MP23 @ II UWr 8 marca 2023 r.

Lista zadań nr 3

Zadanie 1.

Zapisz poniższe wyrażenia w równoważnej postaci tak, by znak cytowania występował jedynie w podwyrażeniach reprezentujących *symbole*. Nie obliczaj żadnych podwyrażeń! Możesz użyć procedur `list` i `cons`.

```
'((car (a . b)) (* 2))
‘(,(car '(a . b)) ,(* 2))
'((+ 1 2 3) (cons) (cons a b))
```

Zadanie 2.

Wzorując się na procedurze z wykładu sumującej liczby przy użyciu `foldl`, zdefiniuj procedurę `product`, obliczającą iloczyn elementów listy. Jaką wartość powinien zwracać `product` dla listy pustej?

Zadanie 3. (2 pkt)

Używając modelu podstawieniowego, prześledź wykonanie wyrażeń:

```
((lambda (x y) (+ x (* x y))) 1 2)
((lambda (x) x) (lambda (x) x))
((lambda (x) (x x)) (lambda (x) x))
((lambda (x) (x x)) (lambda (x) (x x)))
```

Możesz założyć, że lambda wyrażenia (wyrażenia postaci `(lambda (...)...)`) obliczają sie do siebie samych, tak jak np. stałe liczbowe.

Zadanie 4.

Złożenie funkcji f i g definiujemy (jak pamiętamy z przedmiotu „Logika dla informatyków”) jako funkcję $x \mapsto f(g(x))$. Zdefiniuj dwuargumentową procedurę

`my-compose`, której wynikiem jest złożenie (jednoargumentowych) procedur przekazanych jej jako argumenty. Używając modelu podstawieniowego, prześledź wykonanie wyrażeń:

```
((my-compose square inc) 5)
((my-compose inc square) 5)
```

Zakładamy, że procedura `square` oblicza kwadrat swojego argumentu, natomiast `inc` – wartość argumentu powiększoną o 1.

Zadanie 5. (2 pkt)

Procedura `(build-list n f)` konstruuje `n`-elementową listę, aplikując `f` do wartości od 0 do $n-1$. Dokładniej:

```
(build-list n f) = (list (f 0) ... (f (- n 1)))
```

Wykorzystaj procedurę `build-list` oraz formę specjalną **`lambda`**, aby napisać następujące procedury:

- `(negatives n)`, zwracającą listę liczb ujemnych od -1 do $-n$,
- `(reciprocals n)`, zwracającą listę odwrotności liczb od 1 do `n` (czyli $1, \ldots, \frac{1}{n}$),
- `(evens n)`, zwracajacą listę pierwszych `n` liczb parzystych,
- `(identityM n)`, zwracającą macierz identycznościową o wymiarach $n \times n$ w postaci listy list:

  ```
  > (identityM 3)
  '((1 0 0) (0 1 0) (0 0 1))
  ```

Zadanie 6. (2 pkt)

Zareprezentuj zbiory wartości Racketowych przy użyciu predykatów charakterystycznych. Zdefiniuj:

- `empty-set` – reprezentacja zbioru pustego,
- `(singleton a)` – zwraca zbiór zawierający wyłącznie element `a` (można wykorzystać predykat `equal?`),

- `(in a s)` – zwraca `#t` gdy `a` należy do zbioru `s`, w przeciwnym wypadku wynikiem jest `#f`,
- `(union s t)` – zwraca sumę zbiorów `s` i `t`,
- `(intersect s t)` – zwraca przecięcie zbiorów `s` i `t`.

Zadanie 7.

Odwracanie kolejności elementów listy można zaimplementować używając procedury `foldr` w następujący sposób:

```
(define (foldr-reverse xs)
  (foldr (lambda (y ys) (append ys (list y))) null xs))
```

Taka implementacja posiada jednak poważną wadę. Zaobserwuj ją przy użyciu następującego wyrażenia:

```
(length (foldr-reverse (build-list 10000 identity)))
```

Jak wiele `cons`-ów tworzy ta procedura dla listy wejściowej długości n? Ile z nich to nieużytki?

Zadanie 8. (2 pkt)

Zdefiniuj alternatywną reprezentację list przy użyciu funkcji jednoargumentowych, które, po zaaplikowaniu do „zwykłej” listy Racketowej, zwracają zwykłą listę będącą konkatenacją listy reprezentowanej oraz listy przekazanej przez parametr. Przykładowo, jeśli `f` reprezentuje listę `'(1 2)`, to:

```
> (f '(3 4))
'(1 2 3 4)
```

Zdefiniuj następujące procedury i wartości:

- `(list->llist xs)` – zwraca funkcję reprezentującą listę Racketową `xs`,
- `(llist->list f)` – zwraca listę Racketową reprezentowaną przez funkcję `f`,
- `llist-null` – zwraca funkcję reprezentującą listę pustą,

- `(llist-singleton x)` – zwraca funkcję reprezentującą listę składającą się z jednego elementu `x`,
- `(llist-append f g)` – dla funkcji `f` oraz `g` reprezentujących pewne listy zwraca funkcję reprezentującą konkatenację tych list.

Zaimplementuj procedurę `foldr-llist-reverse`, analogiczną do `foldr-reverse`. Procedura ta powinna przyjmować jako argument oraz zwracać zwykłą listę, ale wykonywać obliczenia przy użyciu reprezentacji opisanej powyżej (w szczególności powinna używać `llist-append` zamiast `append`). Jaka jest wydajność tej procedury w porównaniu do `foldr-reverse`?