Софийски университет „Климент Охридски“
Факултет по математика и информатика

ПРОЕКТ ПО ФУНКЦИОНАЛНО ПРОГРАМИРАНЕ

зимен семестър 2024/2025

Проект
ФАЙЛОВА СИСТЕМА

Изготвил: Стоян Стоянов Иванов
Специалност: Информатика, 3 курс
Факултетен номер: 9MI0400132

Ръководител на курса:
доц. д-р Трифон Трифонов

януари 2025 г.

СЪДЪРЖАНИЕ

1. Увод:

Основната цел на проекта е създаването на програма, която симулира файлова система, реализирана на Haskell. Тази система предлага функционалности за създаване, управление, навигиране и визуализиране на файлове и папки в йерархична структура. Допълнително, програмата интегрира персонализиран анализатор (parser) за обработка на потребителски команди, предоставяйки интуитивно взаимодействие, наподобяващо Shell интерфейс, за ефективно управление на файловата система.

2. Функционалност

2.1. Дефиниране на файловата система:

Може да бъде един от следните два варианта:

- **File String String**: Представлява файл, който съдържа две текстови стойности. Първата е низ с името на файла (напр. "readme.md"), а втората - низ със съдържанието на файла (напр. "This is a basic file system structure.").
- **Root String [FileSystem]**: Представлява директория/папка с две части - низ с името на директорията (напр. "projects") и [FileSystem], който е списък от поддиректории или файлове вътре в тази директория.

Чрез deriving(Show) се позволява автоматично генериране на текстово представяне на данните.

2.2. Реализация на stack[1]:

- **headStack**: Връща елемента на върха на stack, ако има такъв. Входът е stack от произволен тип [a]. Резултатът е Maybe a, което означава: Just x, ако stack не е празен, и Nothing, ако е празен.

Пр. headStack [1, 2, 3] ---> Just 1

- **pushStack**: Добавя нов елемент към върха на stack. Входът са елемент и stack. Резултатът е нов stack с елемента, добавен накрая.

Пр. pushStack 4 [1, 2, 3] ---> [1, 2, 3, 4]

- **popStack**: Премахва елемента на върха на stack. Входът е stack. Резултатът е нов stack без последния елемент.

Пр. popStack [1, 2, 3] ---> [1, 2]

- **topStack**: Връща последния елемент в stack. Входът е stack. Резултатът е елементът на върха.

Пр. topStack [1, 2, 3] ---> 3

Бележка: Функциите за head, push, pop и top са взаимствани от упражнения по Логическо програмиране, проведени през летен семестър 2022/2023. Преименувани са, за да няма препокриване с Prelude модула. Идеята за използване на стек е взета от тук.

2.3. Валидация:

- **isNameFolder**: Проверява дали даденото име съвпада с името на директория/папка. Първият аргумент е името, което се проверява. Вторият аргумент е файловата система. Резултатът е True, ако съответното име съвпада с името на папката, или False в противен случай.

---

[1] *абстрактен тип данни, който съдържа подредена, линейна последователност от елементи*

Пр. isNameFolder "projects" (Root "projects" []) ---> True

Пр. isNameFolder "documents" (File "resume.pdf" "My resume") ---> False

- **isNameFile**: Проверява дали даденото име съвпада с името на файл. Първият аргумент е името, което се проверява. Вторият аргумент е файловата система. Резултатът е True, ако съответното име съвпада с името на файла, или False в противен случай.

Пр. isNameFile "resume.pdf" (File "resume.pdf" "My resume") ---> True

Пр. isNameFile "welcome.txt" (Root "archives" []) ---> False

- **isFilePath**: Проверява дали даден низ представлява път. Аргументът е низ, който трябва да се провери. Резултатът е True, ако низът започва със /, или False в противен случай.

Пр. isFilePath "/home/user/documents" ---> Резултат: True

- **isValidName**: Проверява дали дадено име, на файл или папка, е валидно в контекста на текущата файловата система. Първият аргумент е функция (напр. isNameFile или isNameFolder), която проверява дали дадено име съществува. Вторият аргумент е името за проверка. Третият аргумент е файловата система. Резултатът е True, ако името не съвпада с името на текущата папка и не се среща сред елементите в нея. Ако името на текущата папка е равно на входното име, функцията връща False - името не е валидно, тъй като вече съществува. За останалите елементи в директорията се използва foldr от библиотеката на Haskell - входната функция проверява дали даден елемент съответства на входното име.

Пр. isValidName isNameFile "newfile.txt" (Root "documents" [File "resume.pdf" "My resume"]) ---> True т.е. файлът newfile.txt не съществува

2.4. Анализатор (Parser)[2]

- **Дефиниция:** Това е нов тип данни Parser, който е обвивка около функция с тип: String -> Maybe (String, prs). Ако разборът е успешен, връща Just (remainingString, parsedResult), където remainingString е низът, който остава след като е извършен разборът, а parsedResult е резултатът от разбора. Ако анализът не успее, връща Nothing. Идеята и имплементацията е взаимствана от тук и тук.

- **Functor Parser**: Позволява прилагане на функция към резултата от анализатора, като се запази структурата му. fmap прилага функция F към резултата от анализатора P. След като е извлечен резултатът X, прилагаме F към X и връщаме нов анализатор P'. Линк към Data.Functor

Пр.
```
uppercaseParser :: Parser String
uppercaseParser = fmap (map toUpper) wordParser
```

- **Applicative Parser**: Позволява композиция на анализатори. Това е важно за комбиниране на анализатори, които правят разбор на различни части от входния низ. pure създава анализатор, който винаги връща дадена стойност X, без да се променя входния низ. (<*>) позволява прилагане

[2] Parser – в този документ да се разбира като „анализатор“, а „parsing“ - „разбор“, „анализ“, като най-близък превод на български език в този контекст.

на анализатор, който връща функция, върху друг анализатор, който връща аргумент. Левият анализатор анализира функция от тип a -> b. Десният анализатор анализира аргумент от тип a. Резултатът е нов анализатор, който връща резултат от тип b. Линк към Control.Applicative и Stackoverflow.

Пр. Анализатор за събиране на две числа:

```
addParser :: Parser (Int -> Int -> Int)
addParser = pure (+)
```

Използване на анализатора за събиране с аргументи:

```
sumParser :: Parser Int
sumParser = (+) <$> addParser <*> intParser <*> intParser
```

- **Alternative Parser**: Предоставя възможност за избор между два анализатора. Ако първият не успее, може да се опита вторият. empty е функция, която никога не успява и връща Nothing. (<|>) позволява да се комбинират два анализатора така, че ако единият не успее, да се опита вторият. Ако и той не успее, резултатът ще бъде Nothing. Линк към Stackoverflow.

Пр. Анализатор, който се опитва да направи разбор на числа, ако не успее, преминава към буквени низове.

```
numberOrWordParser :: Parser String
numberOrWordParser = many digit <|> many letter
```

2.5. Индивидуални анализатори:

- **parserForChar**: Приема един символ и връща анализатор, който ще обработи входен низ и ще се опита да намери този символ. Функцията foo приема списък и проверява дали първият символ е равен на търсения символ x. Ако съвпада, се връща Just (ys, x), което означава, че е намерен символа и се връща останалата част от низа. Ако символите не съвпадат, се връща Nothing.
- **parserForString**: Използва функцията traverse от библиотеката на Haskell, за да приложи parserForChar върху всеки символ в низа. Приема низ от символи и връща анализатор, който търсения низ от входния.
- **parserForMany**: Функцията приема функция, която проверява дали даден символ е валиден, и връща анализатор, който извлича всички символи от входния низ, които съответстват на условието на подадената функция. Вътре в анализатора е използвана функцията span от стандартната библиотека на Haskell. Тя взема входния низ и го разделя на две части - първата част съдържа всички символи, които отговарят на условието на входната функция, а втората съдържа останалите символи от входния низ. След това се връща резултатът като низ, което означава, че са изчетени всички символи, които съвпадат с условието, и връща останалата част от низа.
- **parserForWhiteSpace**: Използва isSpace - стандартна функция в Haskell, която проверява дали даден символ е интервал (напр. ' ', '\t', '\n' и т.н.). Резултатът е низ, съдържащ всички последователни интервални символи от входния низ, а остатъкът от низа се връща за по-нататъшна обработка.

➢ **parserForSlash**: Резултатът е низ, който започва след първата наклонена черта и съдържа всички символи до следващата наклонена черта или края на входния низ. Операторът *> от Control.Applicative означава: изпълни първия разбор, но запази резултата само от втория. В този случай се игнорира резултатът от parserForChar, извикан със символа за наклонена черта, и се преминава към следващата част, която извлича всички символи, които не са наклонена черта, от остатъка на входния низ.

➢ **getNextDirectory**: Тази функция използва parserForSlash, за да извлече следващата директория от низ, който изглежда като път.

Пр. /home/user/docs [1]–> Just ("/user/docs", "home") [2]–> Just ("/docs", "user") [3]–> Just ("", "docs") [4]–> Just ("", "")

➢ **parserForSET**, където SET = {cd, pwd, ls, show, quit, cat, rm, mk}. Този код дефинира анализатор, който проверява дали входният низ започва с точно думата SET.

➢ **parserForWord**: Тази функция създава анализатор, който извлича дума от входния низ до първия интервал и премахва последващите интервали. Операторът <* е предназначен да изпълни първия разбор и да върне неговия резултат, но също така да изпълни втория без да връща неговия резултат, т.е. премахва всички последващи интервали, без да ги добавя към резултата.

Пр. "hello world" ---> Just ("world", "hello")

➢ **parserForEndFile**: Извлича текст от входния низ, който завършва със символа ~, и премахва последващите интервали.

Пр. "file content ~ next" ---> Just ("next", "file content")

➢ **parserForCommand**: Това е комбиниран анализатор, който може да разпознава различни команди. Използва оператора <|>, който означава "или". Ако първият анализатор се провали, се пробва следващият, и така нататък.

➢ **parseCommand**: Тази функция използва parserForCommand, за да разпознава команда от входния низ, и премахва всички последващи интервали.

Пр. "cd /home/user" ---> Just ("/home/user", "cd")

2.6. Функции за извличане на данни:

➢ **getFile**: Намира файл с дадено име в списък от файлове и папки. Използва filter от библиотеката на Haskell с условие isNameFile, което връща списък с всички файлове, чието име съвпада с даденото.

➢ **getFolder**: Намира папка с дадено име в даден корен от файловата система. Използва filter от библиотеката на Haskell с условие isNameFolder върху децата на този корен.

➢ **getNameOfRoot**: Тази функция извлича името на корена от даден файлова система.

➢ **getFileFromRoot**: Тази функция намира файл с дадено име в децата на даден корен.

➢ **getFileByDirectory**: Тази функция намира рекурсивно файл, базирайки се на даден път в структурата на файловата система. Чрез getNextDirectory разбива пътя на текуща директория и

остатъчен път, или файл, ако пътят завършва – извиква getFile. Ако има остатъчен път - търси текущата папка чрез getFolder.

2.7. Добавяне и премахване на файлове:

- **add**: Тази функция добавя файл или папка в дадена структура на файловата система на базата на подадения път. Работи рекурсивно, като преминава през всяка част на пътя и актуализира съответната папка. Проверява дали пътят е празен чрез getNextDirectory. Ако е, добавя директно към текущия списък от елементи в корена и връща новия корен. При рекурсивния случай разделя пътя на текуща директория и остатък от пътя с помощта на getNextDirectory. Извиква switchDirectories, за да намери папката, съответстваща на текущата част от пътя. Извиква add върху намерената папка, за да добави елемента в остатъка от пътя. Ако актуализирането на папката е успешно, връща новия корен, където старият е заменен с актуализирания чрез switchRoot.

Бележка: Дооформена чрез chatGPT.

- **addFile**: Специализиран вариант на add, който добавя файл към файловата система.
- **addFolder**: Специализиран вариант на add, който добавя нова папка към файловата система.
- **removeFileFromRootHelper** и **removeFileFromRoot**: Премахва файл с дадено име от списъка с файлове в дадена директория. Преглежда всеки елемент в списъка от файлове. Ако текущият елемент е файл, сравнява текущото име на файла с името на файла, който трябва да се премахне. Ако имената съвпадат, премахва този файл от списъка. Ако името не съвпада, връща текущия елемент и продължава търсенето рекурсивно.
- **removeFileFromPathHelper** и **removeFileFromPath**: Това е рекурсивна функция, която премахва файл в дадена директория, като преминава по пътя към него. Използва getNextDirectory, за да разбие пътя. Ако пътят завършва с името на файл, извиква removeFileFromRoot, за да премахне този файл от текущата директория. Ако има остатък от пътя, извиква switchDirectories, за да намери съответната поддиректория. Рекурсивно преминава през поддиректориите, за да стигне до файла.

2.8. Спомагателни функции:

- **fancyList**: Тази функция приема списък от Maybe a и го преобразува в Maybe [a]. Идеята е да премахва всички Nothing стойности от списъка и да събира само тези елементи, които са Just.
- **filesToString**: Функцията преобразува списък от файлови системи в списък от низове, като преминава през файловата система и създава низове за всяка директория. Събира името на директорията и добавя наклонена черта пред всяко име, като рекурсивно обработва поддиректориите. Ако срещне директория, т.е. наклонена черта, започва от корена. Ако срещне други директории, добавя името им към пътя.

Пр. filesToString [Root "home" [], Root "user" []] – Резултат: ["/home", "/user"]

- **catFiles**: Функцията събира съдържанието на два файла и създава нов файл с комбинираното съдържание.
- **switchDirectories**: Тази функция търси дадена директория по име в списък с директории. Функцията използва filter от библиотеката на Haskell, за да избере само тези елементи от списъка, които съответстват на даденото име на папка. След това извиква headStack, за да вземе първата съвпаднала директория.
- **switchRootHelper** и **switchRoot**: Тази помощна функция е предназначена за актуализиране на кореновата директория, ако името на текущата директория съвпада с името на новата. Ако текущата директория има същото име като новата, просто я замества. Ако не съвпада, преминава към следващата директория.
- **printFile**: Извежда информация за файл.
- **printRoot**: Тази функция връща низ, който представя кореновата директория или файл.
- **printSystem**: Тази функция връща низ, който представя структурата на файловата система като пълен път.

2.9. Основни функции:

- **PWD**: Извежда текущия път на файловата система. Преобразува списъка с файловата система в низ с помощта на printSystem и го отпечатва на екрана. – Фигура 1
- **CD**: Променя директорията на текущата работна директория в зависимост от входа. Ако входът е празен, остава в същата директория. Ако входът е "..", преминава към родителската директория, използвайки popStack. Ако е наименована директория, премества се в нея с помощта на switchDirectories и pushStack *(Фигура 1).*

```
cd archives
pwd
/[:] /archives/[:] path:
/archives/
```

Фигура 1

- **LS**: Разделя входния низ чрез parseCommand и изпълнява съответната команда, като връща новото състояние на файловата система. Отпечатва информация за файловете в директорията *(Фигура 2).*

```
ls
/[:] file: welcome.txt
file: readme.md
root: projects
root: documents
root: archives
```

Фигура 2

- **CAT**: Използва се за събиране на съдържанието на файлове и евентуалното комбиниране на съдържанието им. Работи рекурсивно. Ако входът съдържа символа ">", съдържанието на

текущия файл и съдържанието на файла, посочен от входа, се съединяват и записват в нов файл. Ако входът не съдържа “>”, то се проверява дали става дума за пълен път до файл. Ако е така, извлича файла и комбинира съдържанието му с останалото. Ако файлът не съществува в кореновата директория или текущата директория, продължава да търси. Ако в командата има директории, ще се премине през тях и ще съедини съдържанието на файловете в тях *(Фигура 3).*

```
cat report.docx budget.xlsx > temp.fmi
ls
/documents/work/[:] /documents/work/[:] file: temp.fmi
file: report.docx
file: budget.xlsx
show temp.fmi
/documents/work/[:] file_name: temp.fmi
content:
Work report for the yearWork budget for the year
```

Фигура 3

- **RM**: Използва removeFileFromPath за премахване на даден файл или директория от файловата система, като обработва входа и актуализира състоянието на файловата система. Командата rm приема вход, който може да бъде име на файл или директория, която да бъде премахната. Входът се анализира с помощта на функцията parserForWord, за да се извлече името на файла или директорията. Ако входът е относителен път, командата използва текущото местоположение и съвпада с пътя към файла или директорията. След това, с помощта на removeFileFromPath, тя премахва файла или директорията. Ако пътят е директория, тя премахва само файловете в нея, без да променя самата директория. Ако се подадат няколко файла, командата ще ги изтрие по ред, като премахва съответните файлове или директории, използвайки рекурсивна логика. След като файловете или директориите са премахнати, се връща обновеното състояние на файловата система (*Фигура 4*).

```
rm temp.fmi report.docx
ls
/documents/work/[:] /documents/work/[:] file: budget.xlsx
```

Фигура 4

- **MK**:
  - ✓ **MKFILE**: Командата се използва за създаване на нов файл в текущата директория. Файлът има име и съдържание, което се задава чрез командата. Първо входът се обработва с функцията parserForWord, за да се извлече името на файла и съдържанието му. Съществува проверка за валидност на името на файла, която използва функцията isValidName. Тази функция проверява дали името е допустимо за файл в текущата директория. Ако името е валидно, файлът се добавя към файловата система с помощта на функцията addFile и новото състояние на файловата система се връща в обвивката на mkfile.
  - ✓ **MKDIR**: Командата създава нова директория в текущото местоположение на файловата система. Подобно на горната функция, входът се обработва чрез parserForWord, за да се получи името на директорията, която трябва да бъде създадена. Чрез функцията isValidName се проверява дали името на директорията е валидно и дали не съществува вече такава директория. Ако името на директорията е валидно, новата директория се добавя към текущото местоположение

на файловата система чрез функцията addFolder и новото състояние на файловата система се връща.

- **SHOW**: Търси файл по име и използва printFile, за да го отпечата *(Фигура 5).*

```
show report.docx
/documents/work/[:] file_name: report.docx
content:
Work report for the year
```

Фигура 5

- **RUN**: Функцията run е основната управляваща функция, която стартира интерактивния режим за работа с файловата система. Тя приема команди от потребителя, обработва ги и актуализира състоянието на файловата система в зависимост от въведените команди. В началото на всяко извикване на run, тя отпечатва текущото състояние на файловата система чрез printSystem, което показва всички коренови директории и файлове. Използва getLine, за да получи вход от потребителя. Проверява командата чрез parseCommand или IsHelper. В зависимост от въведената команда, тя предприема съответното действие: pwd, ls, show и/или quit. След изпълнението на командата, run отново се извиква, за да чака следваща команда. Това създава интерактивен цикъл, в който потребителят може да изпълнява различни файлови операции.

- **QUIT** *(Фигура 6):*

```
quit
/[:] Goodbye! :D
```

Фигура 6

***Бележка**: putStr, putStrLn и getLine*

3. Заключение

В заключение, проектът успешно реализира симулация на файлова система на езика Haskell, като осигурява функционалности за създаване, управление и навигация в йерархична структура от файлове и папки. Чрез интеграцията на персонализиран анализатор, проектът предоставя лесен и интуитивен интерфейс, който имитира командния интерфейс на операционни системи, като Shell. Възможностите за добавяне, премахване и редактиране на файлове и директории, както и за извършване на основни операции с тях, са реализирани с помощта на рекурсивни функции и ефективни алгоритми за работа с данни. Проектът е добре структуриран и предлага гъвкави решения за работа с файлови системи, които могат да бъдат полезни за приложения, изискващи манипулиране на файлови структури в Haskell.

4. Използвана литература:

https://stackoverflow.com/questions/7203686/haskell-what-is-control-applicative-alternative-good-for

https://stackoverflow.com/questions/41404647/how-to-implement-search-in-file-system-in-haskell?utm_source=chatgpt.com

https://stackoverflow.com/questions/26002415/what-does-haskells-operator-do

https://hoogle.haskell.org/?q=putStrLn

https://hoogle.haskell.org/?hoogle=span

https://hoogle.haskell.org/?hoogle=isSpace

https://hoogle.haskell.org/?hoogle=getLine

https://hackage.haskell.org/package/base-4.21.0.0/docs/Data-Traversable.html

https://hackage.haskell.org/package/base-4.21.0.0/docs/Data-Functor.html

https://hackage.haskell.org/package/base-4.21.0.0/docs/Control-Applicative.html

https://github.com/YanaRGeorgieva/Logic-programming

https://github.com/tsoding/haskell-json/commit/bafd97d96b792edd3e170525a7944b9f01de7e34

https://developer.mozilla.org/en-US/docs/Glossary/Wrapper

http://www.zvon.org/other/haskell/Outputprelude/foldr_f.html

http://www.zvon.org/other/haskell/Outputprelude/filter_f.html