

Zadanie nr 1 na pracownię

Najpopularniejsze rozwiązania służące do magazynowania i przetwarzania danych – *relacyjne bazy danych* – oparte są na tak zwanym *modelu relacyjnym*. W modelu relacyjnym dane przechowywane są w *relacjach* (albo *tabelach*), którymi manipuluje się przy użyciu przystosowanego do tego języka zapytań. Celem zadania jest zaimplementowanie pewnego zestawu operacji na relacjach zaimplementowanych przy użyciu list w języku Racket.

Relacja składa się ze schematu oraz danych – *wierszy*:

```
(define-struct table (schema rows) #:transparent)
```

Schemat jest listą struktur opisujących *kolumny* relacji. Każda kolumna posiada swoją nazwę, za pomocą której można się do niej odwoływać, oraz typ, który określa, jakiego rodzaju wartości mogą być przechowywane w tej kolumnie:

```
(define-struct column-info (name type) #:transparent)
```

Nazwy kolumn będziemy zapisywać jako symbole Racketowe, natomiast ich typy – jako jeden z czterech symboli: 'number, 'string, 'symbol lub 'boolean.

Wiersze tabeli będą listami list o długości oraz typach elementów zgodnych ze schematem. Przykładowa definicja tabeli znajduje się poniżej.

```
(define cities
  (table
    (list (column-info 'city      'string)
          (column-info 'country  'string)
          (column-info 'area     'number)
          (column-info 'capital  'boolean))
    (list (list "Wroclaw" "Poland" 293 #f)
          (list "Warsaw"  "Poland" 517 #t)
          (list "Poznan"   "Poland" 262 #f)
          (list "Berlin"   "Germany" 892 #t)
          (list "Munich"   "Germany" 310 #f)
          (list "Paris"    "France" 105 #t)
          (list "Rennes"   "France" 50  #f)))))
```

A oto inna tabela:

```
(define countries
  (table
    (list (column-info 'country 'string)
          (column-info 'population 'number))
    (list (list "Poland" 38)
          (list "Germany" 83)
          (list "France" 67)
          (list "Spain" 47)))))
```

Zaimplementuj następujące operacje przetwarzające relacje:

- (table-insert row tab) – wstawianie wiersza do relacji. Procedura powinna weryfikować, czy wstawiany wiersz odpowiada schematowi relacji – tzn. czy posiada właściwą liczbę kolumn oraz czy elementy są właściwych typów. W przypadku niezgodności, procedura powinna kończyć się błędem (wywołanie error). Kolejność wierszy w wynikowej relacji jest nieistotna. Przykład:

```
> (table-rows (table-insert (list "Rzeszow" "Poland" 129
  #f) cities))
(list (list "Rzeszow" "Poland" 129 #f)
      ...)
```

- (table-project cols tab) – projekcja, czyli wybór podzbioru kolumn. Argument cols powinien zawierać listę nazw kolumn, z których powinna składać się wynikowa relacja. Kolejność wierszy powinna zostać zachowana. Przykład:

```
> (table-project '(city country) cities)
(table
  (list (column-info 'city 'string)
        (column-info 'country 'string))
  (list (list "Wroclaw" "Poland")
        ...))
```

- (table-rename col ncol tab) – zmiana nazwy kolumny. Wynikowa relacja powinna składać się z tych samych wierszy, co wejściowa, ale kolumna o nazwie col powinna zmienić nazwę na ncol. Kolejność wierszy powinna zostać zachowana. Przykład:

```
> (table-rename 'city 'name cities)
(table
 (list (column-info 'name 'string)
       ...))
(list (list "Wroclaw" "Poland" 293 #f)
      ...))
```

- `(table-sort cols tab)` – sortowanie. Wiersze relacji powinny zostać posortowane niemalejąco, w kolejności leksykograficznej, używając kolumn `cols` jako klucza. Sortowanie powinno być stabilne – tzn. kolejność kolumn o równym kluczu powinna pozostać niezmieniona. Operacja powinna działać dla kolumn o dowolnym typie. Schemat relacji powinien nie ulec zmianie. Przykład:

```
> (table-rows (table-sort '(country city) cities))
(list (list "Paris" "France" 105 #t)
      (list "Rennes" "France" 50 #f)
      (list "Berlin" "Germany" 892 #t)
      (list "Munich" "Germany" 310 #f)
      (list "Poznan" "Poland" 262 #f)
      (list "Warsaw" "Poland" 517 #t)
      (list "Wroclaw" "Poland" 293 #f))
```

- `(table-select form tab)` – selekcja. Wynikowa relacja powinna zawierać te i tylko te wiersze relacji `tab`, które spełniają warunek opisany formułą `form`. Formuły konstruowane są za pomocą następujących struktur:

```
(define-struct and-f (l r))      ; koniunkcja formuł
(define-struct or-f (l r))       ; dysjunkcja formuł
(define-struct not-f (e))        ; negacja formuły
(define-struct eq-f (name val))  ; wartość kolumny name równa
    val
(define-struct eq2-f (name name2)) ; wartości kolumn name i
    name2 równe
(define-struct lt-f (name val))  ; wartość kolumny name
    mniejsza niż val
```

Wartości pojawiające się w formułach powinny być zgodne z typem odpowiedniej kolumny w przetwarzanej relacji. Kolejność wierszy w wyniku jest nieistotna. Przykład:

```
> (table-rows (table-select (and-f (eq-f 'capital #t)
                                   (not-f (lt-f 'area
                                                300))))
                                   cities))
(list (list "Warsaw" "Poland" 517 #t)
      (list "Berlin" "Germany" 892 #t))
```

- (table-cross-join tab1 tab2) – złączenie kartezjańskie. Wynikowa tabela powinna zawierać w swoim schemacie wszystkie kolumny obydwu relacji tab1 i tab2. Wiersze wynikowej relacji powstają przez konkatencję każdego wiersza relacji tab1 z każdym wierszem relacji tab2. Jeśli wejściowe relacje mają odpowiednio n_1 i n_2 wierszy, relacja wynikowa powinna mieć $n_1 \times n_2$ wierszy. Kolejność wynikowych wierszy jest nieistotna. Przykład:

```
> (table-cross-join cities
    (table-rename 'country 'country2 countries))
(table
 (list
  (column-info 'city 'string)
  (column-info 'country 'string)
  (column-info 'area 'number)
  (column-info 'capital 'boolean)
  (column-info 'country2 'string)
  (column-info 'population 'number))
 (list (list "Wroclaw" "Poland" 293 #f "Poland" 38)
       (list "Wroclaw" "Poland" 293 #f "Germany" 83)
       (list "Wroclaw" "Poland" 293 #f "France" 67)
       (list "Wroclaw" "Poland" 293 #f "Spain" 47)
       (list "Warsaw" "Poland" 517 #t "Poland" 38)
       ...))
```

- (table-natural-join tab1 tab2) – złączenie naturalne. Wynikowa tabela powinna posiadać w schemacie wszystkie kolumny pojawiające się w tabelach tab1 i tab2, bez powtórzeń. Wynik złączenia naturalnego dwóch tabel powinien być równoważny wynikowi następującego ciągu operacji:

- Zmień nazwę tych kolumn tabeli tab2, które występują również w tab1.

- Wykonaj złączenie kartezjańskie.
- Wybierz tylko te wiersze, dla których wartości w kolumnach powtarzających się w obu tabelach są równe.
- Wykonaj projekcję, usuwając z wyniku kolumny, które zostały prze-nazwane w pierwszym punkcie.

Kolejność wynikowych wierszy jest nieistotna. Przykład:

```
> (table-natural-join cities countries)
(table
 (list
  (column-info 'city 'string)
  (column-info 'country 'string)
  (column-info 'area 'number)
  (column-info 'capital 'boolean)
  (column-info 'population 'number))
 (list (list "Wrocław" "Poland" 293 #f 38)
       (list "Warsaw" "Poland" 517 #t 38)
       (list "Poznan" "Poland" 262 #f 38)
       (list "Berlin" "Germany" 892 #t 83)
       (list "Munich" "Germany" 310 #f 83)
       (list "Paris" "France" 105 #t 67)
       (list "Rennes" "France" 50 #f 67)))
```

Ten sam rezultat (modulo kolejność wierszy i kolumn) powinien dać następujący ciąg operacji:

```
> (table-project '(city country area capital population)
  (table-select (eq2-f 'country 'country1)
    (table-cross-join cities
      (table-rename 'country 'country1
        countries))))
```

Wersja zaawansowana: Implementacja złączenia naturalnego nie musi wyliczać wszystkich wierszy złączenia kartezjańskiego (których jest dużo: tyle, co iloczyn liczby wierszy pierwszej i drugiej relacji). Efektywna implementacja może np. wstępnie posortować obie tabele, a następnie złączać kartezjańsko zbiory wierszy zgadzające się wartościami w kolumnach użytych w złączeniu.

Kolejność kolumn w wynikowych tabelach jest nieistotna. Kolejność wierszy w wynikowych tabelach jest nieistotna dla wszystkich operacji za wyjątkiem `table-sort` (specyfikującej kolejność wierszy wyjściowych) oraz `table-project` i `table-rename` (zachowujących oryginalną kolejność wierszy). Działanie operacji na tabelach posiadających powtarzające się nazwy kolumn może być dowolne. Działanie operacji źle otypowanych (np. złączenie naturalne dwóch relacji, które posiadają kolumny o tych samych nazwach, ale różnych typach) również może być dowolne.

Pisząc rozwiązanie zadania, staraj się stosować poznane na wykładzie zasady tworzenia czytelnego i łatwego do modyfikacji kodu. Nie powtarzaj się! Funkcjonalność wspólną dla implementacji powyższych procedur wydziel do procedur pomocniczych.

Wykorzystaj szablon rozwiązania dostępny na SKOS. Nie usuwaj formy `provide` z szablonu! Plik z rozwiązaniem, nazwany `solution.rkt`, zgłoś przez system Web-CAT (dostęp przez odnośnik na SKOS) do dnia **14 kwietnia 2022, godz. 6:00**.