Курсови проекти по Функционално програмиране

СУ-ФМИ, Специалност „Компютърни науки“, 2 курс

Зимен семестър 2013/2014

Обща информация за проектите

Курсовите ви проекти трябва да бъдат разработени на езика Scheme.

Предаване на курсовите проекти

След като приключите с работата по проекта, трябва да качите решението си в системата moodle.openfmi.net. За целта на страницата на курса ще бъде публикуван специален формуляр.

- Изходният ви код трябва да бъде оформен в един файл с разширение .scm или .rkt.
- Програмата ви трябва да може да работи на DrRacket версия 5.3.6 или по-нова.
- За език в DrRacket можете да изберете между R5RS или Pretty Big.

Важно: При предаване на курсовите си работи, изпращайте само и единствено файла съдържащ изходния код на решението. Моля, не предавайте други файлове (например .bak файлове, файлове с описание на заданието на проекта и т.н.). Файлът трябва да бъде качен директно, **без да се компресира.**

Защита и оценяване на курсовите работи

По време на защитата проектите се оценяват по няколко критерия, вкл. и следните:

- Дали предаденото решение работи коректно.
- Дали предаденото решение отговаря на всички изисквания, вкл. на тези в настоящия раздел.
- Дали кодът е добре оформен и дали е добре документиран.
- Дали решението е било тествано. (Проверява се как точно е било тествано решението и как е проверено поведението му в различни ситуации).
- Дали по време на защитата можете да внесете корекции в кода.
- Дали добре разбирате и можете да обясните как работи решението.

Всеки от вас трябва да може да отговори на различни въпроси свързани с решението, а също и да бъде в състояние да внесе малки корекции в кода на място.

Ако искате, по време на защитата можете да донесете и да работите на личния си лаптоп, вместо на компютрите в залите.

Съвместна работа / използване на чужд код

Проектите са предвидени за реализация от един човек и трябва да бъдат разработвани самостоятелно.

В рамките на проекта е допустимо да използвате код написан от някой друг (напр. готова библиотека или помощ от ваш приятел/колега), като това трябва ясно да се обяви:

1. При предаването на проекта, под формата на бележка, която ще оставите в системата
2. При защитата на проекта, като още в началото посочите коя част от проекта сте разработили самостоятелно.

Когато в даден проект се използва чужд код се оценяват и (1) способността ви за внедряване на този код във вашето решение (напр. в случаите, когато се използва външна библиотека) и (2) дали добре разбирате какво прави чуждият код.

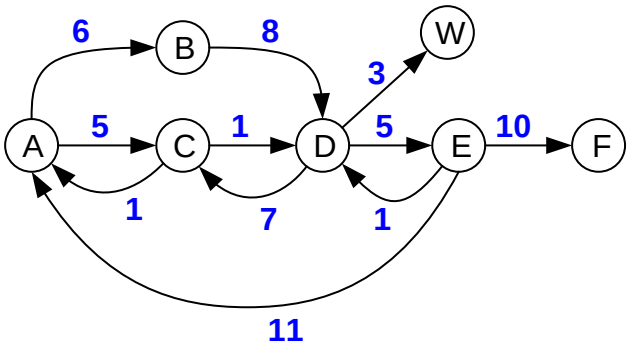
Оценката на всеки от проектите се формира от онази негова част, която е била самостоятелно разработена от вас.

В случаите на плагиатство се поставя оценка слаб 2,00 за целия курс.

1: Откриване на път в граф

Даден е насочен граф с тегла на дъгите – естествени числа. (Считаме, че естествените числа включват нулата). Възлите в графа могат да бъдат от произволен тип – числа, символи, стрингове, списъци и т.н. Графът се представя като асоциативен списък с ключове – възлите на графа. Всеки възел N е асоцииран със списък съдържащ нула, една или повече наредени двойки от вида (<възел> . <тегло>). Всяка такава асоциация представя дъга, която започва от N, завършва във <възел> и има тегло <тегло>. Например:

Граф



Представяне като списък

```
( (A (B 6) (C 5))
  (B (D 8))
  (C (A 1) (D 1))
  (D (C 7) (E 5) (W 3))
  (E (A 11) (F 10))
  (F)
  (W) )
```

Напишете функция `(paths start end G)`, която намира най-евтиния (като сума от теглата на участващите в него ребра) и най-краткия (като брой ребра) път между възлите `start` и `end`. Най-евтиният път трябва да се намери чрез алгоритъма на Дейкстра, а най-краткият – чрез алгоритъма за търсене в широчина (BFS).

- Функцията трябва да връща `#f`, ако между `start` и `end` няма път или ако някой от `start` и `end` не е възел в графа.
- Ако между двата възела съществува поне един път, функцията връща наредена двойка от двата намерени пътя (най-евтин и най-кратък). Път между два върха се представя като списък с първи елемент – число, съответстващо на теглото му, следвано от възлите, в реда, в който те се срещат в пътя (вижте дадените по-долу примери)

За определеност ще считаме, че за всеки възел X съществува тривиален път от X до X с дължина 0. Например:

```
(min-path 'A 'A G) → ( (0 A) . (0 A) )
```

Това правило важи само за възлите на графа. Ако X не е възел в графа, ще имаме:

```
(min-path 'X 'X G) → #f
```

Примери

За дадения по-горе граф ще имаме:

Израз	Резултат
`(min-path 'A 'F G)`	`( (21 A C D E F) . (21 A C D E F) )`
`(min-path 'A 'Z G)`	`#f  ;(защото в графа няма възел Z)`
`(min-path 'W 'E G)`	`#f  ;(защото в графа няма път между W и E)`
`(min-path 'E 'A G)`	`( (9 E D C A) . (11 E A) )`

2: Хъфманово кодиране

Даден е списък L, който може да съдържа произволни елементи (числа, списъци, наредени двойки, други списъци и т.н.). Честотен списък за L ще наричаме асоциативен списък, в който за всеки от елементите на L е указано колко пъти се среща той в L. Например за списъка

```
( (1 . 2) “Hello world” a (1 . 2) 5 5 5)
```

ще имаме следния честотен списък:

```
( ( (1 . 2) . 2 )
  ( "Hello world" . 1)
  ( a . 1 )
  ( 5 . 3 ) )
```

Напишете функции `(encode L compare?)` и `(decode E)`, които извършват описаните по-долу дейности.

Функцията `(encode L compare?)` получава като вход произволен списък L и го кодира с помощта на алгоритъма на Хъфман. Функцията трябва да връща списък с точно три елемента `(<frequencies> <codes> <encoded>)`, където:

- `<frequencies>` е честотен списък за L.
- `<codes>` е асоциативен списък, в който за всеки елемент на L е указано какъв е неговият код според алгоритъма на Хъфман. Кодът се представя като списък от нули и единици, например `(0 1 1 0)`.
- `<encoded>` е кодирания чрез алгоритъма на Хъфман списък L. Кодираният списък се представя като списък от нули и единици.

За да може функцията да свърши своята работа, тя трябва да може да определи дали два елемента в списъка са идентични. За тази цел тя получава аргумента `compare?`. Това е двуместен предикат, който връща `#t` ако два елемента съвпадат. Когато извършва сравнения между елементите на списъка, `compare` трябва да използва този предикат. Например бихме могли да извикаме функцията по следните начини:

```
(encode ‘(1 (2 2) 3 (2 2) 4) equal?)
(encode ‘(1 (2 2) 3 (2 2) 4) eq?)
```

Поради спецификата на `equal?` и `eq?`, в първия случай функцията ще трябва да счита, че има две срещания на елемента `(2 2)`, а във втория – че има два отделни срещания на `(2 2)`.

Функцията encode трябва да може да работи и върху празния списък. Той се кодира като:

```
( () () () )
```

Функцията `decode` получава като вход списък E, който е бил създаден от функцията `encode` и декодира оригиналното съдържание на кодирания списък. Функцията трябва да връща декодирания списък.

Пример

Нека е даден следният входен списък:

```
(a a a a b b c d a)
```

Една възможна реализация на функцията би могла да изчисли:

<frequencies>	**<codes>**	**<encoded>**

((a 5)
  (b 2)
  (c 1)
  (d 1))
	((a (1))
	  (b (0 1))
	  (c (0 0 0))
	  (d (0 0 1)))
		(1 1 1 1 0 1 0 1 0 0 0 0 0 1 1)

Респективно бихме имали:

```
(define L ‘(a a a a b b c d a) )

(encode L eq?) ; връща
                  ( ( (a 5) (b 2) (c 1) (d 1) )
                    ( (a (1))  (b (0 1)) (c (0 0 0)) (d (0 0 1)) )
                    ( 1 1 1 1 0 1 0 1 0 0 0 0 0 1 1 ) )

(define E (encode L eq?) )

(decode E) ; връща (a a a a b b c d a)
```