Wybrane elementy praktyki projektowania oprogramowania
Zestaw 3

Javascript - obiekty, tablice, programowanie funkcyjne

2025-10-21

Liczba punktów do zdobycia: **9/27**
Zestaw ważny do: 2025-11-04

1. (**1p**) Napisać jeszcze raz rekurencyjną implementację funkcji `fib(n)` ale tym razem użyć pokazanej na wykładzie techniki memoizacji. Porównać czasy obliczeń z poprzednimi implementacjami.

   Czy da się (a jak tak to jak) mieć taką wersję, która memoizuje nie tylko bieżącą wyliczaną wartość ale również - wszystkie biorące udział w obliczeniach? Czyli - żeby wyliczenie przykładowo `fib(41)` zapamiętało również wartość funkcji dla 40, 39, ...

2. (**1p**) Zaimplementować funkcję `divisors(n)`, która dla zadanego parametru (liczba naturalna) zwraca tablicę zawierającą wszystkie jego dzielniki. Przykładowo:

   - `divisors(10) = [2, 5]`
   - `divisors(20) = [2, 2, 5]`

   Przygotować **dwie** wersje implementacji: imperatywną i funkcyjną.

   W wersji imperatywnej, użyć pętli i napełniania tablicy wyniku za pomocą `push`.

   W wersji funkcyjnej nie wolno używać pętli ani operacji przypisania, wolno natomiast użyć pomocniczej funkcji (może być zdefiniowana wewnątrz funkcji `divisors`) oraz rekurencji. Funkcja wewnętrzna może (ale nie musi) używać omówionego na wykładzie stylu *accumulator passing style*.

3. (**1p**) Poniżej pokazano przykład funkcji, która w swoim dokmnięciu "łapie" zmienną lokalną w sposób niekoniecznie zgodny z oczekiwaniami.

```
function createFs(n) { // tworzy tablicę n funkcji
   var fs = []; // i-ta funkcja z tablicy ma zwrócić i
   for ( var i=0; i<n; i++ ) {
     fs[i] =
        function() {
           return i;
        };
   };
   return fs;
}

var myfs = createFs(10);

console.log( myfs[0]() ); // zerowa funkcja miała zwrócić 0
console.log( myfs[2]() ); // druga miała zwrócić 2
console.log( myfs[7]() );

// 10 10 10              // ale wszystkie zwracają 10!?
```

Jednym ze sposobów skorygowania tego nieoczekiwanego zachowania jest zastąpienie `var` przez `let`. Wyjaśnić dlaczego tak jest.

Wskazówka: jak działa `let`? Jak napisać ten kod z `var` tak żeby zadziałał? Wskazówka (znacznie ułatwiająca rozwiązanie): rozszerzenie języka o `let` pojawiło się stosunkowo niedawno i nie bylo obsługiwane przez starsze przeglądarki. Znaleźć jakiś onlineowy konwerter ES6 do ES5 (standard nieobsługujący `let`) i zobaczyć jak transpiluje się `let` na `var` w tym przypadku.

4. (**2p**) Omówiona na wykładzie funkcja potoku (`pipe`) współpracuje z pomocniczymi operatorami, kilka z nich pokazano na wykładzie.

   Dodać nowy operator, `groupBy`. Parametrem operatora powinna być funkcja wyboru **klucza grupującego**. Funkcja ta zamienia element listy na wartość, która służy do grupowania. Wynikiem działania operatora grupowania dla tablicy jest tablica tablic. Każda tablica w tej tablicy tablic zawiera te elementy oryginalnej tablicy, które mają tę samą wartość funkcji wyboru klucza grupującego. Dodatkowo, każda tablica zawierająca pogrupowane elementy ma właściwość `key`, która jest wartością klucza danej grupy.

   Przykładowo, wynikiem grupowania

```
pipe(
[
  { name: 'jan', surname: 'kowalski' },
  { name: 'jan', surname: 'malinowski' },
  { name: 'tomasz', surname: 'baranowski' },
  { name: 'jan', surname: 'nowak' },
  { name: 'tomasz', surname: 'kochanowski' }
],
groupBy( e => e.name ),
);
```

   powinna być tablica tablic

```
[
  [
    { name: 'jan', surname: 'kowalski' },
    { name: 'jan', surname: 'malinowski' },
    { name: 'jan', surname: 'nowak' }
  ],
  [
    { name: 'tomasz', surname: 'baranowski' },
    { name: 'tomasz', surname: 'kochanowski' }
  ]
]
```

   w której dodatkowo pierwsza tablica ma właściwość `key` równą `jan`, a druga `tomasz`.

   Wskazówka: `Object.groupBy` to gotowy algorytm grupujący dla tablicy i selektora. Wynikiem tego grupowania jest jednak nie tablica tablic tylko obiekt zawierający właściwości odpowiadające grupom. Wystarczy więc zamienić ten obiekt na tablicę tablic. Zbiór klucz obiektu można pobrać przez `Object.keys` a konwersja tego zbioru kluczy na tablicę tablic to zwykłe `map`.

   W dalszej części zadania, zmodyfikować operator `sort`. Aktualnie jego implementacja przyjmuje jako parametr funkcję sortującą wymagającą dwóch parametrów, na przykład

```
...
sort( (n,m) => n-m )
```

bo wewnętrznie, potokowy `sort` używa metody `sort` dla tablicy, a tamta metoda wymaga takiej właśnie definicji funkcji sortującej. W nowej, zmodyfikowanej wersji `sort`, funkcja sortująca powinna przyjmować **jeden** parametr, funkcję wyboru wartości **klucza sortowania** dla elementu tablicy.

Czyli, zamiast pisać

```
...
sort( (person1,person2) => person1.name.localeCompare(person2.name) )
```

programista chciałby napisać

```
...
sort( person1 => person.name )
```

Wszystkie elementy potoku, operator `groupBy` oraz zmodyfikowany operator sortowania, zademonstrować na poniższym przykładzie, w którym rejestr adresów IP (na przykład rejestr adresów IP użytkowników odwiedzających witrynę internetową), jest grupowany, sortowany po długości grupy (czyli liczbie odwiedzin) malejąco, wybierane są 3 pierwsze grupy a następnie z każdej grupy tworzony jest nowy obiekt zawierający klucz grupy (czyli adres IP) oraz długość grupy.

```
pipe(
  [
    { ip: '192.168.0.1' },
    { ip: '192.168.0.1' },
    { ip: '192.168.0.2' },
    { ip: '192.168.0.2' },
    { ip: '192.168.0.3' },
    { ip: '192.168.0.17' },
    { ip: '192.168.0.1' },
  ],
  groupBy( e => e.ip ),
  sort( g => -g.length ),
  take(3),
  map( a => ({ ip: a.key, count: a.length }) )
);
```

Wynikiem działania powyższego powinno być

```
{ip: '192.168.0.1', count: 3}
{ip: '192.168.0.2', count: 2}
{ip: '192.168.0.3', count: 1}
```

5. (**1p**) Poniżej pokazano definicję prostego iteratora

```
function createGenerator() {
    var _state = 0;
    return {
        next : function() {
            return {
                value : _state,
                done : _state++ >= 10
            }
        }
    }
}
```

i jej użycie w obiekcie umożliwiające iterowanie jego zawartości za pomocą `for-of`

```
var foo = {
    [Symbol.iterator] : createGenerator
};

for ( var f of foo )
    console.log(f);
```

Pokazać jak sparametryzować definicję tego generatora czyli formalnie - zastąpić stałą `10`, która pojawia się w ciele metody `createGenerator` przez parametr. Zdefiniować kilka różnych obiektów `foo1`, `foo2` z generatorami zainicjowanymi różnymi wartościami argumentów.

6. (**1p**) Zarówno iteratory jak i generatory mogą być "nieskończone", czyli zawsze zwracać kolejną wartość. Zaimplementować takie nieskończone generatory dla **liczb fibonacciego**: zwykły iterator (zwracający obiekt z funkcją `next`) oraz generator (czyli funkcję wewnętrznie używającą `yield` do zwracania kolejnych wartości).

```
function fib() {
   ...
   return {
      next : function() {
         ...
         return {
            value : ...,
            done  : ...
         }
      }
   }
}

function *fib() {
  ...
  ... yield ...
}
```

W obu przypadkach możliwe jest iterowanie się po kolejnych wartościach za pomocą pokazanej na wykładzie konstrukcji

```
var _it = fib();
for ( var _result; _result = _it.next(), !_result.done; ) {
    console.log( _result.value );
}
```

Czy w którymś z przypadków możliwe jest iterowanie się po kolejnych wartościach za pomocą `for-of`:

```
for ( var i of fib() ) {
    console.log( i );
}
```

7. (**2p**) Próba iterowania nieskończonych iteratorów/generatorów takich jak w poprzednim zadaniu powoduje problem - taki nieskończony iterator/generator zawsze zwraca kolejną wartość i naiwne iterowanie nigdy się nie kończy.

   Pokazać jak rozwiązać ten problem za pomocą dodatkowej funkcji generującej, która jako argumenty przyjmuje iterator/generator oraz liczbę elementów które powinna zwrócić i zwraca dokładnie taką, skończoną liczbę elementów:

```
function* take(it, top) {
   ... yield ...
}

// zwróć dokładnie 10 wartości z potencjalnie
// "nieskończonego" iteratora/generatora
for (let num of take( fib(), 10 ) ) {
  console.log(num);
}
```

Wiktor Zychla