Тема 1: Source beautifier:

Даден е синтактично коректен “src” файл, на подмножество на C++/JS/Java/. Да се напише програма, която по дефиниран “style guide”, форматира текста. Например:

Входен файл:

```
#include<iostream> int main{return 0;}
```

Резултат:

```
#include <iostream>

int main
{
    return 0;
}
```

Вие можете да дефинирате (в документ към проекта) какво точно подмножество на езика използвате. Също така дефинирайте правила за подреждане (style guide), но така че да могат да бъдат лесно променяни.

Тема 2: Интерпретатор:

Да се напише интерпретатор на езика, определен със запазените думи:
LABEL, GOTO, LET, READ, PRINT, IF, ENDIF, ELSE, WHILE, DONE, PUSH, POP
И оператори =, ==, !=, <, <=, >, >=, +, *, /, -, %, &&, ||, !
Работещ над целочислени данни. Вашият интерпретатор трябва да може да работи над подадени текстови файлове, както и в REPL режим. Проверявайте за синтактични грешки и за всички такива изведете подходящи съобщения.

- С LABEL се задава етикет за безусловен преход адресиран с GOTO. Може да се разполага на произволно място в програмата.
- С LET се въвежда променлива. Всяка променлива трябва да бъде обявена преди да се използва. Областта на действие е от декларацията до края на блока.
- READ и PRINT се използват за четене на стойности от стандартния вход и извеждане на екрана на стойност.
- Конструкцията за условен преход IF има следният синтакси:

```
IF <булев или целочислен израз> <блок код> [ELSE <блок код>] ENDIF
```

- Конструкцията за цикъл WHILE има следниятсинтаксис:

```
WHILE <условие> <блок код> DONE
```

- PUSH и POP съответно съхраняват един елемент в стека и изваждат стойността на един елемент от стека. Тези конструкции могат да се използват за реализиране на рекурсивни схеми, както и на масиви или други не-скаларни типове данни.

Като примери за използване на вашия интерпретатор напишете на този език програма, която въвежда числа и намира най-малкото от тях. За по-заинтересованите, напишете също и програма, която прочита определен брой числа, сортира ги и ги извежда на екрана.

Пример за програма, която намира сума на определен брой числа:

```
LET count
READ count
IF count < 0
      GOTO end
ENDIF

LET sum
sum = 0
WHILE count > 0
      LET x
      READ x
      sum = sum + x
      count = count - 1
DONE
PRINT sum

LABEL end
```

Тема 3: Алокатор за памет:

Библиотеката, която реализирате в този проект, трябва да работи върху предварително заделен блок памет. Тя трябва да отговаря на команди `alloc <size>` и `free <ptr>`. Трябва да изберете ефективен алгоритъм за разпределение на паметта, който да работи с произволни по размер блокове. Валидирайте работата с указателите (т.е. предпазете се от грешно или двойно освобождаване на памет). Демонстрирайте работата с различни алгоритми. Ако можете - демонстрирайте употреба със стандартни контейнери в stl.

Тема 4: Архиватор на директории

Напишете програма, която архивира съдържанието на подадена директория (файлове и поддиректории) във файл.
Трябва да се поддържат следните функционалности:

- създаване на архив по подадена директория (create)
- разархивиране на архив (extract)
- извеждане на списък на съдържанието на архив (list)
- добавяне на файл към вече създаден архив (add)
- премахване на файл от архив (remove)

При създаване на файл или добавяне на файл към архив трябва да поддържате възможност за защита на съдържанието на файла с парола.
При подаване на файловете, които да се добавят да се поддържат маски, * и ?, като за целта вие трябва да имплементирате работата с тях.

Тема 5: Хъфман-компресиране.

Напишете програма, която компресира множество файлове в една директория в един файл, използвайки алгоритъма на Хъфман. Трябва да се поддържат следните функционалности:

- създаване на архив по подаден списък от файлове (zip)
- разархивиране на един или всички файлове от архива (unzip)
- извеждане на списък на компресираните файлове с информация за нивото на компресия (info)

Тема 6: Файлова система върху единичен файл.

Напишете програма, която симулира файлова система върху единичен файл.
Трябва да поддържате следните функционалности:

- създаване на директория (mkdir)
- изтриване на празна директория (rmdir)
- извеждане на съдържанието на директория (ls)
- копиране на файл (cp)
- изтриване на файл (rm)
- извеждане на съдържанието на файл (cat)
- записване на текст към файл (write). Винаги се създава празен файл (ако такъв съществува се презаписва)
- добавяне на текст към края на файл (append). Ако файлът не съществува се създава празен

Напишете прост команден интерпретатор за работа с файловата система.

Тема 7: Минимизатор на C++ код.

Напишете програма, която получава файл с валиден код на езика C++ и го минимизира, като премахва всички интервали. Също променя имената на всички идентификатори, така че да станат с минимална дължина. Вашите корекции не трябва да променят семантиката на кода. Не се предполага да има overloading на функции. За по-лесна проверка и все пак минимум четивност - нека всяка функция започва на нов ред.
Напълно е възможно да има глобални променливи, както и класове и именовани пространства.

Пример:
Вход:

```
#include <iostream>

int sum (int a1, int a2)
{
    return a1 + a2;
}

int main
{
    int value1 = 0, var2 = 10;
    std::cout << sum (value1, var2) << endl;
    return 0;
}
```

Изход:

```
#include<iostream>
int f(int a,int b){return a+b;}
int main{int a=0,b=10;std::cout<<f(a,b)<<endl;return 0;}
```

Тема 8: Basic Version Control System

Увод

Системите за контрол на версиите са програми, които се използват, за да улеснят работа на разработчиците на софтуер. Те имат няколко основни предназначения:

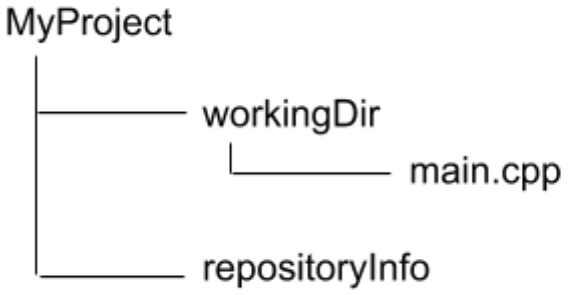
- да следят промените в кода на един проект и извършителите им
- да се контролира разработката на различни части от проекта
- да се синхронизира работата на разработчиците.

Вашата задача е да разработите проект, реализиращ част от тези действия.

Описание

Директорията, в която се намира следеният от програмата код, ще наричаме хранилище (repository/repo). Програмата ще може да работи с различни хранилища, които ще се подават като аргументи на командите. Едно хранилище може да съдържа различните версии на кода, който следи. Работна директория ще наричаме мястото, където седи текущата версия на кода. В хранилището трябва да има и друга директория, в която ще се съхранява информацията за всички commit-и и файловете във всеки един от commit-ите.

Пример за структурата на хранилище:

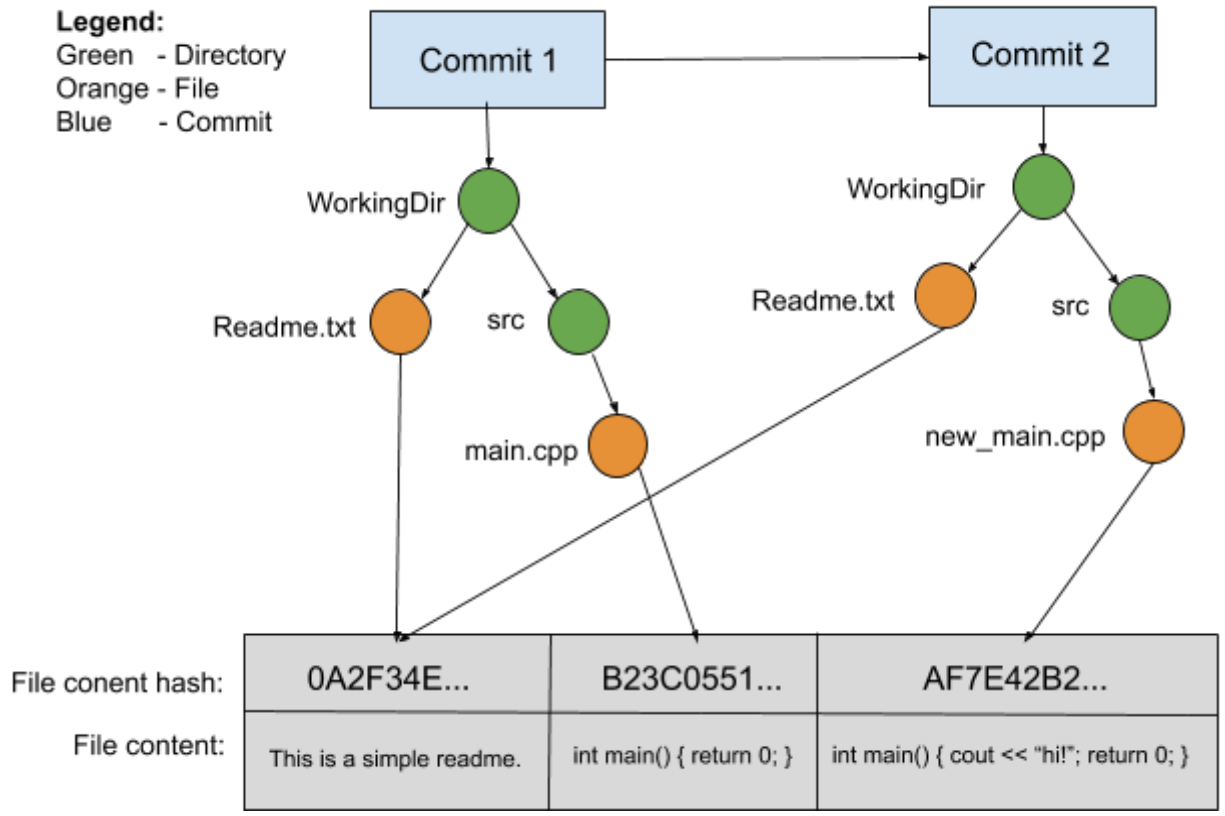


Промените в кода ще се следят чрез snapshot-и на работната директория на проекта, наречени commit-и. Те представят хронологично промените на всички файлове в работната директория, от момента в който е инициализиран проекта, до текущата му версия. Историята на commit-ите е линейна, тоест няма разклонения, за разлика от други системи за контрол на версиите.

Един commit е съставен от всички под-директории и файлове в работната директория, по времето на създаването му. Всеки commit пази **своето id**, **времето**, когато е създаден, **кой го е създал**, **описание**, **имената** на директориите и файловете в него. Структурата на един commit е дървовидна, защото имитира тази на файловите системи.

Понякога се случва разликата на един commit с предишния да е само в един файл. В този случай, ако просто копираме съдържанието на файловете от текущия commit в следващия, ще има голямо разхищение на памет. Причината за това е това, че единствения нов файл ще е този, в който има промяна. Този проблем ще трябва да решите като създадете място за съхранение на уникални файлове в хранилището ви. То ще представлява hash map, който за ключ ще има хешираното съдържание на файла, а за стойност самото съдържание. Така листата на дървото на всеки commit, вместо да са самите файлове, ще са хешове на съдържанието на файловете, сочещи към елемент в hash map-а.

Пример за хранилище с два commit-а:



За хеширането на файловете ще ви трябват готови хеширащи функции като: MD5, SHA1, SHA512 и т.н. Изборът на хешираща функция е ваш. Подходящи библиотеки за това са: Crypto++, Openssl, Libgcrypt и всякакви други, имплементиращи подходящи алгоритми за хеширане.

Не забравяйте, че commit-ите и hash map-а със съдържанието на файловете трябва да се записват на диска, за да могат да се зареждат при всяко пускане на програмата ви, за определено хранилище.

Операции, които трябва да поддържате:

`-init <directory>`
Инициализира дадената директория като хранилище, като добавя нужните файлове/директории, за следенето на състоянието ѝ.

Пример:
```
> myGit.exe init D:\Projects\MyProject
Repository initiated!
```

`-commit <repo directory> <author name> <commit description>`
Прави commit с текущите промени в хранилището, в който присъства даденото име на автора на промените и описание. Възможно е подаване на част от съдържанието -- например отделен файл или под-директория. По желание може да имплементирате и работа с файлови маски. (най-общо * и ?)

Примери:
```
> myGit.exe commit D:\Projects\MyProject Pesho "Added main.cpp"
```

```
Changes commited with id: 1

> myGit.exe commit D:\Projects\MyProject\main.cpp Pesho "Printing greetings"
Changes commited with id: 2
```

`-log <repo directory>`
Извежда по подходящ начин на екрана историята на commit-ите, включително тяхното id, автора им, кога са направени и описанието, свързано с тях.

Пример:

```
> myGit.exe log D:\Projects\MyProject
 Commits:
 1: "Added main.cpp" from 22:30 11.12.2017 by Nikolay Babulkov
 2: "Printing greetings" from 23:00 11.12.2017 by Nikolay Babulkov
```

`-status <repo directory>`
Извежда на екрана информация за текущия commit и за всички файлове в дадената директория, които не са включени в последния commit. Ако няма такива, съобщава това по подходящ начин.

Пример:

```
> myGit.exe status D:\Projects\MyProject
 Current commit: 2
 Not commited:
   new_main.cpp
```

`-checkout <repo directory> <commit id>`
Връща работната директория в състоянието, което е имала, когато е бил създаден commit с даденото id.

Пример:

```
 > myGit.exe checkout D:\Projects\MyProject 1
 Current commit 1: ”Added main.cpp” from 22:30 11.12.2017 by Nikolay Babulkov
```

Когато директорията е върната назад в историята, не може да се правят нови commit-и, докато тя не се върне обратно в последното ѝ състояние.

Връщането към последна версия се случва чрез специална версия на същата команда, където вместо id на commit се подава думата “HEAD”
(HEAD означава id-то на последния commit-в историята).

Пример:

```
 > myGit.exe checkout D:\Projects\MyProject HEAD
 Current commit: 2: ”Printing greetings” from 23:00 11.12.2017         by
 Nikolay Babulkov
```

Горната команда може да се изпълни и когато хранилището е в последната си версия. Тогава работата директория се замества с последния commit, изчиствайки всички промени, които не са били commit-нати.

`-revert <repo directory> <commit id> <file path>`
Ако даденият файл от commit-а, с дадено id, съществува, то той се добавя към работната директория. Ако файл със същия път вече съществува, той се презаписва. Пътят на файла е относителен, спрямо пътя на хранилището. По желание може да имплементирате и поддръжка на файлови маски за пътя.

Например ако искаме да върнем файла C:\MyProject\src\main.cpp от първия commit, можем да използваме:

```
> myGit.exe revert C:\MyProject 1 src\main.cpp
  Added new commit: 3, that reverts the file "src/main.cpp" to its content from
  commit: 1
```

Тема 9: База данни с числа и символни низове

Реализирайте просто СУБД, което поддържа работа с множество таблици. Всяка от тях се състои от произволен брой колони, всяка от които може да е от тип или цяло число, или символен низ. Всяка колона има име - символен низ. Вашите таблици трябва да се съхраняват на диска.
При работа с тази СУБД трябва да можете да ползвате следните команди:

1. За работа с таблици:
   a. ***CreateTable*** - създава нова таблица по подадено име и списък от имената и типовете на съставящите я колони; Като бонус можете да реализирате възможност за стойности по подразбиране или автоматично-генерирани стойности
   b. ***DropTable*** - премахва таблица по нейното име;
   c. ***ListTables*** - извежда списък от имената на всички налични таблици
   d. ***TableInfo*** - извежда информация (схема и брой записи) за таблица по подадено име.
2. За работа с данните:
   a. ***Select*** - Изпълнява заявка за извличане на данни от вашата система. Трябва да се поддържа ***WHERE*** клауза при която можете да филтрирате записи чрез изрази включващи оператори за сравнение (равенство и наредба) върху посочените колони. Добре е да можете да сглобявате по-сложни изрази чрез логически операции (**AND** и **OR**). Също така трябва да можете да задавате подредба на редовете чрез ***ORDER BY*** и да премахвате повторения, чрез ***DISTINCT.*** Не се изисква реализация на ***JOIN***, но реализацията му ще носи допълнителни точки;
   b. ***Remove*** - Премахва определени редове от таблица. Те се посочват с ***WHERE*** клауза, според описанието за ***Select***;
   c. ***Insert*** - Добавя един или повече нови редове в таблица.

Примерна употреба:

```
FMISql> CreateTable Sample (ID:Int, Name:String, Value:Int)
Table Sample created!
FMISql> ListTables
There is 1 table in the database:
       Sample
FMISql> TableInfo Simple
There is no such table!
FMISql> TableInfo Sample
Table Sample : (ID:Int, Name:String, Value:Int)
Total 0 rows (0 KB data) in the table
FMISql> Insert INTO Sample {(1, “Test”, 1), (2, “something else”, 100))
Two rows inserted.
FMISql> Select Name FROM Sample WHERE ID != 5 AND Value < 50
| Name |
--------
|“Test”|
Total 1 row selected
FMISql> Select * FROM Sample WHERE ID != 5 AND Name > “Baba” ORDER BY Name
|ID|      Name      | Val |
---------------------------
| 2|“something else”|  100|
| 1|          “Test”|    1|
Total 2 rows selected
FMISql> Quit
Goodbye, master ;(
```

Тема 10: Интерпретатор за функционалния език **ThisFunc**.

Интерпретаторът трябва да може да се пуска в интерактивен режим, в който позволява на потребителя да пише ред код, който интерпретаторът да оценява и да принтира резултат. Също така вашият интерпретатор трябва да може да се стартира върху файлове които да може да се изпълняват и резултатът от тях да се отпечатва на изхода на вашата програма.

Нека разгледаме как се дефинира езика **ThisFunc**. В този език има само един тип литерали и те са реални числа които ще отбелязваме в това описание с <real-number>. За да е функционален този език ще трябва да може да дефинираме и изпълняваме функции. Изпълнението на една функция ще става чрез нейното име, и списъкът от аргумент на функцията заграден в кръгли скоби. Като може да има функции без аргументи (с празен списък с аргументи).

```
<expression> ::= <real-number> | <function-call>
<function-call> ::= <function-name>([<expression>, ...])
```

Вашият интерпретатор трябва да осигури няколко на брой предварително имплементирани функции, които могат да се извикват наготово: **add, sub, mul, div, pow, sqrt, sin, cos, eq, le, nand**.

- **nand**(#0, #1) връща булевата оценка на `!#0 || !#1`
- **eq**(#0, #1) връща булевата оценка на `#0 == #1`

Освен това езикът трябва да може да позволява да се декларират функции - с име и израз съдържащ техните аргументи. Аргументите на функцията ще се дефинират с цяло число, индексът на аргумента, предхождащ се от символа ‘#’

```
<param-expression> ::= <expression> | #integer | <function-name>([<param-expression>
, ...])

<function-declaration> ::= <function-name> <- <param-expression>
```

Оценката на декларация на функция не връща резултат.

Трябва да поддържате и няколко специални функции:

1. Функцията **if,** приемаща 3 аргумента: `<test>, <if-true>, <if-false>`. Тази функция трябва да изчисли първия аргумент и ако той е различен от 0, то резултатът трябва да е втория аргумент. Ако първият аргумент се оценява на 0, то функцията трябва да върне третия си аргумент.
2. Функцията **list,** приемаща произволен брой аргументи. Тя се оценява на списък от аргументите си.
3. Функцията **head,** която приема един аргумент от тип списък и връща първия елемент на списъка.
4. Функцията **tail,** която приема един аргумент от тип списък и връща нов списък, чиито елементи са елементите на подаденият но без първият.
5. Функцията **map,** която приема 2 аргумента `<function-name>, <list>`. Резултатът от тази функция с списък с брой елементи колкото елементи има в подадения списък (втория аргумент), и стойността на всеки елемент е резултата от оценяването на подадената функция върху стойността от входния списък.
6. Функцията **concat,** която приема 2 аргумента от тип списък и връща нов списък, чиито елементи са елементите на първия списък и към тях са добавени елементите от втория списък

Във някои случаи функциите като **if** и **nand** не е нужно да изчисляват стойността на всичките си аргументи за да изчислят резултатът си.

При четене и изпълнение на кода могат да възникнат няколко типа грешки - грешки при използването на не-декларирани функции, използването на невалидни символи в имената на функциите, или грешки при изпълнение на кода (например деление на нула). За всяка грешка трябва да се погрижите да се изпише подходящо съобщение.

Обърнете внимание при декларирането и оценяване на функциите е възможно да се получи рекурсия която трябва да поддържате.

Нека разгледаме няколко примера:

- Извикване на функцията за събиране на две числа:
  ```
  add(3, 7)
  > 10
  ```
- Вложено извикване на функцията за събиране:
  ```
  add(add(3, 5), add(10, 10))
  > 28
  ```
- Деклариране и извикване на функция без аргументи, която винаги връща константа:
  ```
  myConst <- 7
  >
  myConst()
  > 7
  ```
- Деклариране и извикване на функция с аргументи:
  ```
  doubleArg <- add(#0, #0)
  >
  doubleArg(5)
  > 10
  ```
- Деклариране и извикване на по сложна функция с аргументи:
  ```
  sumSqr <- add(mul(#0, #0), mul(#1, #1))
  >
  sumSqr(5, 10)
  > 125
  ```
- Използване на условната функция:
  ```
  if(1, 7, 11)
  > 7
  if(0, 7, 11)
  > 11
  if(add(5, -5), 3, 5)
  > 5
  ```
- Използване на списъци:
  ```
  myList <- list(1, 2, 3, 4)
  >
  myList
  > [1, 2, 3, 4]
  double <- mul(#0, #0)
  >
  map(double, myList)
  > [1, 4, 9, 16]
  concat(list(1, 2), list(3, 4))
  ```

```
> [1, 2, 3, 4]
```

- Рекурсия:

```
fact <- if(eq(#0, 0), 1, mul(#0, fact(sub(#0, 1))))
>
fact(0)
> 1
fact(5)
> 120
```