l \* p;

} } } } }

for (i = 0; i < n1; i += 1) {

for (j = 0; j < n2; j += 1) {

for (k = 0; k < n3; k += 1) {

for (l = 0; l < n4; l += 1) {

for (p = 0; p < n5; p += 1) {

if (i \* j \* k \* l \* p != A[i][j][k][l][p]) {

print("error", "A[", i, "]", "[", j, "]", "[", l, "]", "[", k, "]", "[", p, "] != ", i \* j \* k \* l \* p, A[i][j][k][l][p]);

} } } } } }

return;

}

void testArrays2()

{

print("Testing arrays2...");

int n1 = 4;

int n2 = 5;

int i;

int j;

double A[n1][n2];

for (i = 0; i < n1; i += 1) {

for (j = 0; j < n2; j += 1) {

A[i][j] = sqrt(i \* j);

} }

for (i = 0; i < n1; i += 1) {

for (j = 0; j < n1; j += 1) {

if (sqrt(i \* j) != A[i][j]) {

print("error", "A[", i, "]", "[", j, "] != ", sqrt(i \* j), A[i][j]);

} } }

return;

}

void testArrays3() {

print("Testing arrays3...");

int n1 = 5;

int n2 = 7;

int i;

int j;

int A[n1][n2];

for (i = 0; i < n1; i += 1) {

for (j = 0; j < n2; j += 1) {

A[i][j] = i \* j;

} }

double B[A[4][4]][A[3][6]];

int n3 = 16;

int n4 = 18;

for (i = 0; i < n3; i += 1) {

for (j = 0; j < n4; j += 1) {

B[i][j] = sqrt(i \* j);

} }

for (i = 0; i < n3; i += 1) {

for (j = 0; j < n4; j += 1) {

if (sqrt(i \* j) != B[i][j]) {

print("error", "B[", i, "]", "[", j, "] != ", sqrt(i \* j), B[i][j]);

} } }

return;

}

void testArrays4() {

print("Testing arrays4...");

int n1 = 5;

int n2 = 7;

int i;

int j;

string A[n1][n2];

for (i = 0; i < n1; i += 1) {

for (j = 0; j < n2; j += 1) {

if (i > 0 && j > 0) {

A[i][j] = "a" + A[i-1][j] + A[i][j-1];

} else {

A[i][j] = "";

}

} }

for (i = 0; i < n1; i += 1) {

for (j = 0; j < n2; j += 1) {

if (i > 0 && j > 0) {

if (A[i][j] != "a" + A[i-1][j] + A[i][j-1]) {

print("error", "A[", i, "]", "[", j, "] != ", "expected...", A[i][j]);

}

} else {

if (A[i][j] != "") {

print("error", "A[", i, "]", "[", j, "] != ", "empty", A[i][j]);

}

}

} }

return;

}

void testArrays5() {

print("Testing arrays5...");

int n1 = 5;

int i;

int A[n1];

int B[n1];

for (i = 0; i < n1; i += 1) {

A[i] = i;

}

for (i = 0; i < n1; i += 1) {

B[A[n1 - A[i] - 1]] = i;

}

for (i = 0; i < n1; i += 1) {

if (n1 - i - 1 != B[i]) {

print("error", "B[", i, "]", " != ", i, B[i]);

}

}

return;

}

void processGlobal() {

print("Testing processGlobal...");

if (GLOB != (-1)) { print("error: global variable isn't valid", GLOB); }

GLOB = GLOB + 1;

return;

}

void testGlobal() {

print("Testing testGlobal...");

if (GLOB != 0) { print("error: global variable isn't valid", GLOB); }

return;

}

int main() {

testIntOperation();

testDoubleOperation();

testStringOperation();

testOperators();

testRecursion();

testArrays();

testArrays2();

testArrays3();

testArrays4();

testArrays5();

processGlobal();

testGlobal();

return 0;

}