Zadania domowe na pracownię nr 5

W tej dwutygodniówce znów dwa zadania! Każde polega na dokończeniu implementacji innego fragmentu bardzo prostego programu do rozwiązywania sudoku. Drobnym utrudnieniem jest to, że jest to sudoku zgeneralizowane na plansze innych rozmiarów. Dokładniej:

- Dla danej liczby naturalnej $n > 0$, plansza do sudoku to siatka o $n^2 \times n^2$ polach, którą dodatkowo podzielono n^2 *pudełek*, czyli kwadratów o boku n.
- Dany jest *alfabet* czyli zbiór n^2 *znaków*.
- Zadanie polega na wpisaniu w każde pole planszy jednego znaku z alfabetu tak, by znaki nie powtarzały się w żadnej kolumnie, żadnym wierszu, ani żadnym pudełku.
- Plansza początkowa zawiera kilka znaków już ustalonych. W zależności od tego początkowego ustawienia, dana instancja może mieć 0, 1 lub więcej rozwiązań.

Przykładowo, dla $n = 2$,

sudoku

		4	
	3		
1	2		4
	4		

ma dwa rozwiązania:

2	1	4	3
4	3	2	1
1	2	3	4
3	4	1	2

oraz

2	1	4	3
4	3	1	2
1	2	3	4
3	4	2	1

Ogólna struktura programu

Program do rozwiązywania sudoku podzielony jest na trzy pliki:

- **sudoku-spec.rkt** definiuje ogólne zasady: alfabet czyli listę znaków pewnych wartości (`alphabet`), procedurę do porównywania znaków (`ch-eq?`) i relację mniejszości (`ch-lt?`). Proszę pamiętać, że znakami mogą być różne wartości, a zadaniem Rozwiązującego jest zdbać o zachowanie abstrakcji danych (w szczególności nie wolno zakładać, że znakami są zawsze liczby).

- **solution.rkt** to główny plik, który zawiera definicję planszy i procedurę generowania listy wszystkich prawidłowych rozwiązań.
- **sudoku-main.rkt** zawiera wyrażenie, które czyta z wejścia standardowego listę wierszy (każdy wiersz to lista znaków, przy czym pole puste oznaczamy symbolem `'_`) i wypisuje na ekran rozwiązanie (o ile takie istnieje).

Program możemy skompilować poleceniem

```
raco exe sudoku-main.rkt
```

A potem uruchomić np. tak:

```
./sudoku-main < example.txt
```

gdzie plik `example.txt` zawiera przykładowo następującą instancję problemu:

```
((_ _ 4 _)
 (_ 3 _ _)
 (1 2 _ 4)
 (_ 4 _ _))
```

Zadanie A (reprezentacja planszy)

W tym zadaniu należy wymyślić reprezentację i uzupełnić definicję interfejsu struktury danych przechowującej planszę do sudoku. Interfejs składa się z następujących procedur:

- `(define (rows->board xs) ... )` – Tworzy planszę z listy zawierającej kolejne wiersze (listy znaków). Np. prawidłowym użyciem dla specyfikacji

  ```
  (define alphabet '(1 2 3 4))
  (define ch-eq? =)
  (define ch-lt? <)
  ```

  jest

  ```
  (rows->board '((_ _ 4 _)
                 (_ 3 _ _)
                 (1 2 _ 4)
                 (_ 4 _ _))
  ```

- `(define (board-rows b) ... )` – Ujawnia listę kolejnych wierszy
- `(define (board-columns b) ... )` – Ujawnia listę kolejnych kolumn (każda kolumna to oczywiście lista znaków czytana od góry do dołu), np.:

```
> (board-columns (rows->board '((_ 4 _ _)
                                (1 2 _ 4)
                                (_ 3 _ _)
                                (_ _ 4 _))))
'((_ 1 _ _) (4 2 3 _) (_ _ _ 4) (_ 4 _ _))
```

- `(define (board-boxes b) ... )` – Ujawnia listę pudełek, gdzie każde pudełko to lista zawierająca wartości znajdujące się w danym pudełku. W wyniku działania tej procedury kolejność pudełek na liście i kolejność elementów w każdym pudełku mogą być dowolne. Np.:

```
> (board-boxes (rows->board '((_ 4 _ _)
                              (1 2 _ 4)
                              (_ 3 _ _)
                              (_ _ 4 _))))
'((_ 4 1 2) (_ _ _ 4) (_ 3 _ _) (_ _ 4 _))
```

Pamiętaj o zachowaniu konwencjonalnej abstrakcji danych względem specyfikacji w pliku **sudoku-spec.rkt**. Postaraj się unikać drogich operacji takich jak `append` czy `list-ref` (nazwany na wykładzie `nth`).

Zadanie B (poprawność częściowego rozwiązania)

Częściowe rozwiązanie to kilka uzupełnionych wierszy (czyli takich, które zawierają tylko znaki z alfabetu) i (być może) częściowo uzupełniony wiersz. Częściowe rozwiązanie budujemy „od dołu" i „od prawej".[1] Częściowe rozwiązanie jest *poprawne*, jeśli nie łamie zasad gry dla już wypełnionych pól (czyli tych w częściowym rozwiązaniu i planszy początkowej). Np. częściowym rozwiązaniem planszy początkowej

		4	
	3		
1	2		4
	4		

może być

		2	1
1	2	3	4
3	4	1	2

Rzeczywiście, nakładając częściowe rozwiązanie na planszę początkową widzimy, że w wypełnionych polach wartości się zgadzają (więc rozwiązanie częściowe pokrywa się z planszą początkową) i wartości nie powtarzają się w kolumnach, wierszach ani pudełkach:

[1] Myślę, że Państwo już doskonale wiedzą czemu: łatwiej rozszerzyć częściowe rozwiązanie, np. consując nowy element do wiersza „z lewej", podczas gdy dodawać element „po prawej" (czyli na końcu) wymaga kopiowania całej listy.

		4	
	3	2	1
1	2	3	4
3	4	1	2

Zadaniem Państwa jest uzupełnić definicję procedury

- `(define (partial-solution-ok? initial-board elems rows) ... )`, która sprawdza czy częściowe rozwiązanie zdefiniowane przy użyciu `elems` (częściowo uzupełnionego wiersza) i `rows` (uzupełnionych wierszy) jest poprawne przy planszy początkowej `initial-board`. Argument `elems` to lista znaków z alfabetu, a `rows` to lista list znaków. Przykładowo, poprawność częściowego rozwiązania z przykładu powyżej można sprawdzić następująco:

```
> (partial-solution-ok?
    (rows->board '((_ _ 4 _) (_ 3 _ _) (1 2 _ 4) (_ 4 _ _)))
    '(2 1)
    '((1 2 3 4) (3 4 1 2)))
#t
```

Należy zadbać o zachowanie abstrakcji danych alfabetu i planszy!

Uwagi

Rozwiązania obu zadań mogą znaleźć się oczywiście w jednym pliku. Jeśli rozwiązują Państwo tylko Zadanie A, można zwyczajnie pozostawić definicję procedury `partial-solution-ok?` taką, jaka jest w szablonie. Jeśli rozwiązują Państwo oba zadania i mają jeden plik, który rozwiązuje oba, proszę wstawić go w WebCAT-cie dwa razy, raz jako rozwiązanie zadania 5a i drugi raz jako rozwiązanie zadania 5b. Nieprzekraczalny termin: **poniedziałek, 12 kwietnia 2021r., godz. 5:30**. Proszę pamiętać o zasadach współpracy opisanych w regulaminie przedmiotu.