Софийски университет „Св. Климент Охридски“
Факултет по математика и информатика

Теми за проекти

курсове *Структури от данни;*
Структури от данни и програмиране

специалности *Информатика, Информационни системи и*
Компютърни науки (1-ви поток)

зимен семестър 2021/22 г.

Редакции

(все още няма редакции по условието)

Обща информация

Преди да преминете към решаване на проектите, моля да прочетете раздела “Критерии за оценка на работите” в документа “Схема за оценяване”.

Всяка от темите за проект предполага да се отдели време за проучване на темата, а също и за планиране на организацията/архитектурата на решението.

Тема 1: LZW компресиране

В този проект трябва да реализирате програма, която архивира и компресира файлове.

Програмата трябва да получава параметрите си от командния ред. Тя трябва да може да архивира и компресира множество файлове и директории в един файл, използвайки LZW алгоритъма (LZ78 или негов производен). Трябва да се поддържат следните функционалности:

1. ZIP: Създаване на архив по подаден списък от файлове и/или директории. Допустимо е използването на символите * и ? в имената за по-лесно посочване на множество файлове. При работа с директории те се архивират в дълбочина (рекурсивно).
2. UNZIP: Разархивиране на един или всички файлове от архива по посочено име и директория в която да се записва резултата. Допустимо е използване на * и ? в имената за по-лесно посочване на множество файлове. При разархивиране трябва да се запази структурата на директориите.
3. INFO: Извеждане на списък на компресираните файлове с информация за нивото на компресия.
4. REFRESH: Обновяване на вече архивиран файл с друга негова версия. Изисква се тази операция да е по-бърза от създаването на целия архив наново.
5. EC: Проверка за повреден файл в архив – проверява дали не е възникнала промяна в архивния файл, която води до това, че един или повече от архивираните файлове не могат да се възпроизведат коректно при разархивиране. За целта включете някакъв вид шумозащитно кодиране.

Уточнения:

- Операциите UNZIP и REFRESH трябва да позволяват да се работи с отделни файлове в архива. Не се считат за коректни решения, при които за разархивиране или промяна на един отделен файл, трябва да се обработят и всички останали в архива.
- При посочване на имената на файлове символ * замества нула или повече произволни символи, а символ ? - точно един произволен символ.
- Възможно е в дървото, което сканирате, да срещнете празна директория (такава, в която няма никакви файлове). Трябва да намерите начин да запазите информация за нея в архива и след това да я възстановите.
- Когато архивирате дадена директория, нейното съдържание трябва да се запази с относителни пътища, които след това да се възпроизведат в директорията, в която тя се разархивира. Например:

Да допуснем, че архивираме директорията `C:\Users\John Smith\Documents`. В нея се съдържат следните:

```
C:\Users\John Smith\Documents\Homework.docx
C:\Users\John Smith\Documents\music\song.mp3
C:\Users\John Smith\Documents\video\video.mp4
```

Ако разархивираме получения файл в директория D:\Extracted, трябва да получим следните:

```
D:\Extracted\Homework.docx
D:\Extracted\music\song.mp3
D:\Extracted\video\video.mp4
```

Тема 2: Файлова система върху единичен файл

Напишете програма, която симулира файлова система върху един двоичен файл (контейнер). Структурата на контейнера е по ваша преценка. В имената на файловете и директориите във файловата система не правете разлика между малки и главни букви. Директориите могат да се влагат с произволна дълбочина. Не е задължително контейнерът да може да се разширява неограничено много. Достатъчно е при създаването му да се окаже максимален размер.

Операции. Трябва да поддържате следните функционалности:

- създаване на директория (mkdir)
- изтриване на празна директория (rmdir)
- извеждане на съдържанието на директория (ls)
- промяна на текущата директория (cd)
- копиране на файл (cp)
- изтриване на файл (rm)
- извеждане на съдържанието на файл на екрана (cat)
- записване на съдържание към файл (write). Ако не е подадена допълнителната опция +append се създава празен файл (ако такъв съществува се презаписва). Ако такава опция е подадена и се записва върху съществуващ файл, тогава съдържанието се добавя в края му. Съдържанието се подава като текст ограден в кавички - параметър на командата.
- копиране на файл от външна файлова система във вашата (import). Тук подавате път към реален файл (source) и към файл от вашата файлова система (destination). Трябва да се поддържа параметър +append, с действие подобно това при командата write.
- копиране на файл от вашата файлова система на външна (export)

Deduplication. Файловата система трябва да разбива всеки от съхранените в нея файлове на един или повече chunk-ове с размер N. Този размер се подава от потребителя при създаването на контейнера и не може да се променя след това. Реализирайте схема за премахване на дублиране (deduplication), която гарантира, че ако даден chunk се използва в повече от един файл, той се пази само веднъж във файловата система.

Resiliency. Реализирайте схема за устойчивост (resiliency). Файловата система трябва да съхранява в себе си подходяща информация, с която да може да провери дали даден chunk не е бил повреден (например при възникване на лош сектор на диска) след неговото записване. Схемата трябва да работи прозрачно за програмиста. Тоест при всеки запис/четене на файл, resiliency информацията трябва да се обновява/проверява. Ако бъде открита повреда в даден chunk, текущата операция трябва да приключи с грешка.

Второ нещо, на което трябва да обърнете внимание е какво би се случило, ако изпълнението на кода ви спре по средата на дадена операция. **Пример:** ако копирате

голям файл и програмата бъде спряна по средата на тази операция, как можете да гарантирате, че контейнерът ще може да продължи да се използва? Разбираемо е, че файлът, който е бил в процес на създаване не е валиден и не може да се очаква с него да може да се работи. Но останалото съдържание не трябва да се загуби и не е допустимо контейнерът да стане неизползваем и да се загуби цялата информация в него.

API. Цялата функционалност трябва да бъде налична през подходящо API, което да позволява да изпълните всяка една от горе-описаните операции с даден контейнер. Например да може да се създаде и/или отвори контейнер, след това в него да се добавят и премахват файлове, да се добавят и премахват директории и т.н. Трябва да обработвате всякакви проблеми и грешки.

Interpreter. Напишете прост команден интерпретатор за работа с тази файлова система. Той трябва да използва API-то и не бива да го заобикаля и да работи директно с контейнера. В този интерпретатор поддържайте текуща директория и съответно работете както с абсолютни, така и с относителни пътища. При стартиране на интерпретатора вашата програма трябва да получава параметър от командния ред – името на контейнера, с който ще работите. Ако такъв не съществува трябва да създадете нова файлова система. В този случай трябва да попитате потребителя за параметрите на контейнера – например размер на chunk, максимален размер (ако контейнерът не може да се разраства неограничено) и т.н.

Важно: Вашата файлова система може да стане по-голяма от количеството оперативна памет. Въпреки това, опитайте да използвате оперативната памет, за да може операциите да бъдат максимално бързи. Решения, които съхраняват изцяло съдържанието на файловата ви система в паметта, не са коректни. Не са коректни и такива, които работят изцяло върху файла, без да съхраняват никаква информация в паметта, чрез която да реализират операциите по-бързо.

Тема 3: Синхронизация на директории

В рамките на този проект трябва да разработите приложение, което сравнява и синхронизира две директории.

Приложението ви ще получи като вход пътищата до две директории. След това то трябва да ги сканира, да анализира получените резултати и да предложи на потребителя как да ги уеднакви, като го информира какви операции ще бъдат извършени. Накрая то трябва да може да извърши съответните операции.

Информацията за това какво трябва да се направи ще получите като аргументи от командния ред. Възможно е или потребителят да подаде цялата информация като набор от аргументи, или да укаже път към конфигурационен файл. Във втория случай цялата конфигурация ще е описана във файла и трябва да се прочете от него. Сами можете да изберете какъв да бъде форматът на входния файл (ini, xml или друг), как да е структуриран и т.н. Когато реализирате приложението, направете го така, че лесно да могат да се добавят и премахват различни файлови формати.

Като минимум в списъка с опции ще бъде указано кои са директориите, които трябва да се синхронизират и как да стане това. Възможно е да се укажат и други опции (например дали да се генерира лог файл, къде да се сложи той и т.н.).

За улеснение, ще считаме, че в двете директории няма "връзки" (soft links, hard links, junctions) или специални файлове. Може да считате, че съдържанието на всяка от двете директории образува дърво. В тях не може да има цикли, нито възли с повече от един родител.

Възможно е в директориите да има файлове, до които приложението ви не може да получи достъп. Например може да има файл, който е отворен в друго приложение; файл, за който нямате права за достъп и т.н. Приложението ви трябва да може да открива кога няма достъп до даден файл или директория и да предприема подходящи действия (например може да пропуснете такива файлове/директории, а в края на работата да изведете съобщение, в което да укажете какво не е било обработено).

Приложението ви трябва да поддържа следните видове синхронизация:

- mirror – R трябва да стане точно копие на L.
- safe – Никакви файлове не трябва да се изтриват или променят. Ако даден файл липсва в една от двете директории, но е наличен в другата, той се копира в съответната посока. Ако има файлове с еднакви пътища, но различно съдържание, дата и т.н., те не се променят по никакъв начин, а вместо това трябва да уведомите за тях потребителят (например като изведете подходящо съобщение в лог файл).
- standard – Никакви файлове не трябва да се изтриват. Файловете, които съществуват в една от директориите и липсват в другата, се копират в съответната

посока. Ако даден файл съществува и в двете, но е с различно съдържание, трябва да запазите това копие, което е с по-нова дата. Другото копие трябва да се презапише с тази, по-нова версия.

При обхождането трябва да може да извличате информация за всеки от намерените файлове (например размер, дата на последна промяна и т.н.). Програмата ви трябва да може да сравнява информацията за двете дървета и да открива какви са разликите между тях. Например:

- Възможно е да има файлове и/или директории, които се намират в едното дърво, но не и в другото;
- Възможно е да има файлове, които се намират и в двете дървета, но не съвпадат по своите дата, размер, съдържание;
- Възможно е да има файл, който се среща в двете дървета, но името или мястото му е сменено (преименуван е);
- и т.н.

Потребителят може да укаже как да се извършва сравнението на файловете. За целта добавете подходяща опция в конфигурационния файл.

- Бърз вариант (quick). Това е такова сравнение, при което се отчитат само атрибутите на файла. Ако два файла съвпадат по дата, размер и път, считаме, че са еднакви.
- Безопасен вариант (safe), при който всички файлове с еднакви пътища, дата и размер, трябва да се сравнят и байт по байт.

След като приложението ви приключи с анализа и открие разликите между двете директории, то трябва да даде информация на потребителя за това какви стъпки са необходими, за може те да се синхронизират. За целта трябва да се създаде текстов файл, в който са описани всички нужни операции. Възможни операции са например:

- копиране на файл от едната към другата директория;
- изтриване на файл;
- преименуване на файл;
- преместване на файл;
- и други.

По-долу е даден пример за това как би могъл да изглежда изходният файл с операции за дадено сканиране. Първите два реда в него указват кои са директориите, които се обработват. Всеки ред указва една операция, която трябва да се извърши. В отделните редове, всеки път започва с L или R. L указва лявата директория, а R – дясната. (Ако предпочитате, използвайте пълните думи LEFT и RIGHT вместо L и R) Символът # указва началото на коментар – текстът след него се игнорира. Можете да използвате коментари, за да поясните отделните операции. Този формат не е задължителен. Можете да го промените и/или разширите, ако решите, че е необходимо.

```
LEFT is C:\Temp
RIGHT is C:\Directory\SubDirectory
COPY L\1.txt R\1.txt   # Left copy is newer
CREATE-DIR R\Z   # Missing on the right
COPY L\Z\2.txt R\Z\2.txt   # Missing on the right
DELETE R\SomeDir\3.txt
DELETE R\SomeDir
```

След като генерира файла с операции, програмата прекратява своето действие. След това потребителят трябва да може да го прегледа и да прецени дали иска да изпълни операциите или не. Възможно е също така да го редактира, като промени или изтрие отделните редове. Така например, ако той реши, че не иска да копира файловете от поддиректорията Z и освен това иска да запази файла 3.txt, потребителят може да промени файла по следния начин:

```
LEFT is C:\Temp
RIGHT is C:\Directory\SubDirectory
COPY L\1.txt R\1.txt # Left copy is newer
CREATE-DIR L\SomeDir
COPY R\SomeDir\3.txt L\SomeDir\3.txt
```

Забележете, че редът на операциите има значение. Например, за да копираме файла 3.txt, най-напред трябва да създадем директорията SomeDir. Ако разменим редът на тези операции или пропуснем създаването на директория, синхронизацията няма да бъде успешна.

След като приключи с преглеждането на файла, потребителят може да изпълни така зададените операции. Обърнете внимание, че е възможно между изпълнението на двете фази, съдържанието на двете директории да се промени. Например правата за достъп до даден файл може да се променят, потребителят може да изтрие или премести даден файл и т.н. Сами преценете как да адресирате този проблем.

Сами можете да изберете дали да направите двете фази в едно приложение или да създадете две различни -- едно, което анализира и второ, което синхронизира, как да го направите като интерфейс и т.н.

В този проект може да използвате наготово:

- STL
- Filesystem библиотеката (налична от C++17)
- Библиотека за логване (например Boost::Log)
- Библиотека за обработка на аргументите от команден ред (например TCLAP)

Тема 4: Система за управление на бази от данни

Реализирайте проста СУБД, която поддържа работа с множество таблици. Всяка от тях се състои от произволен брой колони, всяка от които може да е от тип число, дата, или символен низ. Всяка колона има име – символен низ. Цялата информация за вашата СУБД трябва да се съхранява на диска във формат, който трябва да проектирате и опишете като част от решението.

При работа с тази СУБД трябва да можете да използвате 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; Трябва да има възможност за задаване на **стойности по подразбиране** или **автоматично-генерирани стойности**. Също за някоя колона може да указвате да бъде **създаден индекс** с цел по-бързо търсене в нея.
    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, заемано пространство и др.) 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. Трябва да можете да сглобявате по-сложни изрази чрез логически операции (**AND, NOT** и **OR**) и скоби. Също така трябва да можете да задавате подредба на резултатните редове чрез ***ORDER BY*** и да премахвате повторения, чрез ***DISTINCT.*** Не се изисква поддържане 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.

Трябва да имате възможност да работите със системата през прост конзолен интерфейс. Трябва да прихващате всякакви грешки по време на работа. Също при стартиране на програмата или чрез специална команда трябва да проверявате дали данните, които сте

съхранили не са повредени по някакъв начин (например заради външна намеса или лош сектор на диска).

Вашата база данни може да стане по-голяма от количеството оперативна памет. Въпреки това, опитайте да използвате оперативната памет за да може операциите да бъдат максимално бързи. Така решения, които съхраняват изцяло съдържанието на базата в паметта, не са коректни, както и такива, които работят изцяло върху диска, без да съхраняват никаква информация в паметта, чрез която да реализират операциите по-бързо.

Примерна употреба:

```
FMISql> CreateTable Sample (ID:Int, Name:String, Value:Int) Index ON ID
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, Indexed; 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
```

Тема 5: Интерпретатор за функционалния език **ListFunc**

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

Нека разгледаме как се дефинира езикът **ListFunc**. В този език има два типа литерали – реални числа <real-number> и списъци <list-literal>. 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 списък от аргументи на функцията, заграден в кръгли скоби. Може да има функции без аргументи (с празен списък с аргументи).

```
<real-number> ::= <всички валидни записи на double>
<list-literal> ::= [ [<expression0>, <expression1>, ...] ] *
<expression> ::= <list-literal> | <real-number> | <function-call>
<function-name> ::= <валиден идентификатор в C++>
<function-call> ::= <function-name>([<expression>, ...])
* външните скоби са литерали, вътрешните показват опционални елементи
```

Вашият интерпретатор трябва да предостави следните вградени функции

- **eq**(#0, #1) връща булевата оценка на #0 == #1
  - две числа са равни когато стойностите им са равни
  - два списъка са равни когато съответните им елементи са равни
  - число и списък са равни ако списъка има 1 елемент и той е равен на числото
- **le**(#0, #1) връща булевата оценка на #0 < #1.
- **nand**(#0, #1) връща булевата оценка на `!#0 || !#1.`
- **length**(#0) връща броя елементи в подадения списък или -1 ако аргумента е число.
- **head**(#0) връща първия елемент на списък.
- **tail**(#0) връща списък от всички без първия елемент на входния списък.
- **list**(#0) връща безкраен списък с начален елемент #0 и стъпка 1.
- **list**(#0, #1) връща безкраен списък с начален елемент #0 и стъпка #1.
- **list**(#0, #1, #2) връща списък с начален елемент #0, стъпка #1, и брой елементи #2.
- **concat**(#0, #1) връща списък, конкатенация на двата аргумента, които са списъци.
- **if**(#0, #1, #2) оценява #0 като булева стойност и връща #1 ако оценката е `true` или #2 ако оценката е `false.`
- **read**() връща число, прочетено от стандартния вход.

- **write**(#0) записва #0 на стандартния изход и връща 0 при успех и различно от 0 число при провал.
- **int**(#0) връща #0 без дробната част.

Някои функции като **if** и **nand** не е нужно да оценява всичките си аргументи за да върнат верен резултат.

Добавете и следните аритметични функции върху числа: **add**, **sub**, **mul**, **div**, **mod, sqrt.** Функцията **mod** е валидна само върху цели числа.

Освен това езикът трябва 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>
```

Декларацията на функция декларира нова или подменя стара декларация, връща 0 ако е нова декларация, връща 1 ако декларацията подменя стара декларация.

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

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

По желание може да поддържате коментари, и възможност за разделяне на сложни изрази с разстояние, табулация и нов ред (както е в последния пример).

Нека разгледаме няколко примера, редовете започващи с > са изход на интерпретатора:

```
32
> 32
[1 2 3]
> [1 2 3]
myList -> [3 4 5 7 9 10]
> 0
myList()
> [3 4 5 7 9 10]
head(myList())
> 3
tail(myList())
```

```
> [5 4 7 9 10]
isOdd -> eq(mod(int(#0), 2), 1)
isEven -> nand(isOdd(#0), 1)
> 0

list(1, 1, 10)
> [1 2 3 4 5 6 7 8 9 10]
list(5, -0.5, 3)
> [5 4.5 4]
list(read(), read(), read())
> read(): 12
> read(): -3
> read(): 4
[12 9 6 3]

list(1)
[1 2 3 4 5 6 7 8 ....
```

Програма която намира/принтира прости числа

```
not -> nand(#0, 1)
> 0
and -> not(nand(#0, #1))
> 0

// генерира всички делители които да се проверят за числото #0
divisors -> concat(
    [2],
    list(3, 2,
       add(1, int(sqrt(#0)))
    )
)
> 0

// проверява дали числото #1 има делител в списъка #0
containsDevisors -> if(
    length(#0),
    if(mod(#1,head(#0)), 0, containsDevisors(tail(#0), #1)),
    0
)
> 0

// проверява дали числото #0 е просто
```

```
isPrime -> not(containsDevisors(divisors(#0), #0))
> 0

// #0 list of numbers, leaves only primes
filterPrimes -> if(
    isPrime(#0),
    concat([head(#0)], filterPrimes(tail(#0))),
    filterPrimes(tail(#0))
)
> 0

// всички прости числа до 10
primes10 -> filterPrimes(concat([2], list(3, 1, 7)))
> 0
primes10()
> [2 3 5 7]

// prints primes for
allPrimes -> filterPrimes(concat([2], list(3)))
> 0
allPrimes()
> [2 3 5 7 11 13 17 19 ... // принтира прости числа до "безкрай"
```

Тема 6: Система за работа с електронни таблици

Имаме правоъгълна таблица, състояща се от клетки. Всяка клетка е или празна, или съдържа текст, или *израз*, по който се изчислява нейната *стойност*. Клетките се адресират чрез номер на реда и колоната, с израз от вида R*x*C*y* , където *x* и *y* задават

съответно реда и колоната на адресираната клетка. Адресирането на клетките може да е абсолютно и релативно. При абсолютното адресиране, редовете и колоните се броят от горния ляв ъгъл на таблицата. При релативното адресиране, редовете и колоните се броят от клетката, в която се среща изразът с релативно адресиране. Релативно адресиране се задава, когато в израз R*x*C*y*, номерата на ред или колона се заграждат с квадратни (средни) скоби.

Примери за адресиране:

R5C3 - абсолютно адресиране на третата клетка от 5-я ред

R[-1]C[0] - релативно адресиране на клетката над текущата (т.е. един ред нагоре и същата колона).

Текстовите клетки съдържат произволен текст, заграден в кавички.

Изразите в клетките представляват аритметични изрази с оператори +,-,*,/ и аргументи цели числа или адреси на други клетки (в абсолютен или релативен вид). Изразът във всяка клетка задава нейната стойност и се преизчислява тогава, когато някоя от клетките, адресирани в него промени стойността си. Във формулите можете да използвате и следните функции: if, and, not, or, както и булевите оператори ==, !=, <, > с естествената им семантика, семантиката на if е като на оператора ?: в C++. Изразите могат да са произволно сложност, като се използват скоби и семантиката е като в C++.

Трябва да поддържате функции sum и count с два аргумента - адреси на клетки. Тези адреси определят ъглите на правоъгълна област от таблицата, върху която вашите функции трябва да оперират. Функцията sum намира сумата от стойностите на тези клетки, докато count намира броя на непразните клетки. Стойностите на празните клетки в този случай приемаме за 0. Ако съдържанието на клетка с текст може да бъде интерпретирано като число (цяло или дробно), то това е нейната стойност. Иначе стойността е 0.

Напишете система, която поддържа такава таблица, като изпълнява следните команди (въвеждани от потребителя в интерактивен режим).

SET *адрес израз*	променя стойността на клетката със съответния адрес
PRINT VAL *адрес*	отпечатва стойността на клетката със съответния адрес
PRINT EXPR *адрес*	отпечатва формулата в клетката със съответния адрес

PRINT VAL ALL	отпечатва форматирано всички клетки в таблицата, като клетките от един ред се печатат на един ред на екрана. (*)Ако ширината на екрана не стига за всички клетки от някой от редовете, таблицата се разделя на подтаблици и те се отпечатват една по една. Всяка подтаблица трябва да съдържа няколко съседни колони от таблицата. Подтаблиците се отпечатват по ред на номерата на най-лявата им колона.
PRINT EXPR ALL	същото като PRINT VAR ALL, но за изразите в клетките
SAVE файл	записва съдържанието на таблицата (като формули) във файл във формат . csv
LOAD файл	чете съдържанието на таблицата (като формули) от файл във формат . csv
++ адрес	ако изразът на съответния адрес представлява цяло число го увеличава с единица
-- адрес	ако изразът на съответния адрес представлява цяло число го намалява с единица

При изпълнението на командите трябва да се проверява дали всички въведени адреси са в рамките на таблицата.

Таблицата се записва в текстовия формат . csv , като на един ред от файла се записват изразите от клетките на един ред на таблицата разделени с точка и запетая (;). Редовете от таблицата се записват последователно.
Имайте предвид, че таблицата може да е доста голяма - например да има стойност в клетка R100000C100000, но повечето от останалите клетки да са празни. Помислете как да се справите с тази особеност.